ProxyHorizon

2026 EDITION

A PROXYHORIZON COURSE

JavaScript
Beyond the Browser

From Fundamentals to APIs, Scraping & Automation



- Zero to modern JavaScript: async, modules and TypeScript
- Runnable Node.js 26 code with real output, plus a practice lab
- Three projects: an API CLI, a polite scraper and a bot

BEFORE YOU BEGIN

About This Book

A course, not a reference: it starts at your first line of code and ends with tools that run without you. Every example runs on your own computer, next to the output it prints.

Who It's For

New to code

Plain words, small steps and results you can see.

START WITH Chapter 1, in order, doing every exercise

Front-end developers moving to Node.js

You know the browser; now learn the runtime.

START WITH a skim of 1–6, then 7, 9 and Part 3

Founders, marketers and analysts

Data collection and reports that run on their own.

START WITH Chapters 1–5, 10, 12, 13, then Project 3

Developers new to scraping

Working Node.js code and the proxy layer done right.

START WITH a skim of Part 1, then 7, 9 and 11–16

How to Read It

Type the code rather than pasting it, run it and compare your output with the book's. Each chapter ends with three exercises (answers are in the appendix), and from Chapter 11 on the examples use the Lab Shop, a practice server you build yourself. Watch for these signposts:

- **Try it** — Code and its output
- **Jargon buster** — A term in plain words
- **Common mistake** — The trap and the fix
- **Going further** — Optional extras
- **Coming from Python** — For Python developers
- **Play fair** — Responsible use
- **Exercises** — Three per chapter
- **Lab badge** — Needs the Lab Shop

What You Need

- A computer with Windows, macOS or Linux
- Node.js 26 LTS (Chapter 1 shows you how)
- A free code editor, such as VS Code
- No paid accounts. Proxies are optional.

2026 Edition. Written for Node.js 26 LTS and ECMAScript 2024–2026. The examples were tested on Node.js 26 in November 2026, and the book flags the few that can't run on a laptop. IPs, timings and GitHub numbers will differ for you.

Domains like `example.com` are placeholders. Nothing here is legal advice. © 2026 ProxyHorizon. Share it freely, unchanged and with credit.

YOUR MAP

Contents

From First Line to Real Tools — What you'll build, and how the Lab Shop works — 04

PART 1 Fundamentals — Chapters 1–5

01 How JavaScript Runs — Install Node.js and run your first program — 05

02 Values, Variables and Operators — Types, let and const, numbers and strings — 10

03 Control Flow and Functions — Decisions, loops and reusable functions — 15

04 Arrays, Objects and Array Methods — Lists, records, and map, filter and reduce — 21

05 Errors, Debugging and Testing — Read errors, handle them and write tests — 28

PART 2 Modern JavaScript — Chapters 6–9

06 Scope, Closures and Classes — Where names live, closures, this and classes — 32

07 Asynchronous JavaScript — The event loop, promises and async/await — 37

08 Iterators, Generators, JSON and Regex — Lazy data, JSON and patterns in text — 44

09 Modules, Packages and TypeScript — import, npm and a first look at types — 49

PART 3 Beyond the Browser — Chapters 10–16

10 Node.js Core and Command-Line Tools — Files, streams and your own CLI — 54

11 Build a Small API Server — HTTP from the inside: build the Lab Shop — 61

12 Calling REST and GraphQL APIs — Auth, pagination, rate limits and retries — 68

13 Scraping Web Pages with Cheerio — Polite scraping, parsing and SQLite — 74

14 Headless Browsers with Playwright — Pages that need JavaScript to show data — 81

15 Proxies, Concurrency and Scheduling — Rotation, rate control and cron — 85

16 Three Projects to Build and Ship — An API CLI, a scraper and a bot — 92

Appendix — Answers, cheat sheets, checklists and a glossary — 99

INTRODUCTION

From First Line to Real Tools

It's 7:30 on a Monday. Before you open your laptop, a few dozen lines of JavaScript have already checked your site's key pages for errors, slow responses and missing titles, and posted a summary in your team chat. You'll build that bot in Chapter 16.

Most JavaScript courses stop at the browser. This one starts at zero and keeps going: with Node.js, the same language reads files, runs servers, calls APIs, collects public web data and runs on a schedule. You don't need any programming experience. Each chapter builds on the one before, explains every new term and shows the real output of every example.

By the End, You'll Be Able To

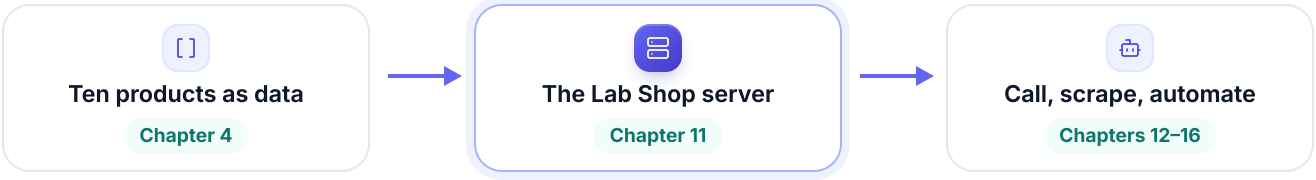
- Install Node.js 26, run scripts, read error messages and debug with confidence.
- Reshape real data with `map`, `filter`, `reduce` and `Object.groupBy`.
- Build command-line tools and a small API server with Node's built-in modules.
- Scrape pages politely with Cheerio and drive a headless browser with Playwright.
- Write modern JavaScript: functions, arrays and objects, classes, modules and destructuring.
- Write async code that waits well, runs in parallel, times out and recovers.
- Call REST and GraphQL APIs with auth, pagination, rate limits and retries.
- Use proxies, control concurrency, store data in SQLite and schedule jobs.

Play fair

Scraping and automation touch other people's servers, and sometimes personal data. This book shows you how to stay on the right side of robots.txt, terms of service, rate limits and privacy law, and it gives you a safe place to practice first. This isn't legal advice.

How the Lab Shop Works

In Chapter 4 you meet ten outdoor products as plain data. In Chapter 11 they become the Lab Shop, a small server on your own computer at `localhost:3000`, which you then call, scrape and automate. The Lab Shop examples run offline with predictable output, and the few that go online say so.



01



PART 1 · FUNDAMENTALS

How JavaScript Runs

Install Node.js, run your first program and see where JavaScript actually runs, so every chapter after this one is hands-on.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

1.1 One language, two homes

1.2 Set up Node.js and your editor

1.3 Your first program

1.4 Read your first error message

1.1 One Language, Two Homes

JavaScript is the language of the web: every major browser runs it. But a language is only text. Something has to read that text and carry it out, and that something is an **engine**. Chrome and Edge use one called V8, Firefox uses SpiderMonkey and Safari uses JavaScriptCore.

In 2009 Node.js took V8 out of the browser and gave it a new home: your computer. Same language, different toolbox. In the browser, JavaScript gets the page and the clicks on it. In Node.js, it gets files, the network, other programs and the ability to run a server, which is what the rest of this book puts to work.

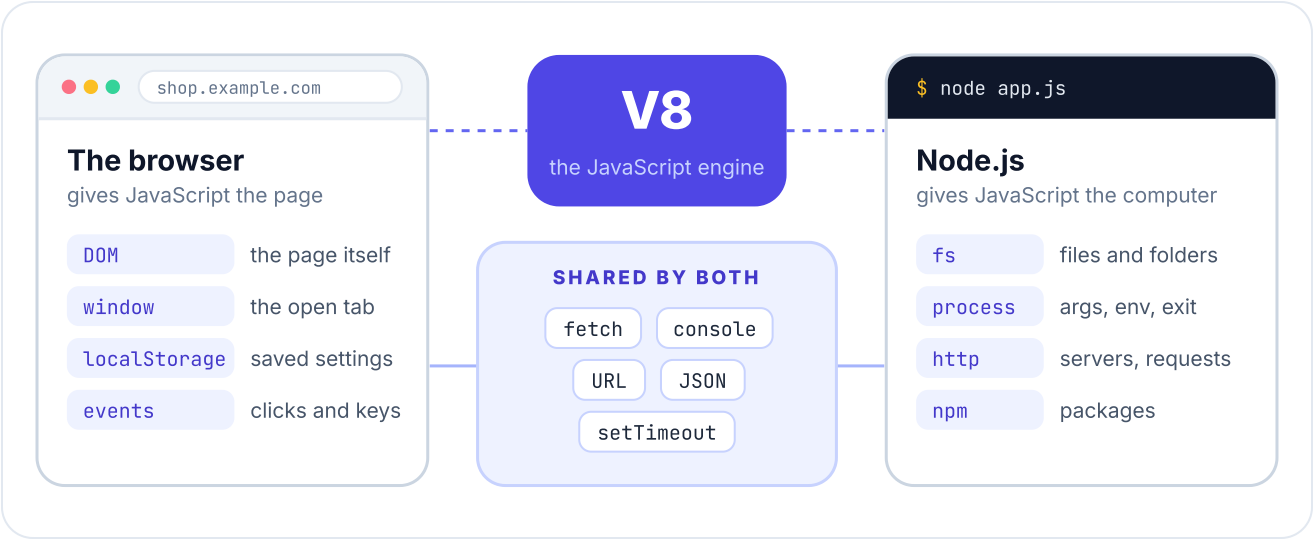


Figure 1.1 One language, two homes. Chrome and Node.js run the same V8 engine. The browser adds the page, Node.js adds the computer, and the tools in the middle work the same in both.

Your code wants to...	In the browser	In Node.js
Show a page and react to clicks	Yes	No, there's no page
Read and write files	Only files the user picks	Yes
Run a web server	No	Yes
Run on a schedule	Only while a tab is open	Yes, with nobody watching
Use npm packages	Yes, through a build tool	Yes, directly

Jargon buster: runtime

The program that runs your code and hands it tools. The browser is one runtime and Node.js is another: both contain an engine, and each adds its own toolbox around it.

1.2 Set Up Node.js and Your Editor

You need two free programs: Node.js, to run your code, and a code editor, to write it. Install the **LTS** version of Node.js, which stands for long-term support: that release line gets bug and security fixes for years. This book uses Node.js 26 LTS, which is supported until April 2029.

1. **Install Node.js 26 LTS.** Download the installer from nodejs.org and pick the LTS version. If you ever need several versions side by side, a version manager such as nvm or fnm switches between them.
2. **Install a code editor.** VS Code is free and runs on Windows, macOS and Linux. Open a terminal inside it with **Terminal → New Terminal**: that's where you'll type every command in this book.
3. **Check the versions.** Run `node -v` and `npm -v`. You want a version starting with `v26`. npm, Node's package manager, comes with it.
4. **Make a course folder.** Run `mkdir js-course`, then `cd js-course` to step inside it. Every file in this book goes here.
5. **Create `package.json`.** Run `npm init -y`, then `npm pkg set type=module`. The second command tells Node.js your files are modern ES modules, the kind this book writes. Chapter 9 explains the difference.

TERMINAL

```
$ node -v
v26.0.0
$ npm -v
11.12.1
$ mkdir js-course
$ cd js-course
$ npm init -y
Wrote to /Users/you/js-course/package.json:

{
  "name": "js-course",
  "version": "1.0.0",
  ... seven more lines ...
  "type": "commonjs"
}
$ npm pkg set type=module
```

Tested on Node.js 26.0.0, which came with npm 11.12.1. The installer gives you the newest v26 release (v26.13.1 with npm 11.20.0 in November 2026); any v26 works. `npm pkg set` prints nothing when it succeeds, and it changes `"commonjs"` to `"module"`.

> ⚠ **Common mistake**
>
> Running an old Node.js. Node 20 reached end of life on 30 April 2026, so it no longer gets security fixes, and several examples in this book need newer features. If `node -v` prints anything below `v26`, install the LTS again or switch with your version manager.

1.3 Your First Program

Create a file called `hello.js` in your course folder, type this in and save it. Then run it from the terminal with `node` and the file name.

hello.js

```
// hello.js: your first program
/* Node.js skips comments.
   They're notes for people. */
console.log('Hello from Node.js!');
console.log('2 + 3 =', 2 + 3);
```

OUTPUT

```
$ node hello.js
Hello from Node.js!
2 + 3 = 5
```

Each line that does something is a **statement**; this book ends each one with a semicolon. `console.log` prints what you give it, with spaces between values, and works out `2 + 3` first. Lines starting with `//`, and text between `/*` and `*/`, are comments.



Figure 1.2 What happens when you type `node hello.js`. The whole trip takes a fraction of a second.

Two habits save time. `node --watch hello.js` runs the file again every time you save it; press `Ctrl`+`C` to stop watching. And plain `node`, with no file, opens the **REPL** (read, evaluate, print, loop): a scratchpad that runs each line as you press `Enter`. Type `.exit` to leave it. For a single expression, `node -p "2 + 3"` prints the answer and quits.

OUTPUT

```
$ node --watch hello.js
Hello from Node.js!
2 + 3 = 5
Completed running 'hello.js'. Waiting
for file changes before restarting...
```

REPL

```
$ node
Welcome to Node.js v26.0.0.
Type ".help" for more information.
> 'js'.toUpperCase()
'JS'
> .exit
```

1.4 Read Your First Error Message

Sooner or later you'll see a wall of red text. Don't panic: an error is a set of directions, not a verdict. Break `hello.js` on purpose by deleting the **e** from `console`, save it as `oops.js` and run it. The first lines of the error (Figure 1.3) usually tell you everything: line 2 of `oops.js` says `consol`, and Node.js has never heard of it.

```
file:///Users/you/js-course/oops.js:2
consol.log('Hello from Node.js!');
^

ReferenceError: consol is not defined
    at file:///Users/you/js-course/oops.js:2:1
    at ModuleJob.run (node:internal/modules/esm/module_job:437:25)
    ...

Node.js v26.0.0
```

1. **Where:** file and line.
2. **Caret:** the exact spot.
3. **Type:** an unknown name.
4. **Message:** what went wrong.
5. **Stack trace:** line 2, col 1.
6. **Node's own code:** skip it.
7. **Version:** yours may be newer.

Figure 1.3 The real error from Node.js 26.0.0, with the path shortened and two internal lines cut.

EXERCISES

Try each one before you check the answer.

1 WARM-UP Run `node -v`, `npm -v` and `node -p "process.platform"`. Note each answer.

2 PRACTICE Write `about-me.js` that prints your name, the year (`new Date().getFullYear()`) and `process.version` on three lines.

3 CHALLENGE Misspell `console`, leave a string unclosed, then call something that isn't a function. Name the error type each mistake prints.

Answers: page 99

KEY TAKEAWAYS

- An engine runs your code. The browser adds the page; Node.js adds the computer.
- Use Node.js 26 LTS and one course folder whose `package.json` says `"type": "module"`.
- Run files with `node file.js`, add `--watch` to re-run on save, try one-liners in the REPL.
- Read errors top to bottom: file and line, type, message, then your own stack frames.

Go deeper: How browsers work · User-Agent Parser · Browser automation

02



PART 1 · FUNDAMENTALS

Values, Variables and Operators

Learn the building blocks every program is made of: values, the names you give them and the operators that combine them.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

2.1 Values and types

2.2 Variables with `let` and `const`

2.3 Operators and comparisons

2.4 Numbers without surprises

2.5 Strings and template literals

2.1 Values and Types

Everything a program works with is a **value**: a price, a product name, a yes or no. Every value has a **type**, which decides what you can do with it. JavaScript has seven simple types, called **primitives**, plus objects for everything bigger. The `typeof` operator tells you which one you're holding.

string `'Sun Hat'` `typeof → 'string'`	**number** `24.5` `typeof → 'number'`	**bigint** `24n` `typeof → 'bigint'`	**boolean** `true` `typeof → 'boolean'`
undefined `typeof → 'undefined'`	**null** `typeof → 'object'`	**symbol** `Symbol()` `typeof → 'symbol'`	**object** `{}` `[]` `typeof → 'object'`

Two of them mean "nothing". `undefined` is what you get when a value was never set; `null` is an empty value someone chose on purpose. `typeof null` says `'object'` because of a mistake in the very first JavaScript, kept so old websites don't break, so test for it with `=== null`.

empty.js

```
// empty.js: two kinds of nothing
let coupon;            // declared, no value yet
const discount = null; // empty on purpose
console.log(coupon, discount);
console.log(typeof coupon, typeof discount);
```

OUTPUT

```
$ node empty.js
undefined null
undefined object
```

2.2 Variables with `let` and `const`

A **variable** is a name for a value, so you can use it again later. Declare it with `const` when the name will always point at the same value, and with `let` when you'll point it somewhere new. Use `const` unless the value changes: it's one less thing to track.



Figure 2.1 A variable is a name tag on a value. `const` fixes the tag to one value; `let` lets you move it to a new one.

Names can use letters, digits, `_` and `$`, but can't start with a digit or be a reserved word such as `return`. They're case-sensitive, so `total` and `Total` differ. Write them in camelCase (`itemsInCart`) and make them say what they hold. Older code uses `var`; it causes bugs Chapter 6 shows, so this book skips it.

⚠ COMMON MISTAKE **Reassigning a `const`**

INSTEAD OF

```
const total = 10;
total = 12;
```

WRITE

```
let total = 10;
total = 12;
```

Node.js stops with `TypeError: Assignment to constant variable.` Use `let` instead.

2.3 Operators and Comparisons

An **operator** combines values into a new one, like `+` in `2 + 3`. Watch `+`: with a string on either side it joins text, so `'5' + 3` is `'53'`, while `'5' * 3` is `15`, because `*` only works on numbers.

Kind	Operators	Example	Result
Arithmetic	`+ - * / % **`	`7 % 3` and `2 ** 10`	`1` and `1024`
Shorthand	`+= -= *= ++ --`	`count += 2`	`count` grows by 2
Comparison	`=== !== < > <= >=`	`89 >= 50`	`true`
Logical	`&& || !`	`true && !false`	`true`
Nullish	`??`	`null ?? 'none'`	`'none'`

⚠ COMMON MISTAKE **Comparing with `==`**

INSTEAD OF

```
0 == '';           // true
'1' == 1;          // true
null == undefined; // true
```

WRITE

```
0 === '';          // false
'1' === 1;         // false
Number('1') === 1; // true
```

`==` converts types first, by rules few people remember. `===` never does: same type and value, or `false`.

An `if` doesn't need a real boolean. JavaScript treats exactly eight values as **falsy** and everything else as truthy, even `'0'`, `'false'`, `[]` and `{}`:

`false` `0` `-0` `0n` `''` `null` `undefined` `NaN`

So `a || b` falls back to `b` for any falsy `a`: `0 || 10` is `10`, even when 0 is a real stock count. `a ?? b` only falls back when `a` is `null` or `undefined`, so `0 ?? 10` stays `0`.

⇄ **Coming from Python**

One number type, so `7 / 2` is `3.5`. Use `===` where Python uses `==`; `&&`, `||` and `!` are `and`, `or` and `not`; `null` or `undefined` stand in for `None`.

2.4 Numbers Without Surprises

Every number, whole or not, is a 64-bit floating-point value. That's usually invisible, but not always:

numbers.js

```
console.log(0.1 + 0.2);
console.log(Number.MAX_SAFE_INTEGER, 2 ** 53 + 1);
console.log(2n ** 64n);
console.log(Number('42'), Number('4 2'), parseInt('42px'));
console.log(parseFloat('3.75 GB'), Number.isNaN(Number('abc')));
const usdFormat = new Intl.NumberFormat('en-US', {
  style: 'currency',
  currency: 'USD',
});
console.log(usdFormat.format(24.5), usdFormat.format(1999));
```

OUTPUT

```
$ node numbers.js
0.30000000000000004
9007199254740991 9007199254740992
18446744073709551616n
42 NaN 42
3.75 true
$24.50 $1,999.00
```

- **Decimals are a hair off.** Most can't be stored exactly in binary, so `0.1 + 0.2` isn't quite `0.3`. Round before you show or compare them.
- **Whole numbers are exact up to a limit.** Past `Number.MAX_SAFE_INTEGER` they round (`2 ** 53 + 1` should end in 3). For bigger ones, add `n` to make a **BigInt**.
- **Text to number.** `Number()` is strict and gives `NaN` (not a number) for messy text; `parseInt` and `parseFloat` stop at the first character that doesn't fit. Check with `Number.isNaN()`: `NaN` never equals itself.
- **Display.** `toFixed(2)` rounds but returns a string: `(19.999).toFixed(2)` is `'20.00'`. `Intl.NumberFormat` formats money properly. `Math.round`, `Math.floor` and `Math.max` do what they say.

Going further

Keep money in whole cents: `19.99 + 4.99` gives `24.979999999999997`, but `1999 + 499` gives exactly `2498`. Many payment APIs take amounts in cents for this reason.

2.5 Strings and Template Literals

Text is a **string**: wrap it in single quotes, double quotes or backticks, and escape a matching quote with a backslash (`'It\'s here'`). Backticks make a **template literal**, which can span lines and drop in values with `${}`. Strings never change in place; methods return new ones.

With `s = 'Trail Runner'`	Result
`s.length`	`12`
`s.at(-1)`	`'r'`
`s.slice(0, 5)`	`'Trail'`
`s.toLowerCase()`	`'trail runner'`
`s.includes('Run')`	`true`
`s.startsWith('T')`	`true`

More string methods	Result
`s.split(' ')`	`['Trail', 'Runner']`
`['a', 'b'].join('-')`	`'a-b'`
`s.replaceAll('r', 'R')`	`'TRail RunneR'`
`'7'.padStart(3, '0')`	`'007'`
`'  hi  '.trim()`	`'hi'`
`s.indexOf('R')`	`6`

slug.js

```
// slug.js: build a product URL from its name
const name = 'Head Lamp';
const words = name.toLowerCase().split(' ');
const slug = words.join('-');
console.log(slug);
console.log(`https://shop.example.com/${slug}`);
console.log(`${name}: ${name.length} characters`);
```

OUTPUT

```
$ node slug.js
head-lamp
https://shop.example.com/head-lamp
Head Lamp: 9 characters
```

EXERCISES

Try each one before you check the answer.

1. **WARM-UP** Predict, then check in the REPL: `typeof null`, `typeof []`, `0.1 + 0.2 === 0.3`, `'5' + 3` and `'5' * 3`.
2. **PRACTICE** Format `1234.5` as `$1,234.50` with `Intl.NumberFormat`.
3. **CHALLENGE** Turn `'  Trail Runner  '` into `'trail-runner'`. Watch the spaces at both ends.

Answers: page 99

KEY TAKEAWAYS

- Every value has a type; `typeof` tells you which, with `null` as the odd one out.
- Use `const` by default and `let` only for values that change. Skip `var`.
- Compare with `===`, and use `??` when 0 or an empty string is a real answer.
- Round decimals for display, keep money in cents and format it with `Intl.NumberFormat`.

Go deeper: URL Encoder · Base64 tool · Timestamp Converter

03



PART 1 · FUNDAMENTALS

Control Flow and Functions

Make your programs decide and repeat, then package that logic into functions you can reuse anywhere.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

3.1 Making decisions

3.2 Loops

3.3 Functions

3.4 Arrow functions and functions as values

3.5 Write functions you can trust

3.1 Making Decisions

An `if` runs a block of code, the part between `{` and `}`, only when its condition is truthy. Add `else if` for more questions and `else` for "none of the above". Figure 3.1 shows a shipping rule as questions in order.

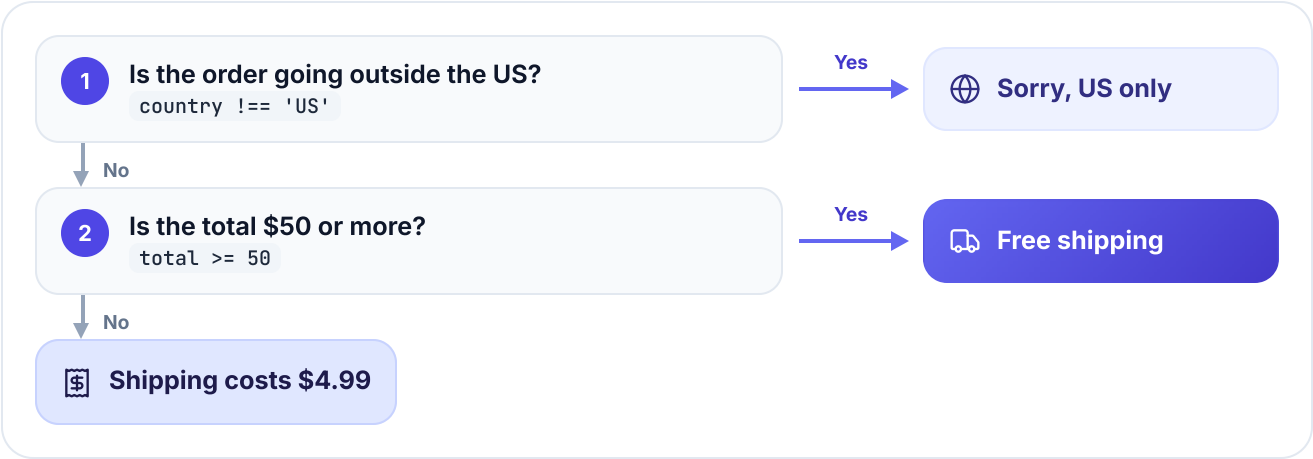


Figure 3.1 An `if`, an `else if` and an `else`: JavaScript asks each question in turn and stops at the first yes.

Here's the same rule in code, next to a `switch`, which compares one value against fixed cases. End each case with `break`, or the code falls through into the next one; stacked cases share an outcome.

shipping.js

```
// shipping.js: pick a shipping price
const total = 42.5;
const country = 'US';
if (country !== 'US') {
  console.log('Sorry, US only for now');
} else if (total >= 50) {
  console.log('Free shipping');
} else {
  console.log('Shipping costs $4.99');
}
```

```
$ node shipping.js
Shipping costs $4.99
```

OUTPUT

ships.js

```
// ships.js: switch handles fixed cases
const day = 'Sat';
switch (day) {
  case 'Sat':
  case 'Sun':
    console.log('Ships on Monday');
    break;
  default:
    console.log('Ships today');
}
```

```
$ node ships.js
Ships on Monday
```

OUTPUT

For a choice between two values, the **ternary** operator fits on one line: `stock > 0 ? 'In stock' : 'Sold out'` reads "if stock is above 0, this, otherwise that".

Coming from Python

Braces mark blocks, not indentation, and it's `else if`, not `elif`. Empty arrays and objects are truthy, so test `list.length === 0`, not `!list`.

3.2 Loops

A **loop** repeats a block. The classic `for` loop packs its whole plan into one line, shown in Figure 3.2: where to start, when to keep going and how to step.

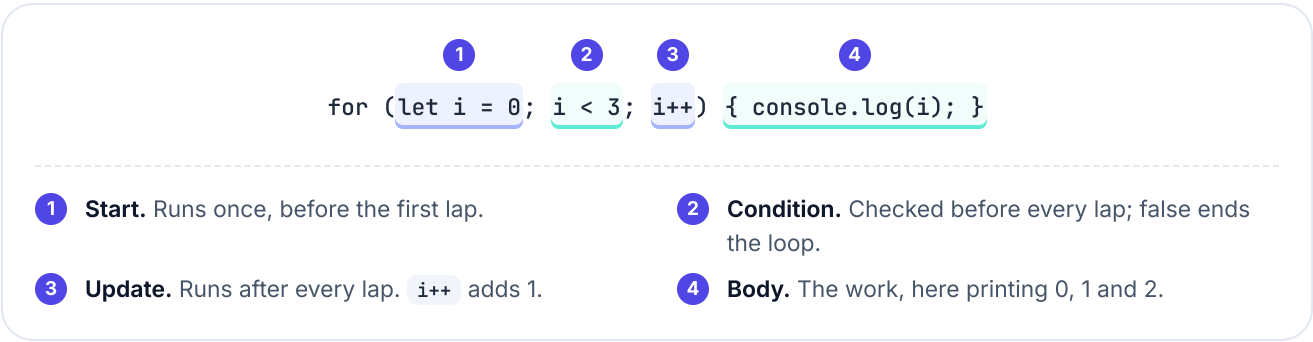


Figure 3.2 The parts of a `for` loop. Counting starts at 0, so `i < 3` means three laps.

Loop	Use it when	Example
`for`	You need a counter or an index	`for (let i = 3; i > 0; i--)`
`for...of`	You want each item of a list or each letter of a string	`for (const letter of 'JS')`
`while`	You don't know how many laps it will take	`while (saved < 100)`
Array methods	You're transforming a list (Chapter 4)	`prices.map(...)`

loops.js

```
// loops.js: three ways to repeat
for (let i = 3; i > 0; i--) {
  console.log(i);
}
console.log('Liftoff!');
for (const letter of 'JS') {
  console.log(letter);
}
let saved = 0;
let weeks = 0;
while (saved < 100) {
  saved += 15;
  weeks++;
}
console.log(`${weeks} weeks to save $${saved}`);
```

OUTPUT

```
$ node loops.js
3
2
1
Liftoff!
J
S
7 weeks to save $105
```

Inside any loop, `break` stops it early and `continue` skips to the next lap. Two traps catch everyone. An **off-by-one** error runs one lap too many or too few, usually `<=` where you meant `<`. And a condition that never turns false loops forever: press `Ctrl` + `C` to stop it, then check that something inside the loop moves it toward the end.

3.3 Functions

So far every line has run once, top to bottom. A function lets you name a piece of work and run it as often as you like.

CONCEPT

Function

A named, reusable piece of code. It takes inputs, called **parameters**, does one job and hands back a result with `return`. Running it is a **call**, and the values you pass in a call are its **arguments**.

THINK OF IT AS

A recipe card: the ingredients change each time, the steps stay the same.

IN CODE

```
shippingCost(42.5);  // returns 4.99
```

```
function shippingCost(total) {
  if (total >= 50) return 0;
  return 4.99;
}
```

1. **Keyword.** Starts a function declaration.
2. **Name.** A verb phrase that says what it does.
3. **Parameter.** The input, named inside the function.
4. **Return.** Hands a value back and stops right there.

Figure 3.3 The parts of a function declaration. Everything between the braces is the body.

order.js

```
// order.js: add shippingCost from above first
function orderTotal(...prices) {
  let sum = 0;
  for (const price of prices) sum += price;
  return sum + shippingCost(sum);
}
function greet(name = 'friend') {
  return `Hi, ${name}!`;
}
console.log(orderTotal(19, 24));
console.log(orderTotal(89, 14.5));
console.log(greet(), greet('Ada'));
```

OUTPUT

```
$ node order.js
47.99
103.5
Hi, friend! Hi, Ada!
```

A **rest parameter**, `...prices`, gathers any number of arguments into a list, so `orderTotal` takes two prices or twenty. A **default parameter**, `name = 'friend'`, kicks in when the caller leaves that argument out. And `orderTotal` reuses `shippingCost` rather than repeating it.

3.4 Arrow Functions and Functions as Values

In JavaScript a function is a value, like a number or a string: you can store it in a variable, pass it around and return it. Arrow functions are the short way to write one. Here's the price formatter this book uses everywhere, shrinking in three steps:

Step	Code
Declaration	`function usd(n) { return `$${n.toFixed(2)}`; }`
1. Store it	`const usd = function (n) { return `$${n.toFixed(2)}`; };`
2. Arrow	`const usd = (n) => { return `$${n.toFixed(2)}`; };`
3. Concise body	`const usd = (n) => `$${n.toFixed(2)}`;`

Step 3 has a **concise body**: with no braces, the arrow returns its expression. You may drop the parentheses around one parameter, but this book keeps them. Chapter 11's Lab Shop uses this exact `usd`.

A function you pass to another is a **callback**, like the arrow `setTimeout` calls a second later below. And `percentOff` returns a new function: Chapter 6 calls that a closure.

arrows.js

```
// arrows.js: functions are values too
const usd = (n) => `$${n.toFixed(2)}`;
const applyTo = (price, format) => format(price);
const percentOff = (p) => (price) => price * (100 - p) / 100;
const tenOff = percentOff(10);
console.log(usd(89), applyTo(14.5, usd));
console.log(usd(tenOff(89)));
setTimeout(() => console.log('A second later'), 1000);
```

OUTPUT

```
$ node arrows.js
$89.00 $14.50
$80.10
A second later
```

The last line appears after a one-second pause. Chapter 4's `map` and `filter` take callbacks the same way.

COMMON MISTAKE **Returning an object from an arrow**

INSTEAD OF

```
const toItem = (name) => { name: name };
toItem('Sun Hat'); // undefined
```

WRITE

```
const toItem = (name) => ({ name });
toItem('Sun Hat'); // { name: 'Sun Hat' }
```

After `=>`, a brace starts a function body, not an object. Wrap the object in parentheses.

3.5 Write Functions You Can Trust

Good functions are small, do one job and are named with a verb that says what they do: `shippingCost`, `formatPrice`, `isInStock`. Check bad input at the top and return early, as in `if (qty <= 0) return 0;`. These **guard clauses** keep the main path from getting buried in nested `if`s, so the function reads top to bottom: bad cases out, then the rule.

Then keep most of your functions **pure**, and gather the ones with side effects in a few clear places:

Pure

Same input, same output, nothing else touched. `shippingCost(42.5)` is 4.99 today and next year. Easy to test, safe to reuse.

Side effects

Changes the world outside the function: prints, writes a file, calls the network. You need them; just don't hide them.

EXERCISES Try each one before you check the answer.

1	WARM-UP	FizzBuzz from 1 to 15: `Fizz` for multiples of 3, `Buzz` for 5, `FizzBuzz` for both, else the number.
2	PRACTICE	Extend Figure 3.3 to `shippingCost(total, country = 'US')`: a flat $14.99 outside the US.
3	CHALLENGE	Write `tryTimes(task, times)`: call `task()` until it returns `true` and return the attempt number, or `-1` if every try fails.

Answers: page 100

KEY TAKEAWAYS

- `if`, `else if` and `else` ask questions in order; `switch` suits fixed cases.
- Use `for` for counters, `for...of` for items and `while` when the count is unknown.
- Functions take parameters and return values, with optional defaults and a rest parameter.
- Functions are values: pass them as callbacks, and wrap an object an arrow returns in parentheses.
- Small, pure, well-named functions with guard clauses are the easiest to trust.

Go deeper: E-commerce price monitoring · What is web scraping?

04



PART 1 · FUNDAMENTALS

Arrays, Objects and Array Methods

Store and reshape real data: lists, records and the `map`, `filter` and `reduce` trio that does most of the work in data scripts.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

4.1 Arrays

4.2 Objects

4.3 Copies, destructuring and spread

4.4 `Map` and `Set`

4.5 Transform data with `map`, `filter` and `find`

4.6 `reduce`, sorting and grouping

4.1 Arrays

An **array** is an ordered list of values in square brackets. Each item has a position, its **index**, and counting starts at 0, so the first item is `sizes[0]`. `length` tells you how many items there are, and `at(-1)` counts from the end.

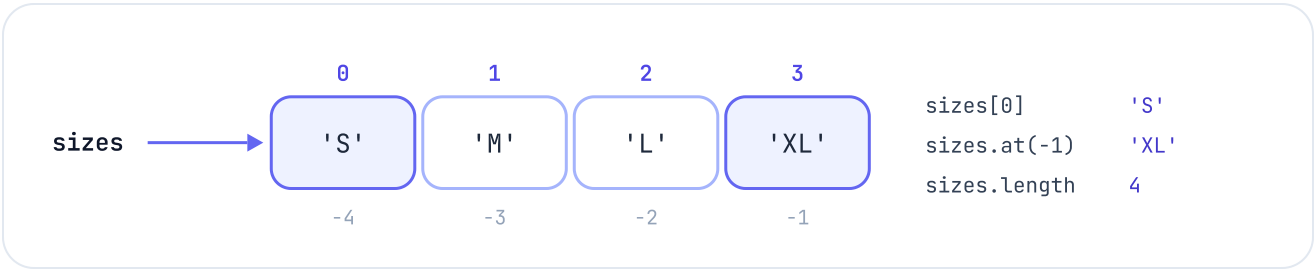


Figure 4.1 Indexes start at 0, so the last item is at `length - 1`. `at()` also takes negative numbers that count back from the end.

arrays.js

```
// arrays.js: an ordered list of values
const sizes = ['S', 'M', 'L'];
console.log(sizes[0], sizes.length, sizes.at(-1));
sizes.push('XL');     // add to the end
sizes.unshift('XS');  // add to the start
console.log(sizes, sizes.indexOf('L'));
const middle = sizes.slice(1, 4); // a copy
sizes.splice(0, 1);  // remove 1 item at index 0
console.log(middle, sizes);
```

OUTPUT

```
$ node arrays.js
S 3 L
[ 'XS', 'S', 'M', 'L', 'XL' ] 3
[ 'S', 'M', 'L' ] [ 'S', 'M', 'L', 'XL' ]
```

`push`, `pop`, `shift`, `unshift` and `splice` **mutate** the array: they change it in place. `slice` returns a copy. `includes` answers yes or no, and `indexOf` gives a position, or `-1` if the item isn't there.

4.2 Objects

An array is a list; an **object** is a record: named values, called **properties**, in curly braces. Read one with a dot (`boot.price`) or with brackets and a string (`boot['price']`), which also works with a variable.

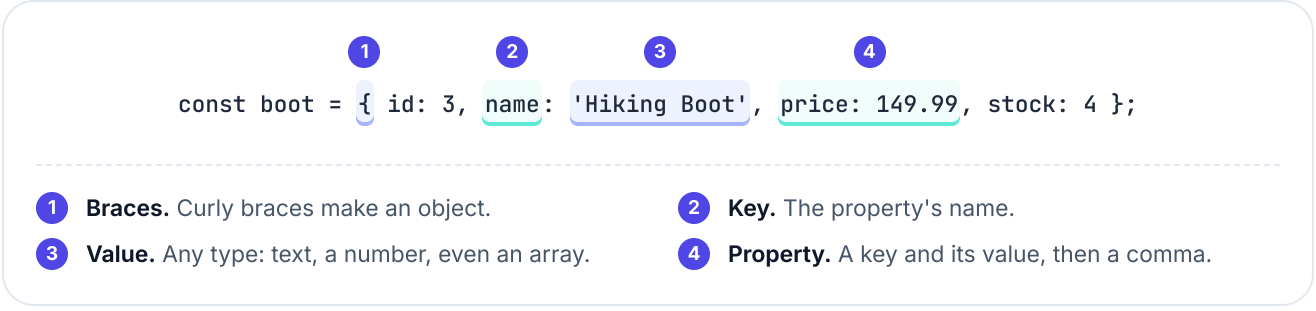


Figure 4.2 The parts of an object literal: one product from the Lab Shop.

Real data is usually an array of objects: a list of records that all have the same fields. Here's the one the rest of this book uses, the ten products of the **Lab Shop**. In Chapter 11 they'll become a real server you call, scrape and automate. Save this file as `products.js` in your course folder:

products.js

```
// products.js: the Lab Shop catalog
export const products = [
  { id: 1, name: 'Trail Runner', price: 89, stock: 12 },
  { id: 2, name: 'Road Racer', price: 120.5, stock: 0 },
  { id: 3, name: 'Hiking Boot', price: 149.99, stock: 4 },
  { id: 4, name: 'Rain Shell', price: 99, stock: 7 },
  { id: 5, name: 'Wool Socks', price: 14.5, stock: 40 },
  { id: 6, name: 'Day Pack', price: 64, stock: 9 },
  { id: 7, name: 'Head Lamp', price: 29.95, stock: 0 },
  { id: 8, name: 'Water Bottle', price: 19, stock: 25 },
  { id: 9, name: 'Trek Poles', price: 79, stock: 3 },
  { id: 10, name: 'Sun Hat', price: 24, stock: 15 },
];
```

`export` lets other files use the list, and one line at the top of any file brings it in: `import { products } from './products.js';`. Copy that line for now; Chapter 9 explains modules properly.

records.js

```
// records.js: one record, then the whole list
import { products } from './products.js';
const boot = products[2];
const key = 'price';
console.log(boot.name, boot[key]);
boot.tag = 'new';
console.log(Object.keys(boot));
const last = products.at(-1);
console.log(products.length, last.name);
```

OUTPUT

```
$ node records.js
Hiking Boot 149.99
[ 'id', 'name', 'price', 'stock', 'tag' ]
10 Sun Hat
```

Assigning a new key adds a property; `delete boot.tag` removes it. `Object.keys`, `Object.values` and `Object.entries` return an object's names, values or `[name, value]` pairs. When a variable has the right name already, `{ name, price }` is short for `{ name: name, price: price }`. Objects can also hold functions, called **methods** (Chapter 6).

4.3 Copies, Destructuring and Spread

Here's the surprise that trips up everyone once. Assigning an object to a second variable doesn't copy it: both names point at the same object, so a change through one shows up through the other. To get a real copy, spread it into a new object with `...`.



Figure 4.3 Left: two names, one object, so `shared.stock = 0` changes `boot` too. Right: spread builds a new object, so `sale` can change on its own.

copies.js

```
// copies.js: references, copies and destructuring
const boot = { name: 'Hiking Boot', price: 149.99, stock: 4 };
const shared = boot;                      // a second name, same object
shared.stock = 0;
const sale = { ...boot, price: 119.99 };  // a new object
console.log(boot.stock, sale.price, boot.price);
const { name, price: cost, color = 'black' } = boot;
console.log(name, cost, color);
const [first, ...rest] = ['S', 'M', 'L'];
console.log(first, rest);
const order = { customer: null };
console.log(order.customer?.address?.city ?? 'no city');
```

OUTPUT

```
$ node copies.js
0 119.99 149.99
Hiking Boot 149.99 black
S [ 'M', 'L' ]
no city
```

Destructuring unpacks values into variables in one line: `price: cost` renames, `color = 'black'` sets a default for a missing property, and `...rest` collects whatever's left. **Optional chaining**, `?.`, stops and gives `undefined` instead of crashing when something along the path is missing, and `??` then supplies a fallback.

⚠ COMMON MISTAKE **Expecting spread to copy everything**

INSTEAD OF

```
const copy = { ...pack };
copy.tags.push('sale');
// pack.tags changed too
```

WRITE

```
const copy = structuredClone(pack);
copy.tags.push('sale');
// pack.tags is untouched
```

Spread copies one level. Arrays and objects inside are still shared; `structuredClone` copies all the way down.

4.4 `Map` and `Set`

Two more containers round out the toolkit. A **Map** stores key-value pairs like an object, but any value can be a key and it knows its size. A **Set** stores each value once, and ES2025 gave it methods for comparing two Sets: perfect for "what changed since yesterday?"

	Object	Map	Set
Holds	Named fields	Key-value pairs	Unique values
Read	`o.id`	`m.get(3)`	`s.has(3)`
Count	`Object.keys(o).length`	`m.size`	`s.size`
Best for	One record	Lookups by ID	"Seen it before?"

With yesterday's IDs `{1, 2, 3, 4}` and today's `{2, 3, 4, 5}`, `today.difference(yesterday)` is `Set(1) { 5 }`, the new arrival, and `yesterday.difference(today)` is `Set(1) { 1 }`, the one that vanished. And `[...new Set(tags)]` de-duplicates a list.

4.5 Transform Data with `map`, `filter` and `find`

Most data scripts boil down to three moves: keep some items, reshape each one, add them up. Each move is an array method that takes a callback; Figure 4.4 follows the Lab Shop's stock through all three.

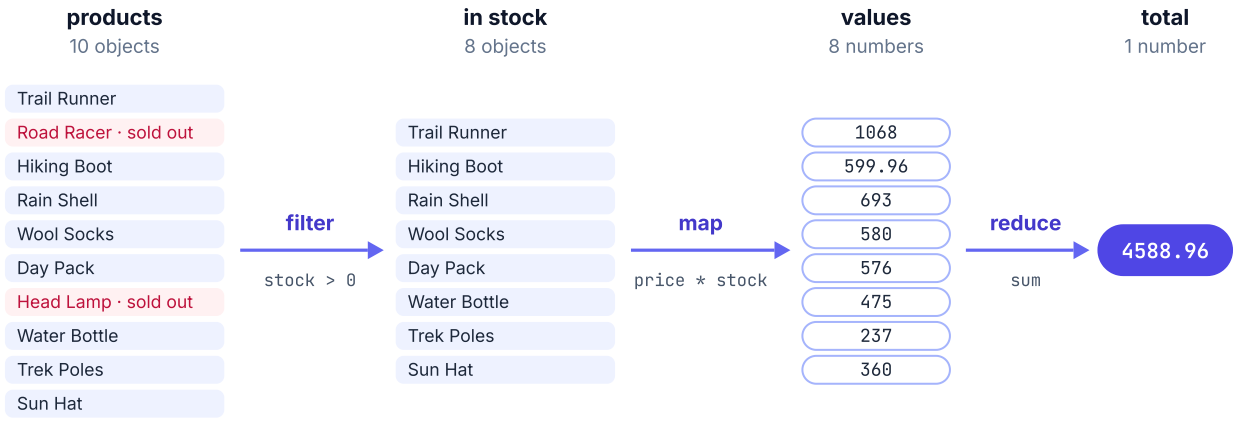


Figure 4.4 Objects, fewer objects, numbers, one number: each method changes the shape of the data.

Coming from Python

Arrays are your lists and objects your dicts, with string keys and dot access. Where you'd write `[p['name'] for p in products if p['price'] < 50]`, JavaScript chains `products.filter((p) => p.price < 50).map((p) => p.name)`.

transform.js

```
// transform.js: map, filter and find
import { products } from './products.js';
const names = products.map((p) => p.name);
const soldOut = products.filter((p) => p.stock === 0);
const boot = products.find((p) => p.name.includes('Boot'));
console.log(names.length, names.slice(0, 3));
console.log(soldOut.map((p) => p.name));
console.log(boot);
```

OUTPUT

```
$ node transform.js
10 [ 'Trail Runner', 'Road Racer', 'Hiking Boot' ]
[ 'Road Racer', 'Head Lamp' ]
{ id: 3, name: 'Hiking Boot', price: 149.99, stock: 4 }
```

`map` returns a same-length array, `filter` the items that pass and `find` the first match or `undefined` (`findLast` searches from the end); `some` and `every` answer yes or no. Chain them freely. Use `forEach` only for side effects such as printing, and never change items inside a `map`: return new ones.

4.6 `reduce`, Sorting and Grouping

`reduce` carries one value, the **accumulator**, through the array and returns what it ends up as: a sum, a count, even an object indexing products by ID. Watch `sort`: it changes the original and compares as text by default, so numbers need a compare function. `toSorted` returns a sorted copy.

report.js

```
// report.js: reduce, sort and group the catalog
import { products } from './products.js';
const items = products.reduce((sum, p) => sum + p.stock, 0);
const priciest = products.toSorted((a, b) => b.price - a.price)[0];
console.log(`${items} items in stock`);
console.log(`Priciest: ${priciest.name}`);
console.log([10, 9, 1].sort(), [10, 9, 1].toSorted((a, b) => a - b));
const band = (p) => (p.price < 50 ? 'budget' : 'premium');
const bands = Object.groupBy(products, band);
console.log(bands.budget.length, bands.premium.length);
console.log(Object.groupBy(['a', 'bb', 'cc'], (s) => s.length));
```

OUTPUT

```
$ node report.js
115 items in stock
Priciest: Hiking Boot
[ 1, 10, 9 ] [ 1, 9, 10 ]
4 6
[Object: null prototype] { '1': [ 'a' ], '2': [ 'bb', 'cc' ] }
```

`Object.groupBy` (ES2024) sorts items into named groups in one call. Node.js prints its result as `[Object: null prototype]` because the object is built without the usual built-in extras, so a key such as `toString` can't clash with them. You use it like any other object. Keep this table nearby until the methods feel familiar:

Method	Returns	Changes the array?
`map`, `filter`, `slice`	A new array	No
`find`, `findLast`	One item, or `undefined`	No
`some`, `every`, `includes`	`true` or `false`	No
`reduce`	One value of any type	No
`toSorted`, `toReversed`, `with`	A changed copy (ES2023)	No
`push`, `pop`, `shift`, `unshift`, `splice`	A length or the removed items	**Yes**
`sort`, `reverse`	The same array, rearranged	**Yes**

EXERCISES Use `products.js` for all three.

1. **WARM-UP** Print the names of the products under $50, using `filter` and `map`.
2. **PRACTICE** Work out the total stock value (price times stock, summed) with `reduce`, and print it formatted as dollars with `Intl.NumberFormat`.
3. **CHALLENGE** Group the products into `inStock` and `soldOut` with `Object.groupBy`, then print the sold-out names A to Z with `toSorted`.

Answers: page 100

KEY TAKEAWAYS

- Arrays are lists indexed from 0; objects are records. Real data is an array of objects.
- Assigning shares an object. Spread copies one level; `structuredClone` copies it all.
- Use a `Map` for lookups by any key and a `Set` for unique values and comparisons.
- `filter`, `map` and `reduce` keep, reshape and combine data without changing the original.
- Prefer `toSorted` to `sort`, and always pass a compare function for numbers.

Go deeper: JSON Formatter · Clean and store scraped data · E-commerce price monitoring

05



PART 1 · FUNDAMENTALS

Errors, Debugging and Testing

Read error messages like a pro, handle failures on purpose, find bugs fast and prove your code works with tests.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

5.1 Reading errors and stack traces

5.2 Throwing and catching

5.3 Debugging tools

5.4 Your first tests with `node:test`

5.1 Reading Errors and Stack Traces

Every Node.js error has the same shape: where it happened, the error type and message, then the **stack trace**, the chain of function calls that led there, newest first. The top frame in your own file is usually the place to look. Here a lookup for a product that doesn't exist fails two calls deep:

TERMINAL

```
$ node cart.js
file:///Users/you/js-course/cart.js:4
  return products.find((p) => p.id === id).price;
                                          ^

TypeError: Cannot read properties of undefined (reading 'price')
    at priceOf (file:///Users/you/js-course/cart.js:4:43)
    at file:///Users/you/js-course/cart.js:6:26
    at ModuleJob.run (node:internal/modules/esm/module_job:437:25)
    ... two more node:internal frames and the Node.js version ...
```

There's no product 99, so `find` returns `undefined`. The highlighted frames are yours: line 4 in `priceOf`, called from line 6.

Type	A typical Node.js message	What usually went wrong
`SyntaxError`	`Unexpected token '}'`	A typo spotted before anything runs: a stray or missing bracket, quote or comma. Check that line and the one above.
`ReferenceError`	`x is not defined`	A name that doesn't exist here: misspelled, or used outside its scope (Chapter 6).
`TypeError`	`Cannot read properties of undefined (reading 'price')`	A value is missing or the wrong kind, often a search that found nothing. Log it, then guard with `?.`
`RangeError`	`Invalid array length`	A number outside the allowed range, such as `new Array(-1)` .

5.2 Throwing and Catching

When your code meets a problem it can't fix, it should **throw**: `throw new RangeError('...')` stops the function and hands the error to its caller. Wrap a risky call in `try` and the error lands in `catch` instead of crashing the program; `finally` runs either way, for cleanup.

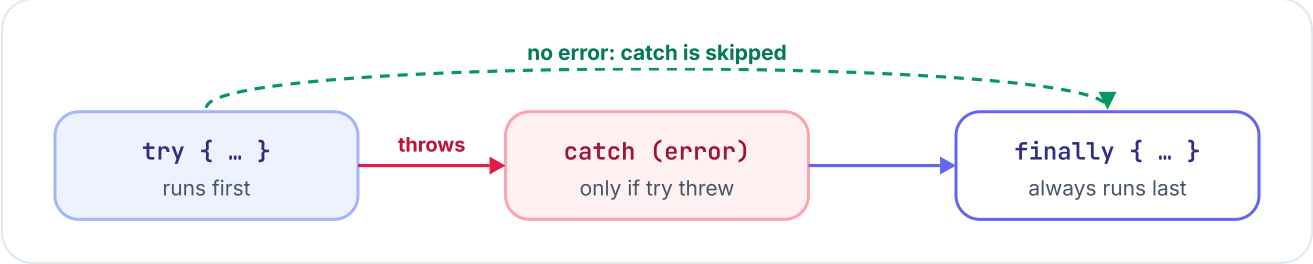


Figure 5.1 The two paths through `try` , `catch` and `finally` . An error nobody catches stops the whole program.

stock.js

```
// stock.js: read stock counts from page text
function parseStock(text) {
  if (text === 'Sold out') return 0;
  const count = parseInt(text);
  if (Number.isNaN(count)) {
    throw new RangeError(`Not a count: ${text}`);
  }
  return count;
}
const texts = ['12 in stock', 'Sold out', 'N/A'];
let checked = 0;
for (const text of texts) {
  try {
    console.log(text, '->', parseStock(text));
  } catch (error) {
    console.log(error.name, error.message);
  } finally {
    checked++;
  }
}
console.log(`Checked ${checked} values`);
```

OUTPUT

```
$ node stock.js
12 in stock -> 12
Sold out -> 0
RangeError Not a count: N/A
Checked 3 values
```

Catch only what you can handle. To add context, rethrow with the original as the **cause**: `throw new Error('Row 7 failed', { cause: error })`. `Error.isError(x)` (ES2026) spots real errors.

> **Common mistake**
> Swallowing errors. An empty `catch {}` hides a bug without fixing it: the data is still wrong, you just won't know. At least report it with `console.error(error)`.

5.3 Debugging Tools

When the output is wrong but there's no error to read, go looking. Three tools cover most bugs:

Console

`console.log` for values, `console.table(rows)` for lists, `console.time` for timing.

Debugger

`node --inspect-brk app.js`, then `chrome://inspect` in Chrome: step line by line.

VS Code

In **Run and Debug**, click beside a line number to set a breakpoint, then step and watch variables.

WHEN YOU'RE STUCK

- [ ] Read the whole error, top to bottom
- [ ] Log the values just before the bad line
- [ ] Shrink it to the smallest example that fails
- [ ] Find the first frame in your own file
- [ ] Halve it: comment out half, does it still break?
- [ ] Explain it out loud, line by line

5.4 Your First Tests with `node:test`

A **test** runs your code and checks the answer after every change. Move `parseStock` into `parse-stock.js`, put `export` in front, and add a test file: `node --test` runs every `.test.js` file.

stock.test.js

```
import { test } from 'node:test';
import assert from 'node:assert/strict';
import { parseStock } from './parse-stock.js';
test('reads the number of items', () => {
  assert.equal(parseStock('12 in stock'), 12);
});
test('rejects text it cannot read', () => {
  const read = () => parseStock('N/A');
  assert.throws(read, RangeError);
});
```

OUTPUT

```
$ node --test
✔ reads the number of items (0.465834ms)
✔ rejects text it cannot read (0.13225ms)
ℹ tests 2
ℹ suites 0
ℹ pass 2
ℹ fail 0
... 4 more summary lines ...
```

`assert.equal` checks a value, `assert.deepEqual` compares arrays and objects, `assert.throws` expects an error. Each ✔ is a pass (your times will differ). Expect 13 instead of 12 and you get ✖, `fail 1` and the reason: `12 !== 13`.

EXERCISES

Try each one before you check the answer.

1 WARM-UP Name each error type: `x is not defined`, `Unexpected token '}'`, `Cannot read properties of undefined (reading 'price')`.

2 PRACTICE Write `parsePrice(text)`: `'$1,299.00'` gives `1299`, unreadable text throws a `RangeError`. Catch it with a friendly message.

3 CHALLENGE Test `parsePrice` with `node:test`: `'$1,299.00'` gives 1299, and `'N/A'` throws.

Answers: page 101

KEY TAKEAWAYS

- Read errors top to bottom: type, message, then the first stack frame in your own file.
- Throw specific errors, catch only what you can handle and never leave a `catch` empty.
- Debug with `console.table`, the inspector or VS Code breakpoints, and halve the problem.
- Tests with `node:test` and `assert` prove your code still works after each change.

Go deeper: JSON Formatter · **For Part 3:** Proxy error codes explained · Handle proxy timeouts and errors

06



PART 2 · MODERN JAVASCRIPT

Scope, Closures and Classes

See which names your code can reach, why functions remember, what `this` really means and when a class is the right tool.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

6.1 Scope: where names live

6.2 Closures: functions that remember

6.3 Understanding `this`

6.4 Classes

6.1 Scope: Where Names Live

Every name you declare lives somewhere. Its **scope** is the part of the code that can see it. Each file is a **module** with its own scope; each function gets one; and each pair of braces, a **block**, gets one for `let` and `const`. The rule that makes it all work: code can look outward, never inward.

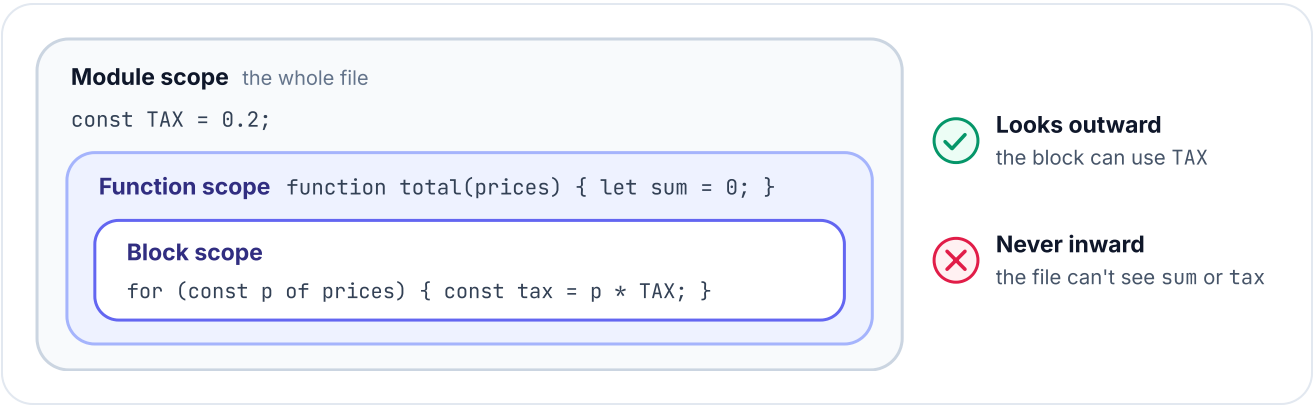


Figure 6.1 Scopes nest like boxes. A name declared in an outer box is visible in every box inside it, never the other way round.

`let` and `const` live in the block that declares them. The old `var` ignores blocks and belongs to the whole function, which is why this book avoids it. A name declared inside can **shadow** an outer one with the same name, which compiles fine and confuses everyone, so pick a new name. And `let` and `const` exist only from their line onward: touch one earlier and Node.js stops with `ReferenceError: Cannot access 'x' before initialization`.

COMMON MISTAKE **Using `var` in a loop with a callback**

INSTEAD OF

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 0);
}
// 3 3 3
```

WRITE

```
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 0);
}
// 0 1 2
```

The callbacks run after the loop ends. `var` gives all three one shared `i`, now 3; `let` gives each lap its own.

6.2 Closures: Functions That Remember

Jargon buster: closure

A function together with the variables it could see where it was created. It keeps them alive and can still use them after the outer function has finished.

Here's why that matters. `makeCounter` below creates `count`, returns an arrow that adds to it, then finishes. Normally `count` would be gone. But the arrow still uses it, so it travels along, as if in a backpack only that function can open.

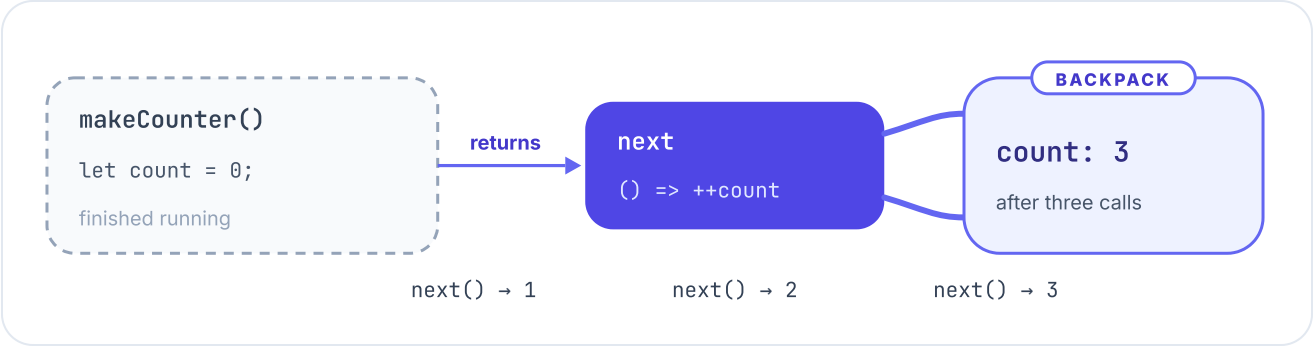


Figure 6.2 The closure backpack. `next` carries `count` with it, and nothing else in the program can reach it.

closures.js

```
// closures.js: functions that remember
function makeCounter() {
  let count = 0;
  return () => ++count;
}
const next = makeCounter();
console.log(next(), next(), next());
const other = makeCounter();
console.log(other(), next());
```

OUTPUT

```
$ node closures.js
1 2 3
1 4
```

Each call to `makeCounter` packs a fresh backpack, so `other` starts at 1 while `next` carries on to 4. Closures are everywhere once you look. `once(fn)` wraps a setup function so it runs only the first time. **Memoizing** caches slow results in a `Map` the function closes over. And Chapter 11's Lab Shop limits each visitor to five calls per second with a function that closes over a `Map` of call counts.

6.3 Understanding `this`

Inside a method, `this` means "the object I was called on", so `shop.greet()` can read `this.name`. The catch: `this` is decided by *how* a function is called, not where it's written:

How it's called	Example	`this` is
As a method	`shop.greet()`	The object before the dot, `shop`
As a plain function	`const f = shop.greet; f()`	`undefined` in a module, so `this.name` throws
Handed to `setTimeout`	`setTimeout(shop.greet, 0)`	Node's timer object, so `this.name` is `undefined`
Arrow function	`() => this`	The `this` of the code around it

COMMON MISTAKE **Losing `this` in a callback**

INSTEAD OF

```
setTimeout(shop.greet, 1000);
// Welcome to undefined
```

WRITE

```
setTimeout(() => shop.greet(), 1000);
// Welcome to Lab Shop
```

Passing `shop.greet` hands over the function without `shop`. Wrap it in an arrow, or pass `shop.greet.bind(shop)`.

6.4 Classes

A **class** is a blueprint for objects that share the same shape and behavior. `new` builds one, running the **constructor** to set it up. A class can **extend** another and inherit everything it has. Figure 6.3 is the custom error Chapters 12 and 16 throw when a request fails.

```
export class HttpError extends Error {
  constructor(status, url) {
    super(`HTTP ${status} for ${url}`);
    this.name = 'HttpError';
    this.status = status;
    this.url = url;
  }
}
```

1. **Class.** A name, capitalized like a type.
2. **Extends.** Inherits everything an `Error` has.
3. **Constructor.** Runs on `new`; `super()` sets up the parent first.
4. **Fields.** Properties set on `this`, the new object.

Figure 6.3 The parts of a class. `new HttpError(404, '/api/products/99')` gives an error whose message reads `HTTP 404 for /api/products/99`.

When should you write one? Use a plain object for data, a function for a single job and a class when many objects share data and the methods that guard it. Classes also give you **private fields**, marked with `#`, that no code outside the class can touch; **getters**, which look like properties but run code; and **static** methods, called on the class itself, handy as factories that build an instance from other data:

classes.js

```
import { products } from './products.js';
class Product {
  #stock; // private: only code in the class sees it
  constructor(name, price, stock = 0) {
    this.name = name;
    this.price = price;
    this.#stock = stock;
  }
  get inStock() { return this.#stock > 0; }
  sell(qty = 1) { this.#stock -= qty; }
  static from(p) { return new Product(p.name, p.price, p.stock); }
}
const poles = Product.from(products[8]);
poles.sell(3);
console.log(poles.name, poles.inStock, Object.keys(poles));
```

OUTPUT

```
$ node classes.js
Trek Poles false [ 'name', 'price' ]
```

Trek Poles started with 3 in stock, so after `sell(3)` the getter says `false`, and `Object.keys` doesn't list `#stock`. `instanceof` checks the blueprint: `poles instanceof Product` is `true`.

EXERCISES

Try each one before you check the answer.

1. **WARM-UP** Predict what the `var` and `let` loops in 6.1 print, then run them.
2. **PRACTICE** Write `makeCounter()` returning `{ increment(), current() }`, with a private count.
3. **CHALLENGE** Build a `Cart` class: private `#items`, `add(product, qty = 1)`, a `total` getter and `static fromJSON(json)`. Two Trail Runners and a Sun Hat should total 202.

Answers: page 101

KEY TAKEAWAYS

- Scopes nest: code sees outward, never inward. Use `let` and `const`, never `var`.
- A closure is a function plus the variables it remembers: counters, limiters and caches.
- `this` depends on how a function is called. Pass methods as arrows or bind them.
- Classes bundle data and behavior, with `#private` fields, getters, `static` factories and `extends`.

Go deeper (for Part 3): Web crawler request queues · Stateless vs stateful scraping

07



PART 2 · MODERN JAVASCRIPT

Asynchronous JavaScript

Understand the event loop, then write async code with promises and `async` / `await` that waits well, runs in parallel and fails gracefully.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

7.1 Why JavaScript waits without blocking

7.2 The event loop

7.3 Callbacks, then promises

7.4 The `async` and `await` keywords

7.5 Running many tasks

7.6 Errors, timeouts and cancellation

7.1 Why JavaScript Waits Without Blocking

JavaScript runs your code on one thread: one line at a time, in order. Yet a single Node.js script can wait on hundreds of network requests at once, because it never sits and waits. It starts the slow work, moves on, and comes back when the result is ready.

coffee.js

```
console.log('Order a coffee');
setTimeout(() => console.log('Coffee!'), 1000);
console.log('Read the news meanwhile');
```

OUTPUT

```
$ node coffee.js
Order a coffee
Read the news meanwhile
Coffee!
```

Node didn't stop at line 2: `setTimeout` booked the callback and moved on, then ran it once the second had passed. That's **asynchronous** ("async") code: start a job now, handle its result later. Timers, network calls and file reads all work this way in Node.js.

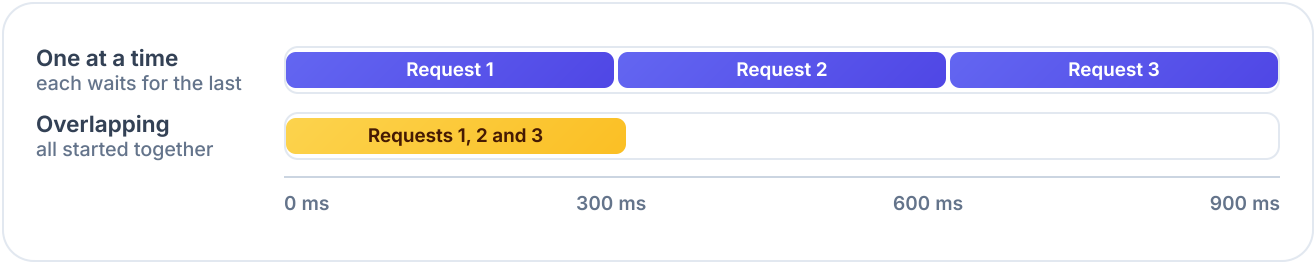


Figure 7.1 Three 300 ms requests take about 900 ms one after another and about 300 ms started together, because the waiting overlaps. A worked example, not a benchmark.

One thread, so never block it

With one thread, a long calculation stops everything: timers, network callbacks, server replies. Here a 100 ms timer has to wait for a loop that keeps the thread busy for a whole second.

busy.js

```
const start = Date.now();
setTimeout(() => {
  console.log(`Timer fired after ${Date.now() - start} ms`);
}, 100);
while (Date.now() - start < 1000) {}   // busy: nothing else can run
console.log('Loop finished');
```

OUTPUT

```
$ node busy.js
Loop finished
Timer fired after 1002 ms
```

Your number will differ by a few milliseconds, but it's never close to 100.

7.2 The Event Loop

So who decides when a callback runs? The **event loop**: a loop inside Node.js that keeps checking for work that's ready and hands it to your code, one piece at a time. It juggles the four parts in Figure 7.2.

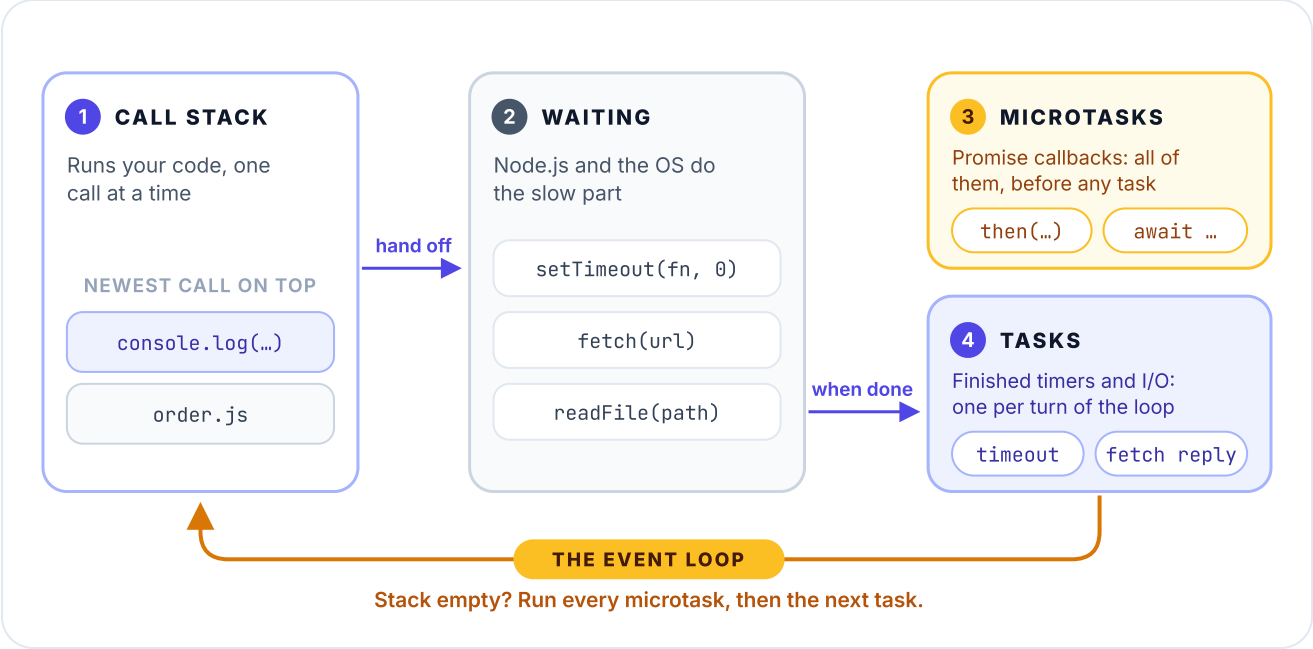


Figure 7.2 Code runs on the call stack (1) while slow work waits outside JavaScript (2). Promise callbacks queue as microtasks (3), and finished timers and I/O as tasks (4).

order.js

```
console.log('1 sync');
setTimeout(() => console.log('4 timeout'), 0);
Promise.resolve().then(() => console.log('3 microtask'));
console.log('2 sync');
```

OUTPUT

```
$ node order.js
1 sync
2 sync
3 microtask
4 timeout
```

The current code always finishes first. Then the promise callback, a microtask, beats the timer even though the timer asked for 0 ms: a 0 ms timer means "soon, after everything else", never "now".

Going further

Node's loop runs in phases, such as timers (`setTimeout`), poll (network and file I/O) and check (`setImmediate`); you'll rarely need them. To queue a microtask yourself, use `queueMicrotask()` , which Node's docs prefer to `process.nextTick()` .

7.3 Callbacks, Then Promises

The first async style was the callback: the slow operation calls your function with the result, error first in Node's classic APIs, as in `readFile(path, (err, text) => { ... })`. When each step needs the one before, callbacks nest into "callback hell". Promises flatten it.

CONCEPT

Promise

An object that stands for a result that isn't ready yet. It starts **pending**, then settles once: **fulfilled** with a value or **rejected** with an error. You say what happens next with `then` and `catch`, or wait for it with `await`.

THINK OF IT AS

The buzzer a burger counter hands you: you get it at once, and it goes off when your order is ready, or when the kitchen runs out.

IN CODE

```
const reply = fetch(url); // a promise
```

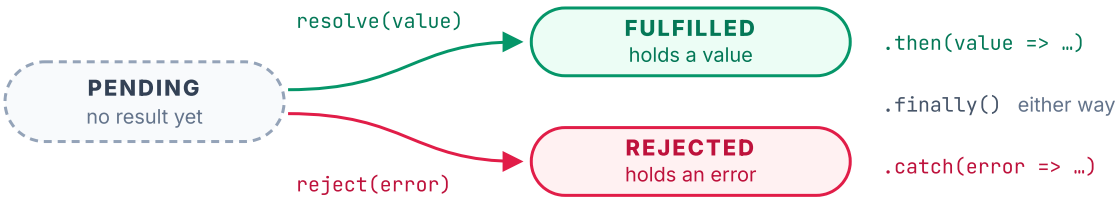


Figure 7.3 A promise settles once and never changes after that. `then` gets the value, `catch` the error, and `finally` runs either way.

stock.js

```
function checkStock(name) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (name === 'Head Lamp') reject(new Error(`${name} is sold out`));
      else resolve(`${name} is in stock`);
    }, 300);
  });
}
checkStock('Sun Hat')
  .then((message) => console.log(message))
  .then(() => checkStock('Head Lamp'))
  .then((message) => console.log(message))   // skipped: it rejected
  .catch((error) => console.log('Problem:', error.message))
  .finally(() => console.log('Done checking'));
```

OUTPUT

```
$ node stock.js
Sun Hat is in stock
Problem: Head Lamp is sold out
Done checking
```

7.4 The `async` and `await` Keywords

Mark a function `async` and it always returns a promise. Inside it, `await` pauses that function until a promise settles and hands you the value, and a rejection is thrown as an error, so plain `try` / `catch` works. Only the function pauses; the event loop keeps everything else running. In an ES module, which every file in this course is, `await` also works at the top level.

timing.js

```
import { setTimeout as sleep } from 'node:timers/promises';

const PRICES = { 'Rain Shell': 99, 'Trek Poles': 79, 'Sun Hat': 24 };
async function getPrice(name) {
  await sleep(300);                    // stands in for a network call
  return PRICES[name];
}
const names = Object.keys(PRICES);

console.time('one at a time');
for (const name of names) await getPrice(name);   // waits for each
console.timeEnd('one at a time');

console.time('all at once');
console.log(await Promise.all(names.map(getPrice)));
console.timeEnd('all at once');
```

OUTPUT

```
$ node timing.js
one at a time: 904.106ms
[ 99, 79, 24 ]
all at once: 302.45ms
```

Your timings will differ slightly. The gap between them won't.

The loop awaits each call before it starts the next, so the waits add up, as in Figure 7.1. `names.map(getPrice)` starts all three at once, and `Promise.all` waits for every result, in input order. `sleep` is the promise version of `setTimeout` from `node:timers/promises`; you'll see it all through the book.

⚠ COMMON MISTAKE **Forgetting `await`**

INSTEAD OF

```
const price = getPrice('Rain Shell');
console.log(price);
// Promise { <pending> }
```

WRITE

```
const price = await getPrice('Rain Shell');
console.log(price);
// 99
```

Without `await` you get the promise itself, still pending, instead of its value.

COMMON MISTAKE An `async` callback in `forEach`

INSTEAD OF

```
names.forEach(async (name) => {
  await save(name);
}); // nothing waits for these
```

WRITE

```
for (const name of names) {
  await save(name);
}
```

`forEach` ignores the promises it gets back. Use `for...of`, or `await Promise.all(names.map(save))`.

Coming from Python

Like asyncio, minus `asyncio.run()`: Node already runs the loop. Calling an async function starts it at once, where a coroutine waits to be awaited. `Promise.all` is your `asyncio.gather`.

7.5 Running Many Tasks

`Promise.all` is one of four combinators: functions that take a list of promises and return one promise. They differ in how they treat failure, so pick by the result you need.

Combinator	Gives you	Rejects when	Use it for
`Promise.all`	Every value, in order	Any one rejects	Steps that must all succeed
`Promise.allSettled`	Every outcome, in order	Never	Batches that report each result
`Promise.race`	The first to settle	The first to settle rejects	Racing work against a deadline
`Promise.any`	The first success	All reject	The fastest of several mirrors

settled.js

```
async function check(site, up) {
  if (!up) throw new Error(`${site} is down`);
  return `${site} is up`;
}
const results = await Promise.allSettled(
  [check('shop', true), check('blog', false), check('api', true)],
);
for (const r of results) console.log(r.status, r.value ?? r.reason.message);
```

OUTPUT

```
$ node settled.js
fulfilled shop is up
rejected blog is down
fulfilled api is up
```

Each outcome is `{ status, value }` or `{ status, reason }`, and one failure never hides the others.

7.6 Errors, Timeouts and Cancellation

A rejected promise throws at its `await`, so `try` / `catch` works as usual. An unhandled rejection is worse: Node prints it and exits with code 1, so `await` or `.catch()` every promise you start.

Give slow work a deadline too: `AbortSignal.timeout(ms)` makes a signal that fires after `ms` milliseconds, for `fetch`, timers and more.

timeout.js

```
import { setTimeout as sleep }
  from 'node:timers/promises';
const signal = AbortSignal.timeout(1000);
try {
  await sleep(3000, undefined, { signal });
  console.log('Report ready');
} catch (error) {
  console.log(error.name, error.cause.name);
}
```

OUTPUT

```
$ node timeout.js
AbortError TimeoutError
```

Timers reject with an `AbortError` (the reason is in `cause`); `fetch` with a `TimeoutError`.

EXERCISES

Try each one before you check the answer.

1 WARM-UP Predict the order of a `console.log`, a `setTimeout(…, 0)`, a `Promise.resolve().then(…)` and a `queueMicrotask(…)`, then run them.

2 PRACTICE Write your own `sleep(ms)` with `new Promise`, then a countdown that prints 3, 2, 1 and `Liftoff!` one second apart.

3 CHALLENGE Run five fake jobs where two throw and print `3 ok, 2 failed` with `allSettled`. Then give each a 1-second timeout and make one take 1.5 seconds.

Answers: page 102

KEY TAKEAWAYS

- Node runs your code on one thread and hands slow work to the system: never block it.
- The event loop finishes the current code, then every microtask, then the next task.
- A promise settles once, and `await` reads it top to bottom, with `try` / `catch`.
- Start independent work together with `Promise.all`, and give slow work a timeout.

Go deeper: Rate limiting in web scraping · Rate limiting · Request throttling · Handling timeouts and errors

08



PART 2 · MODERN JAVASCRIPT

Iterators, Generators, JSON and Regex

Process data one item at a time, move it in and out of JSON safely and find patterns in text with regular expressions.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

8.1 Iterables and generators

8.2 Async generators and pagination

8.3 JSON in and out

8.4 Regular expressions

8.1 Iterables and Generators

`for...of` works on anything **iterable**: arrays, strings, Maps, Sets and more. Each one can hand out an **iterator**, an object whose `next()` method returns `{ value, done }`. `for...of` and the spread operator `...` just call `next()` until `done` is `true`.

A **generator** function, written `function*`, builds iterators for you. Each `yield` hands out one value and pauses the function until the next `next()` call. Values are made only when someone asks, so a generator can even run forever without filling memory.

range.js

```
function* range(start, end) {
  for (let n = start; n < end; n++) yield n;   // hand out n, then pause
}
const counter = range(1, 3);
console.log(counter.next());
console.log(counter.next());
console.log(counter.next());
console.log([...range(1, 6)]);

function* orderIds() {                          // endless, but lazy
  let n = 1;
  while (true) yield `ORD-${String(n++).padStart(4, '0')}`;
}
console.log(orderIds().take(3).toArray());
console.log(orderIds().filter((id) => id.endsWith('7')).take(2).toArray());
```

OUTPUT

```
$ node range.js
{ value: 1, done: false }
{ value: 2, done: false }
{ value: undefined, done: true }
[ 1, 2, 3, 4, 5 ]
[ 'ORD-0001', 'ORD-0002', 'ORD-0003' ]
[ 'ORD-0007', 'ORD-0017' ]
```

`take`, `filter` and `toArray` are ES2025 iterator helpers (with `map` and `drop`). They're lazy, so `take(3)` stops the endless generator after three values; `Iterator.from(x)` adds them to any iterable.

8.2 Async Generators and Pagination

Put `async` in front of `function*` and you can `await` inside the generator. You read an **async generator** with `for await...of`, which waits for each value. It's the right shape for paginated data: fetch a page, hand out its items one by one, and fetch the next page only when they run out.

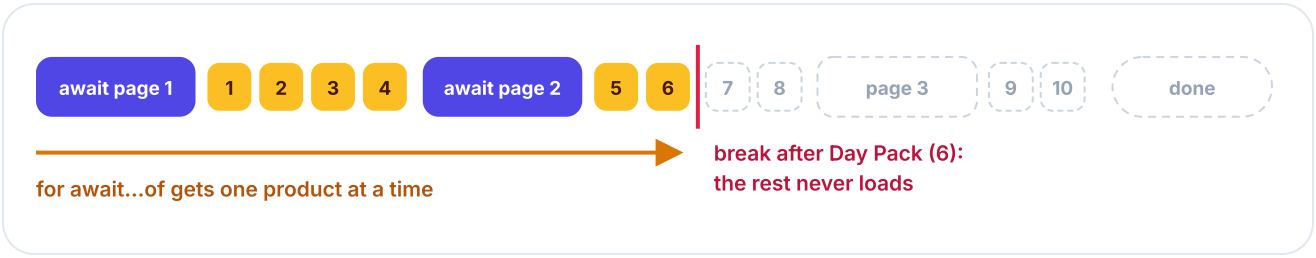


Figure 8.1 How `pages.js` below runs. The generator awaits a page, hands out its products, and only then awaits the next page.

pages.js

```
import { setTimeout as sleep } from 'node:timers/promises';

const NAMES = ['Trail Runner', 'Road Racer', 'Hiking Boot', 'Rain Shell',
  'Wool Socks', 'Day Pack', 'Head Lamp', 'Water Bottle', 'Trek Poles', 'Sun Hat'];

async function* allProducts() {
  for (let page = 1; page <= 3; page++) {
    await sleep(200);                          // stands in for an API call
    console.log(`  (loaded page ${page})`);
    yield* NAMES.slice((page - 1) * 4, page * 4); // hand out the page's items
  }
}

for await (const name of allProducts()) {
  console.log(name);
  if (name === 'Day Pack') break;              // page 3 never loads
}
```

OUTPUT

```
$ node pages.js
  (loaded page 1)
Trail Runner
Road Racer
Hiking Boot
Rain Shell
  (loaded page 2)
Wool Socks
Day Pack
```

The loop sees products, never pages; `yield*` hands out a whole array one item at a time. To collect everything instead, ES2026's `await Array.fromAsync(allProducts())` does it in one line. Chapter 12 uses this shape on real API pages.

8.3 JSON In and Out

JSON is text that looks like JavaScript objects, and it's how APIs and config files exchange data. `JSON.stringify(value)` turns a value into that text (add `null, 2` for readable indenting) and `JSON.parse(text)` turns it back. Only some values survive the round trip:

You stringify	You get back
Text, numbers, booleans, `null` **, arrays, objects**	The same values
`undefined` **and functions**	Dropped from objects; `null` inside arrays
A `Date`	A string, such as `"2026-11-21T07:30:00.000Z"`
A `Map` **, a** `Set` **or a class instance**	`{}` , or a plain object without its methods
`NaN` **and** `Infinity`	`null`
A `BigInt`	A `TypeError` , so convert it to a string first

For the rest, `stringify` takes a **replacer**, such as `(key, value) => value instanceof Set ? [...value] : value`, and `parse` a **reviver** that turns the array back:

reviver.js

```
const text = '{"id":42,"tags":["gift"]}';
const order = JSON.parse(text, (key, value) =>
  key === 'tags' ? new Set(value) : value);
console.log(order);
```

OUTPUT

```
$ node reviver.js
{ id: 42, tags: Set(1) { 'gift' } }
```

Wrap `JSON.parse` in `try` / `catch` when text comes from outside: an HTML error page gives `Unexpected token '<', "<!doctype "... is not valid JSON`. To import a JSON file, add `with { type: 'json' }` to the `import`.

8.4 Regular Expressions

A **regular expression** (regex) is a pattern for text, written between slashes with flags after. Use it to test, find, extract or replace: prices in a description, dates in a log, IDs in a URL.



1. **Named group** `currency` : three capitals.
2. **Whitespace:** a space, tab or newline.
3. **Named group** `amount` opens.
4. **One or more** digits or commas.
5. **A real dot.** Plain `.` is any character.
6. **Exactly two** digits.
7. **Flag:** find every match, not just one.

Figure 8.2 A price pattern, piece by piece. The appendix has a full regex cheat sheet.

prices.js

```
const text = 'Prices: USD 1,299.00 and EUR 89.50';
const price = /(?<currency>[A-Z]{3})\s(?<amount>[\d,]+\.\d{2})/g;
for (const match of text.matchAll(price)) {
  console.log(match.index, match.groups);
}
console.log(text.replace(price, '$<amount> $<currency>'));
console.log(/sold out/i.test('Head Lamp: SOLD OUT'));   // i: ignore case
```

OUTPUT

```
$ node prices.js
8 [Object: null prototype] { currency: 'USD', amount: '1,299.00' }
25 [Object: null prototype] { currency: 'EUR', amount: '89.50' }
Prices: 1,299.00 USD and 89.50 EUR
true
```

`matchAll` gives each match's position and groups (a null-prototype object, see Chapter 4); `replace` reuses them as `$<name>`. `RegExp.escape(text)` (ES2025) makes text safe in a pattern.

Common mistake

Parsing HTML with regex. Tags nest and attributes move, so the pattern breaks on the next redesign. Parse HTML with Cheerio (Chapter 13), then use regex on the text you pull out.

EXERCISES Try each one before you check the answer.

1. **WARM-UP** Add `step = 1` to `range` so `[...range(0, 10, 3)]` is `[0, 3, 6, 9]`.
2. **PRACTICE** Round-trip `{ savedAt: new Date() }` through JSON; restore the `Date`.
3. **CHALLENGE** Turn the prices text into `{ currency, amount }` objects, amounts as numbers.

Answers: page 103

KEY TAKEAWAYS

- Anything iterable works with `for...of`; generators make lazy iterables with `yield`.
- Async generators and `for await...of` turn page after page into one stream of items.
- JSON keeps plain data only: convert dates, Sets and Maps with a replacer and a reviver.
- Named groups make regex readable. Use regex on text, never to parse HTML.

Go deeper: JSON Formatter · Schema Extractor · CSS selectors vs XPath

09



PART 2 · MODERN JAVASCRIPT

Modules, Packages and TypeScript

Split code into modules, use npm packages safely and take a practical first look at TypeScript, which Node.js 26 runs directly.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

9.1 Modules: import and export

9.2 ES modules vs CommonJS

9.3 Packages with `npm` and `package.json`

9.4 Choosing packages safely

9.5 A first look at TypeScript

9.1 Modules: `import` and `export`

As a program grows, one file becomes many. Each file is a **module**: everything in it stays private unless you `export` it, and other files `import` only what they need. Names never clash between files, and each piece is easier to test and reuse.

money.js

```
const fmt = new Intl.NumberFormat('en-US',
  { style: 'currency', currency: 'USD' });

export const TAX_RATE = 0.08;

export function usd(amount) {
  return fmt.format(amount);
}

export default function withTax(amount) {
  return amount * (1 + TAX_RATE);
}
```

main.js

```
import withTax, { usd, TAX_RATE as tax }
  from './money.js';
import * as money from './money.js';

console.log(usd(1234.5));
console.log(`Tax rate: ${tax * 100}%`);
console.log(usd(withTax(89)));
console.log(Object.keys(money));
```

OUTPUT

```
$ node main.js
$1,234.50
Tax rate: 8%
$96.12
[ 'TAX_RATE', 'default', 'usd' ]
```

Figure 9.1 `money.js` has two named exports and one default export. `main.js` imports them by name, renames one with `as` and gathers them all with `* as`.

Relative imports need the file extension (`'./money.js'`), braces must name real exports, and a default export takes any name you like. In a module, `import.meta.dirname` is the file's folder, `await import('./big.js')` loads a module only when you need it, and `await` works at the top level.

9.2 ES Modules vs CommonJS

Node had modules before JavaScript did. Its original system, **CommonJS**, uses `require()` and `module.exports`, and you'll meet it in older code and tutorials. **ES modules** (ESM) are the standard, and this course uses them everywhere.

	ES modules	CommonJS
Import	`import { usd } from './money.js'`	`const { usd } = require('./money')`
Export	`export function usd() {}`	`module.exports = { usd }`
Switched on by	`"type": "module"` or `.mjs`	`"type": "commonjs"` or `.cjs`
Top-level `await`	Yes	No
This file's folder	`import.meta.dirname`	`__dirname`

Node picks the system per file: `.mjs` is always ESM, `.cjs` always CommonJS, and `.js` follows the `"type"` field in the nearest `package.json`. With no `"type"`, Node spots `import` syntax, runs the file as ESM and warns. Node 26 can even `require()` an ES module that has no top-level `await`.

COMMON MISTAKE **Keeping the `package.json` that `npm init -y` wrote**

INSTEAD OF

```
$ npm init -y   # "type": "commonjs"
$ node hello.js
SyntaxError: Cannot use import ...
```

WRITE

```
$ npm init -y
$ npm pkg set type=module
$ node hello.js   # imports work
```

npm 11 writes `"type": "commonjs"`, so `import` fails with "Cannot use import statement outside a module". One command switches the project to ES modules.

9.3 Packages with `npm` and `package.json`

npm is two things: the public registry of JavaScript packages and the command that installs them. Your project's `package.json` records its name, its scripts and the packages it depends on.

package.json

```
{
  "name": "js-course",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "start": "node lab.js",
    "check": "tsc"
  },
  "dependencies": { "cheerio": "^1.2.0" },
  "devDependencies": { "typescript": "^7.0.2" }
}
```

Command	What it does
`npm install cheerio`	Adds a dependency
`npm install -D typescript`	Adds a dev tool
`npm uninstall cheerio`	Removes one
`npm ci`	Installs the lockfile exactly
`node --run check`	Runs a script
`npx tsc`	Runs a package's command

Figure 9.2 A course `package.json` and the commands that change it. `dependencies` ship with your code; `devDependencies` are tools you only need while developing.

`npm install` puts packages in `node_modules` (never commit it: list it in `.gitignore`) and records exact versions in `package-lock.json` (do commit it, so `npm ci` rebuilds the same tree anywhere). `node --run` starts scripts faster than `npm run` and skips `pre` and `post` scripts.

Coming from Python

npm is pip with a built-in virtual environment: packages install into this project's `node_modules`, so projects never clash. `package.json` plays the part of `pyproject.toml`, and `package-lock.json` pins exact versions like a lock file.

In `package.json`, the symbol in front of a version says which updates `npm install` may pick up later:

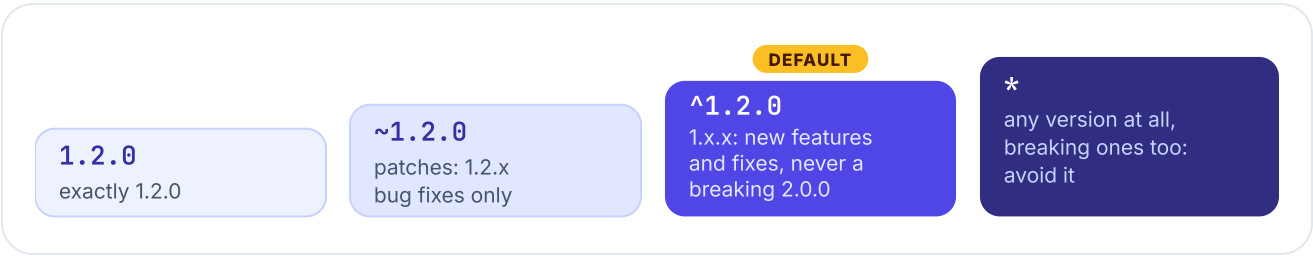


Figure 9.3 Version ranges, from strict to loose. A version reads major.minor.patch, and `npm install` saves `^` ranges unless you say otherwise.

9.4 Choosing Packages Safely

Every package you install runs with your permissions, and so do the packages it depends on. Attackers know it: they publish **typosquats**, names one letter off popular packages, and hide code in **install scripts** that run the moment you install. A minute of checking is cheap insurance.

- [ ] **Is it built in?** Node 26 has `fetch`, tests, SQLite and argument parsing.
- [ ] **Is it maintained?** Look for recent releases and answered issues.
- [ ] **Does the license fit?** MIT and Apache-2.0 suit most projects.
- [ ] **Is the name exactly right?** Copy it from the docs, not from memory.
- [ ] **Is it widely used?** Check weekly downloads on its npm page.
- [ ] **Unsure?** Install with `--ignore-scripts`, then run `npm audit`.

9.5 A First Look at TypeScript

TypeScript is JavaScript plus type annotations: you write `price: number`, and a checker warns you when a string sneaks in. Node.js 26 runs `.ts` files directly by deleting the annotations first, called **type stripping**. It never checks them; that's the job of the TypeScript compiler, `tsc`.

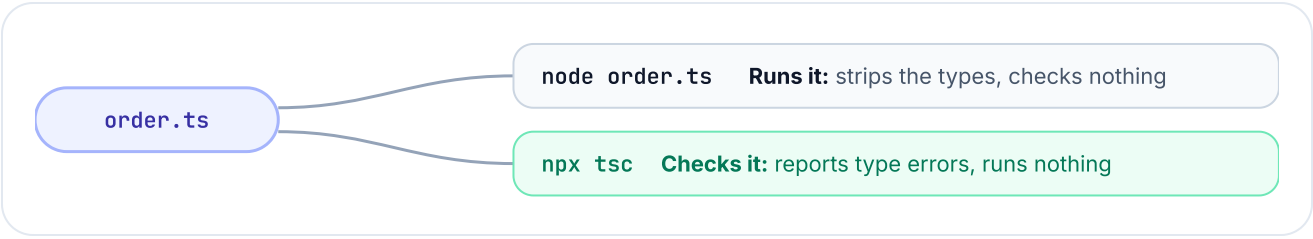


Figure 9.4 One file, two tools. Run with `node`, check with `tsc`.

> ⚠ **Common mistake**
>
> Using syntax that needs rewriting: Node rejects `enum` and namespaces. Use an `as const` object instead, `import type` for types, and `.ts` in import paths.

order.ts

```ts
type Product = { name: string; price: number; tags?: string[] };
function total(items: Product[]): number {
  return items.reduce((sum, p) => sum + p.price, 0);
}
const cart: Product[] = [
  { name: 'Trail Runner', price: 89 },
  { name: 'Sun Hat', price: '24' },           // a string by mistake
];
console.log(total(cart));
```

OUTPUT

```
$ node order.ts
8924
$ npx tsc --noEmit order.ts
order.ts:7:22 - error TS2322: Type 'string' is not assignable to type 'number'.
...the line, then where Product declares price...
Found 1 error in order.ts:7
```

Node ran the bug and printed `8924`, gluing the numbers together as text; `tsc`, installed once with `npm install -D typescript`, caught it without running anything. `tags?` marks an optional property, and outside data such as parsed JSON should be typed `unknown` until you've checked it. For a whole project, add the `tsconfig.json` in the appendix and run plain `npx tsc`.

EXERCISES

Try each one before you check the answer.

1. **WARM-UP** — Move `usd()` into `money.js`, import it, and fix the CommonJS error if it appears.
2. **PRACTICE** — Make a project with `cheerio`, dev tool `typescript`, and `start` and `check` scripts.
3. **CHALLENGE** — Rewrite Chapter 6's `Cart` as `cart.ts`, then make `tsc` flag a string price.

Answers: page 104

KEY TAKEAWAYS

- Each file is a module: export what others need and import it by name, with the `.js` extension.
- Run `npm pkg set type=module` once; `require()` is the older CommonJS style.
- Commit `package-lock.json`, never `node_modules`, and vet packages first.
- Node 26 runs TypeScript by stripping the types; `tsc` is what checks them.

Go deeper: Node.js web scraping with proxies · JSON Formatter · The Scraping Blueprint

10



PART 3 · BEYOND THE BROWSER

Node.js Core and Command-Line Tools

Use Node's built-in modules to work with files, processes, events and streams, then turn a script into a real command-line tool.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

10.1 Files and paths

10.2 Saving data: JSON, NDJSON and CSV

10.3 Arguments, environment and exit codes

10.4 Events and streams

10.5 Build a command-line tool

10.1 Files and Paths

Node's built-in modules give your code the computer: files, folders, processes and the network. They ship with Node, so there's nothing to install; import them with the `node:` prefix. These eight cover most scripts:

node:fs Read and write files	**node:path** Paths for every system	**process** Arguments, env, exit	**node:events** Emit and listen
node:stream Data in chunks	**node:http** Servers (Chapter 11)	**node:util** parseArgs, styleText	**node:readline** Input line by line

For files, use the promise versions in `node:fs/promises`, and build paths with `join` from `node:path`, which uses the right separator on every system:

notes.js

```
import { mkdir, writeFile, appendFile, readFile } from 'node:fs/promises';
import { join } from 'node:path';

const dir = join(import.meta.dirname, 'data');   // next to this script
await mkdir(dir, { recursive: true });           // no error if it exists
const file = join(dir, 'notes.txt');

await writeFile(file, 'Restock Head Lamp\n');    // create or replace
await appendFile(file, 'Email the supplier\n');  // add to the end
console.log(await readFile(file, 'utf8'));
```

OUTPUT

```
$ node notes.js
Restock Head Lamp
Email the supplier
```

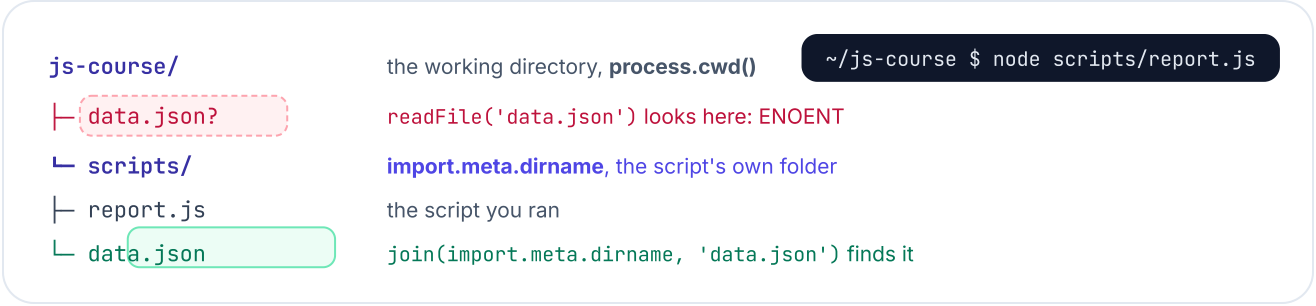


Figure 10.1 Same file name, two folders. Relative paths start from where you run `node`; `import.meta.dirname` starts from the script.

> ⚠ **Common mistake**
>
> Trusting a relative path. `'data.json'` is resolved from the folder you ran `node` in, the current working directory, not from the script's folder, so anchor paths with `import.meta.dirname`. A missing file throws an error whose `code` is `'ENOENT'`: check for exactly that, and rethrow anything else.

10.2 Saving Data: JSON, NDJSON and CSV

Plain files are the simplest home for collected data. Pick the format by who reads it next:

Format	Looks like	Best for
JSON	`[{ "id": 1, "name": "Trail Runner" }]`	One document: settings, API replies
NDJSON	One JSON object per line	Logs and long runs: safe to append
CSV	`id,name,price` then a row per record	Spreadsheets and sharing
SQLite	A database in one file	Queries and updates (Chapter 13)

csv.js

```
import { writeFile } from 'node:fs/promises';

export function toCsv(rows) {
  const cell = (value) => {
    const text = String(value ?? '');
    return /[",\r\n]/.test(text) ? `"${text.replaceAll('"', '""')}"` : text;
  };
  const headers = Object.keys(rows[0] ?? {});
  const lines = [headers, ...rows.map((row) => headers.map((h) => row[h]))];
  return lines.map((line) => line.map(cell).join(',')).join('\n') + '\n';
}

const rows = [
  { id: 1, name: 'Trail Runner', note: 'light, fast' },
  { id: 4, name: 'Rain "Pro" Shell', note: null },
];
await writeFile('products.csv', toCsv(rows));
console.log(toCsv(rows));
```

OUTPUT

```
$ node csv.js
id,name,note
1,Trail Runner,"light, fast"
4,"Rain ""Pro"" Shell",
```

`toCsv` quotes a cell only if it holds a comma, quote or line break, and doubles inner quotes (RFC 4180); Chapters 13 and 16 reuse it. For NDJSON, append `JSON.stringify(row) + '\n'` per record.

10.3 The Process: Arguments, Environment and Exit Codes

Every script runs as a **process**, and the global `process` object describes it. `process.argv` holds the command line (your arguments start at index 2, after the paths to `node` and the script), and `process.env` holds **environment variables**, the usual home for settings and secrets.

greet.js

```
const [name = 'friend'] = process.argv.slice(2);
const greeting = process.env.GREETING ?? 'Hello';
console.log(`${greeting}, ${name}!`);
```

OUTPUT

```
$ node greet.js Ada
Hello, Ada!
$ node --env-file=.env greet.js Ada
Welcome back, Ada!
```

The second run loads `.env`, holding `GREETING=Welcome back`; keep secrets there and list `.env` in `.gitignore`. Send errors to `console.error` and set `process.exitCode = 1`, which, unlike `process.exit()`, lets output finish. `Ctrl`+`C` sends `SIGINT`: catch it with `process.on('SIGINT', save)`.

You want	Use	Good to know
Your arguments	`process.argv.slice(2)`	Always strings
A setting or a secret	`process.env.GREETING`	`undefined` when it isn't set
To fail, but finish	`process.exitCode = 1`	The script ends normally
To stop at once	`process.exit(1)`	Output still on its way can be lost
To clean up on `Ctrl`+`C`	`process.on('SIGINT', save)`	Then exit when you're done

10.4 Events and Streams

Much of Node runs on events: servers emit `request`, streams `data`, the process `SIGINT`. With `EventEmitter`, `on` listens every time, `once` once, and `emit` fires:

events.js

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

const shop = new EventEmitter();
shop.on('sale', (p) => console.log('Sold:', p));
shop.once('sale', () => console.log('Ka-ching!'));
shop.emit('sale', 'Sun Hat');
shop.emit('sale', 'Wool Socks');
```

OUTPUT

```
$ node events.js
Sold: Sun Hat
Ka-ching!
Sold: Wool Socks
```

A **stream** delivers data in chunks. `readFile` on a 4 GB log loads all 4 GB into memory; a file stream reads 64 KiB at a time, however big the file. `readline` turns it into lines. Our test log has 200,000:

count-404s.js

```
import { createReadStream } from 'node:fs';
import { createInterface } from 'node:readline';

const lines = createInterface({ input: createReadStream('access.log') });
let total = 0;
let notFound = 0;
for await (const line of lines) {         // one line at a time
  total += 1;
  if (line.startsWith('404 ')) notFound += 1;
}
console.log(`${notFound} of ${total} requests were 404s`);
```

OUTPUT

```
$ node count-404s.js
4000 of 200000 requests were 404s
```



Figure 10.2 `pipeline` (`node:stream/promises`) chains streams, paces them and cleans up after errors.

10.5 Build a Command-Line Tool

Providers often hand out proxy lists as `host:port:user:pass`, but code wants `http://user:pass@host:port`. **proxyfmt** converts them from a file or piped input; `--protocol socks5` switches the scheme.



Figure 10.3 The parts of a command line, and where `parseArgs` puts each one.

proxyfmt.js

```
#!/usr/bin/env node
import { parseArgs, styleText } from 'node:util';
import { readFile } from 'node:fs/promises';
import { text } from 'node:stream/consumers';

const { values, positionals } = parseArgs({
  allowPositionals: true,
  options: {
    protocol: { type: 'string', short: 'p', default: 'http' },
    help: { type: 'boolean', short: 'h' },
  },
});
if (values.help) {
  console.log('Usage: proxyfmt [file] [--protocol http|socks5]');
} else {
  const [file] = positionals;                          // no file? read stdin
  const input = file ? await readFile(file, 'utf8') : await text(process.stdin);
  for (const line of input.split('\n').filter((l) => l.trim())) {
    const [host, port, user, ...rest] = line.trim().split(':');
    const pass = rest.join(':');                       // passwords may hold ':'
    if (!host || !port || !user || !pass) {
      console.error(styleText('red', `Skipped: ${line}`));
      process.exitCode = 1;
      continue;
    }
    const auth = `${encodeURIComponent(user)}:${encodeURIComponent(pass)}`;
    console.log(`${values.protocol}://${auth}@${host}:${port}`);
  }
}
```

TERMINAL

```
$ cat list.txt
gate.example.net:7000:student:p@ss
gate.example.net:7001:student:s3cret:with:colons
198.51.100.7:8080
$ node proxyfmt.js list.txt --protocol socks5
socks5://student:p%40ss@gate.example.net:7000
socks5://student:s3cret%3Awith%3Acolons@gate.example.net:7001
Skipped: 198.51.100.7:8080
$ echo 'gate.example.net:7000:student:p@ss' | node proxyfmt.js
http://student:p%40ss@gate.example.net:7000
```

`parseArgs` splits the command line into `values`, the options (`--protocol socks5`, or `-p socks5`) with defaults filled in, and `positionals`, bare arguments such as `list.txt`. An unknown flag throws, so typos never pass silently. `encodeURIComponent` turns the `@` in `p@ss` into `%40`, which would otherwise end the password early, and `styleText` colors the warning red.

Install it as a command

The first line, `#!/usr/bin/env node` (a "shebang"), tells macOS and Linux to run the file with Node. Add `"bin": { "proxyfmt": "./proxyfmt.js" }` to `package.json`, run `npm link`, and `proxyfmt list.txt` works in any folder (npm makes a launcher on Windows). Prefer a browser? The Proxy Format tool does the same job.

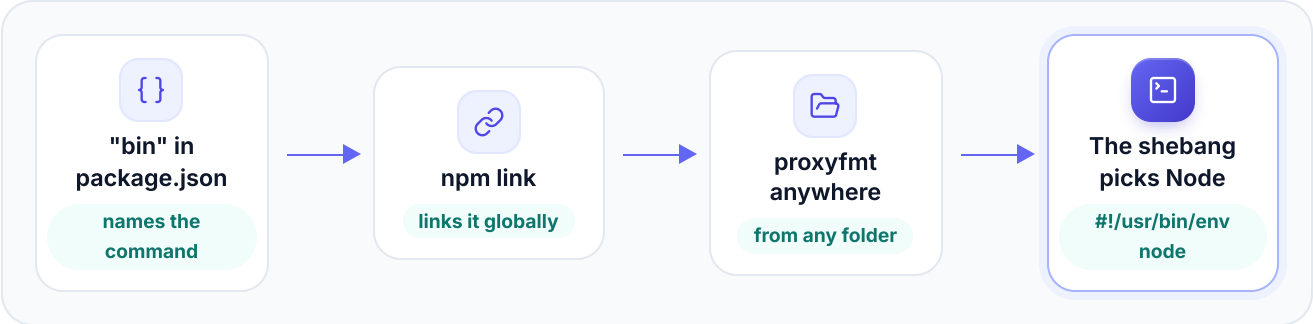


Figure 10.4 How a script becomes a command you can run from any folder.

EXERCISES Try each one before you check the answer.

1	WARM-UP	Print `process.cwd()`, `import.meta.dirname` and `process.argv.slice(2)` from two folders.
2	PRACTICE	Read a `products.json` array and write `products.csv` with `toCsv`.
3	CHALLENGE	Accept `user:pass@host:port` too, reject ports outside 1 to 65535, exit with code 1.

Answers: page 105

KEY TAKEAWAYS

- Import core modules with `node:`, and anchor paths with `join(import.meta.dirname, ...)`.
- JSON for documents, NDJSON for logs you append to, quoted CSV for spreadsheets.
- Settings come from `process.env`; errors go to stderr; failure sets `process.exitCode`.
- Streams handle big data in chunks, and `parseArgs` plus `"bin"` make a script a command.

Go deeper: Proxy Format · cURL Builder · cURL with a proxy · Proxies on a VPS

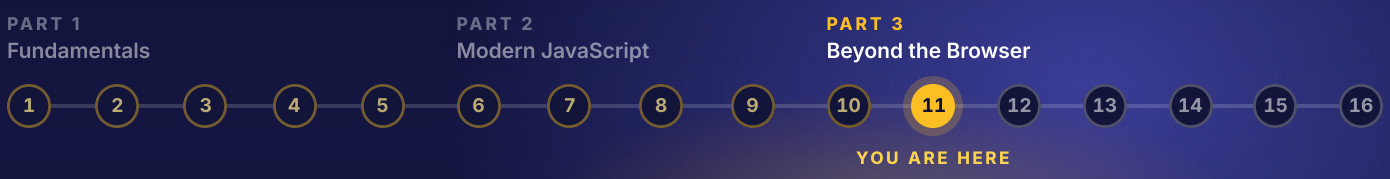
11



PART 3 · BEYOND THE BROWSER

Build a Small API Server

Learn HTTP from the server's side by building the Lab Shop: the practice API you'll call, scrape and automate for the rest of the book.



IN THIS CHAPTER

11.1 How HTTP works
11.2 Your first server
11.3 The Lab Shop, part 1: data and helpers
11.4 The Lab Shop, part 2: the JSON API
11.5 Try the Lab Shop

11.1 How HTTP Works

Every time a browser opens a page or a script calls an API, two programs hold a short conversation. The **client** sends a **request**, the **server** sends back a **response**, and the language they speak is **HTTP**. Chapter 12 makes you the client; this chapter puts you behind the counter.

REQUEST what curl sent

```
GET /api/products?page=2 HTTP/1.1
Host: localhost:3000
User-Agent: curl/8.7.1
Accept: */*
```

RESPONSE what the Lab Shop sent back

```
HTTP/1.1 200 OK
content-type: application/json
link: </api/products?page=3>; rel="next"
Date: Sat, 21 Nov 2026 10:44:09 GMT

{"page":2,"pages":3,"items":[...]}
```

1. **Method.** GET reads, POST sends.
2. **Path and query.** Which data, which page.
3. **Request headers.** Who's asking, and how.
4. **Status.** A three-digit code and a short reason.
5. **Response headers.** Body type, next-page link, date.
6. **Body.** The data itself, here JSON.

Figure 11.1 A real exchange with the Lab Shop, as `curl -v` shows it. An empty line ends the headers; this GET request has no body. Three more response headers are trimmed, and your date and curl version will differ.

The **method** says what the client wants: `GET` reads, `POST` creates or sends, `PUT` and `PATCH` replace or change, `DELETE` removes. Headers are `name: value` lines about the message, and the **status code** says how it went. The first digit gives the family:

2xx — Success
- `200` OK
- `201` Created
- `204` No Content

3xx — Look elsewhere
- `301` Moved Permanently
- `302` Found
- `304` Not Modified

4xx — The request's fault
- `400` Bad Request
- `401` Unauthorized
- `403` Forbidden
- `404` Not Found
- `429` Too Many Requests

5xx — The server's fault
- `500` Internal Server Error
- `502` Bad Gateway
- `503` Service Unavailable
- `504` Gateway Timeout

Jargon buster: port and localhost

localhost is the name your computer uses for itself, so `http://localhost:3000` never leaves your machine. A port is a numbered door, so many servers can share one computer: the Lab Shop takes door 3000. Websites use 443 for HTTPS and 80 for HTTP, so URLs rarely show one. Underneath, HTTP rides on TCP/IP.

11.2 Your First Server

Node's `node:http` module is all you need. `createServer` takes a function that runs once per request and hands it two objects: `req`, the request that arrived, and `res`, the response you build. Start `server.js`, then ask it for a page from a second terminal with **curl**, the command-line HTTP client that comes with macOS, Windows 10 and later, and most Linux systems.

server.js

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

const server = createServer((req, res) => {
  console.log(req.method, req.url);               // runs once per request
  res.writeHead(200, { 'content-type': 'text/plain' });
  res.end('Hello from your own server!\n');
});

server.listen(3000, () => {
  console.log('Listening on http://localhost:3000');
});
```

TERMINAL 1

```
$ node server.js
Listening on http://localhost:3000
GET /hello
(still running)
```

TERMINAL 2

```
$ curl -i localhost:3000/hello
HTTP/1.1 200 OK
content-type: text/plain
Date: Sat, 21 Nov 2026 10:43:56 GMT
... three connection headers ...

Hello from your own server!
```

The server logs `GET /hello` when curl's request arrives. Your date will differ.

`writeHead` sets the status and headers, and `end` sends the body and finishes the response: forget `end` and the client waits until it gives up. Node added `Date` and the connection headers itself, and `-i` told curl to print them. In Windows PowerShell 5.1, type `curl.exe`, because plain `curl` there means `Invoke-WebRequest`. A browser pointed at `localhost:3000/hello` shows the same text.

Tip

A server runs until you press `Ctrl`+`C`. While you're editing one, start it with `node --watch server.js` instead: Node restarts it every time you save.

Common mistake

Starting a second server on a port that's taken. Node stops with `Error: listen EADDRINUSE: address already in use :::3000`, usually because the first copy is still running in another terminal. Stop it, or pick another port: the Lab Shop reads one from the `PORT` variable, as in `PORT=3001 node lab.js` (in PowerShell, `$env:PORT = "3001"; node lab.js`).

11.3 The Lab Shop, Part 1: Data and Helpers

Now build `lab.js`, the shop the rest of the book calls, scrapes and automates, in parts. Window titles give line numbers in the finished file; Table 11.1 maps each idea.

lab.js (lines 1–29)

```
// lab.js: the course's practice server. Run it with: node lab.js
// Chapter 11 builds everything except the two marked website blocks, which
// Chapters 13 and 14 paste in above the "Everything else" section.
import { createServer } from 'node:http';
import { setTimeout as sleep } from 'node:timers/promises';

const PRODUCTS = [                          // [name, price, stock] -> { id, name, ... }
  ['Trail Runner', 89, 12], ['Road Racer', 120.5, 0], ['Hiking Boot', 149.99, 4],
  ['Rain Shell', 99, 7], ['Wool Socks', 14.5, 40], ['Day Pack', 64, 9],
  ['Head Lamp', 29.95, 0], ['Water Bottle', 19, 25], ['Trek Poles', 79, 3],
  ['Sun Hat', 24, 15],
].map(([name, price, stock], i) => ({ id: i + 1, name, price, stock }));
const PER_PAGE = 4;
const PAGES = Math.ceil(PRODUCTS.length / PER_PAGE);          // 3 pages: 4, 4 and 2
const pageOf = (n) => PRODUCTS.slice((n - 1) * PER_PAGE, n * PER_PAGE);
const byId = (id) => PRODUCTS.find((p) => p.id === Number(id));
const usd = (n) => `$${n.toFixed(2)}`;
const html = (title, body) =>
  `<!doctype html><html><head><title>${title}</title></head><body>${body}</body></html>`;

const hits = new Map();                     // address -> { second, count }
function tooMany(address) {                 // more than 5 API calls in the same second?
  const second = Math.floor(Date.now() / 1000);
  const entry = hits.get(address);
  if (entry?.second === second) entry.count += 1;
  else hits.set(address, { second, count: 1 });
  return hits.get(address).count > 5;
}
let flakyCalls = 0;
```

Lines	What they do, and how	Met in
7–12	Map each `[name, price, stock]` list to an object, destructuring it	Chapter 4
13–16	A page of four with `slice`; a product by id (URLs are text: `Number`)	Chapter 4
17–19	Arrow functions and template literals: a price, an HTML page	Chapters 2–3
21–28	A closure over a `Map` counts each visitor's calls per second	Chapter 6
29	Counts calls to `/api/flaky`, which fails on purpose (11.4)	Chapter 2

Table 11.1 Nothing in lines 1–29 is new.

11.4 The Lab Shop, Part 2: The JSON API

Line 31 starts the request handler, which tries each route in turn until one responds.

lab.js (lines 31–63)

```
const server = createServer(async (req, res) => {
  const url = new URL(req.url, 'http://localhost');
  const path = url.pathname;
  const page = Number(url.searchParams.get('page') ?? 1);
  const pageOk = Number.isInteger(page) && page >= 1 && page <= PAGES;
  const send = (status, body, headers = {}) => {
    const isHtml = typeof body === 'string';
    res.writeHead(status, { 'content-type': isHtml ? 'text/html' : 'application/json',
      ...headers });
    res.end(isHtml ? body : JSON.stringify(body));
  };
  res.on('finish', () => console.log(res.statusCode, req.method, req.url));

  // ── The JSON API (called in Chapter 12) ──
  if (path.startsWith('/api/') && tooMany(req.socket.remoteAddress)) {
    return send(429, { error: 'Too many requests' }, { 'retry-after': '1' });
  }
  if (path === '/api/products') {
    if (!pageOk) return send(404, { error: 'No such page' });
    const next = `</api/products?page=${page + 1}>; rel="next"`;
    return send(200, { page, pages: PAGES, items: pageOf(page) },
      page < PAGES ? { link: next } : {});
  }
  const apiId = path.match(/^\/api\/products\/(\d+)$/)?.[1];
  if (apiId) {
    return byId(apiId) ? send(200, byId(apiId)) : send(404, { error: 'Not found' });
  }
  if (path === '/api/me') {
    if (req.headers.authorization !== 'Bearer lab-token') {
      return send(401, { error: 'Missing or wrong token' });
    }
    return send(200, { user: 'student', plan: 'free' });
  }
```

- **Read the URL.** `req.url` is just the path and query, so `new URL` parses it against a stand-in base; `page` becomes a number, 1 by default.
- **One way out.** `send` writes the status, headers and body. Every route says `return send(...)`: drop the `return` and Node crashes with `ERR_HTTP_HEADERS_SENT`.
- **Log, then limit.** `res.on('finish')` (Chapter 10) logs each response; the 429 check runs before every API route.

The rest mirrors your request and misbehaves on purpose for Chapter 12. `req` is a stream (Chapter 10), so `for await` collects the body. Leave the website comment: Chapter 13 replaces it.

lab.js (lines 64–78 and 115–121)

```
  if (path === '/api/echo') {              // shows exactly what your request sent
    let body = '';
    for await (const chunk of req) body += chunk;
    const { 'content-type': type, 'user-agent': agent, authorization } = req.headers;
    return send(200, { method: req.method, type, agent, authorization, body });
  }
  if (path === '/api/slow') {              // takes 3 seconds to answer
    await sleep(3000);
    return send(200, { ok: true });
  }
  if (path === '/api/flaky') {             // fails twice, succeeds once, repeats
    flakyCalls += 1;
    if (flakyCalls % 3 === 0) return send(200, { ok: true, call: flakyCalls });
    return send(503, { error: 'Try again' }, { 'retry-after': '1' });
  }

  // Chapters 13 and 14 add website routes here.

  // ── Everything else (Chapter 11) ──
  if (path === '/') return send(200, html('Lab Shop', '<h1>Lab Shop is running</h1>'));
  return send(404, { error: `No route for ${path}` });
});

const PORT = Number(process.env.PORT ?? 3000);
server.listen(PORT, () => console.log(`Lab running at http://localhost:${PORT}`));
```

11.5 Try the Lab Shop

Run `node lab.js` and leave it running: every LAB example needs it. Open `localhost:3000/api/products` in a browser, or curl it to see the `Link` header. In zsh, the macOS shell, quote any URL containing `?`, or zsh says `no matches found`.

TERMINAL 1

```
$ node lab.js
Lab running at http://localhost:3000
200 GET /api/products
404 GET /api/products/99
```

TERMINAL 2

```
$ curl -i localhost:3000/api/products
HTTP/1.1 200 OK
content-type: application/json
link: </api/products?page=2>; rel="next"
... four more headers ...

{"page":1,"pages":3,"items":[...]}
$ curl localhost:3000/api/products/99
{"error":"Not found"}
```

Route	What it does	Used in
`GET /api/products?page=N`	4 a page, 3 pages, a `Link` to the next; bad page: 404	Ch 12, 15, 16
`GET /api/products/:id`	One product, or 404 `{ "error": "Not found" }`	Ch 12, 15
`GET /api/me`	200 with `Bearer lab-token`, otherwise 401	Ch 12
`ANY /api/echo`	Echoes your method, body and three key headers	Ch 12, 16
`GET /api/slow`	Answers `{ "ok": true }` after 3 seconds	Ch 12
`GET /api/flaky`	503 twice, then 200, repeating	Ch 12
Any `/api/` route	The 6th call in one second gets 429, `Retry-After: 1`	Ch 12, 15
`GET /`, anything else	"Lab Shop is running", or 404 `{ "error": ... }`	Ch 11
Coming in Ch 13–14	`/robots.txt`, `/products` pages, product pages, `/app`	Ch 13, 14, 16

Table 11.2 The Lab Shop's routes so far; the appendix lists them all.

Going further

Express, Fastify and Hono do this routing for you, with patterns like `/api/products/:id`. Now you know what they do underneath.

EXERCISES

Try each one before you check the answer.

1. **WARM-UP** Add `/api/health`, returning `{ ok: true, uptime: process.uptime() }`, and curl it.
2. **PRACTICE** Add a case-insensitive `?q=` search before paging: `?q=hat` finds the Sun Hat.
3. **CHALLENGE** Send `x-ratelimit-limit` and `x-ratelimit-remaining` on every API response.

Answers: page 106

KEY TAKEAWAYS

- HTTP is a request (method, path, headers, body) and a response (status, headers, body).
- `createServer` calls your function per request; answer each one exactly once.
- Status codes speak for you: 2xx success, 4xx the client's fault, 5xx the server's.
- The Lab Shop runs at `localhost:3000`: start it with `node lab.js` before any Lab example.

Go deeper: APIs explained · Rate limiting · Forward vs reverse proxies · robots.txt · cURL Builder

12



PART 3 · BEYOND THE BROWSER

Calling REST and GraphQL APIs

Call real APIs with `fetch`: query strings, JSON bodies, auth, pagination, rate limits, retries and timeouts, plus a short look at GraphQL.

PART 1 Fundamentals
PART 2 Modern JavaScript
PART 3 Beyond the Browser
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
YOU ARE HERE

IN THIS CHAPTER

12.1 Five minutes with `fetch`
12.2 Sending data and authenticating
12.3 Pagination
12.4 Rate limits, retries and timeouts
12.5 A short look at GraphQL

12.1 Five Minutes with `fetch`

In Chapter 11 you stood behind the counter. Now every program is a **client** that asks a server for data, with the `fetch` built into Node.js. Start the Lab Shop in one terminal and run this in another.

first-fetch.js — LAB

```
const base = 'http://localhost:3000/api/products';

const res = await fetch(`${base}/3`);              // send the request
console.log(res.status, res.ok, res.headers.get('content-type'));
console.log(await res.json());                     // read the body

const missing = await fetch(`${base}/99`);         // no error thrown!
console.log(missing.status, missing.ok, await missing.json());
```

OUTPUT

```
$ node first-fetch.js
200 true application/json
{ id: 3, name: 'Hiking Boot', price: 149.99, stock: 4 }
404 false { error: 'Not found' }
```

`fetch` returns a promise of a **Response**: a status, headers and a body you read once (`res.json()` for JSON, `res.text()` for HTML). Look at the last line: a 404 is still a response, so `fetch` doesn't throw. It rejects only when no response arrives at all, because the server is down or the network failed.

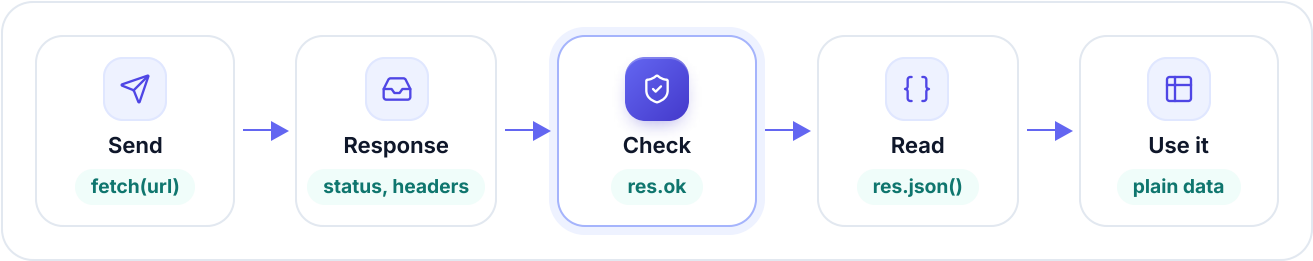


Figure 12.1 Five steps for every API call. Beginners skip the middle one: `res.ok` is true only for statuses 200 to 299.

⚠ COMMON MISTAKE **Reading the body without checking `res.ok`**

INSTEAD OF

```
const res = await fetch(url);
const data = await res.json();
// a 404 answer, used as if
// it were your data
```

WRITE (IN HELPERS.JS)

```
export async function getJson(url, opts) {
  const res = await fetch(url, opts);
  if (!res.ok) {
    throw new HttpError(res.status, url);
  }
  return res.json();
}
```

Throw Chapter 6's `HttpError` on a bad status. Keep `getJson()` in `helpers.js`, your reusable pieces.

12.2 Sending Data and Authenticating

Reading is a `GET`. To send data, give `fetch` a second argument: the `method`, a `content-type` header saying what the body is, and the body as text (`JSON.stringify` turns an object into JSON). The lab's `/api/echo` answers with exactly what it received.

post-echo.js LAB

```
const lab = 'http://localhost:3000';
const res = await fetch(`${lab}/api/echo`, {
  method: 'POST',
  headers: {
    'content-type': 'application/json',
    'user-agent': 'js-course-bot/1.0',
  },
  body: JSON.stringify({ item: 3, qty: 2 }),
});
console.log(await res.json());
```

OUTPUT

```
$ node post-echo.js
{
  method: 'POST',
  type: 'application/json',
  agent: 'js-course-bot/1.0',
  body: '{"item":3,"qty":2}'
}
```

The `user-agent` header names your script. Leave it out and Node's `fetch` just says `node`. A descriptive name tells site owners who's calling; Chapter 13 makes that a rule.

Prove who you are

Style	What you send	Good to know
API key	`x-api-key: KEY`	Simple; the header name varies by API
Bearer token	`authorization: Bearer TOKEN`	The most common; often a JWT (JWT Decoder)
Basic auth	`authorization: Basic BASE64`	`user:pass` encoded, not encrypted (Base64)
Key in the URL	`?api_key=KEY`	Avoid: URLs end up in logs and history

me.js LAB

```
const url = 'http://localhost:3000/api/me';
const token = process.env.LAB_TOKEN;
let res = await fetch(url);
console.log(res.status, await res.json());
res = await fetch(url, {
  headers: { authorization: `Bearer ${token}` },
});
console.log(res.status, await res.json());
```

OUTPUT

```
$ node --env-file=.env me.js
401 { error: 'Missing or wrong token' }
200 { user: 'student', plan: 'free' }
```

The token sits in `.env` (`LAB_TOKEN=lab-token`), loaded with `--env-file` as in Chapter 10. Keep `.env` out of git and never log a request's headers. To try a request before coding it, use the cURL Builder.

12.3 Pagination

APIs split long lists into **pages** and tell you how to ask for the next one. Check the docs for which of four styles an API uses, and follow its lead instead of guessing.



Figure 12.2 Four ways APIs page through results. GitHub and the Lab Shop use the `Link` header.

Follow the Link header with an async generator

Chapter 8's async generators fit pagination perfectly. The function fetches a page, hands out its items one at a time with `yield*`, then follows `rel="next"` until there isn't one. Callers just loop.

helpers.js (add)

```
export async function* paginate(url, options = {}) {
  while (url) {
    const res = await fetch(url, options);            // 12.4 adds retries
    if (!res.ok) throw new HttpError(res.status, url);
    const data = await res.json();
    yield* (Array.isArray(data) ? data : data.items);  // one item at a time
    const link = res.headers.get('link');              // <...>; rel="next"
    const next = link?.match(/<([^>]+)>;\s*rel="next"/)?.[1];
    url = next ? new URL(next, url).href : null;       // stop on the last page
  }
}
```

all-products.js LAB

```
import { paginate } from './helpers.js';
const url = 'http://localhost:3000/api/products';
const names = [];
for await (const item of paginate(url)) {
  names.push(item.name);
}
console.log(names.length, names.at(-1));
```

OUTPUT

```
$ node all-products.js
10 Sun Hat
```

One loop, three requests: the lab's terminal logs `/api/products`, then `?page=2` and `?page=3`.

A real API. GitHub pages its REST API the same way (up to 100 items per page, 60 requests an hour without a token, and a required `user-agent`), so `paginate()` works on it unchanged. Building a query string yourself? `url.searchParams.set()` encodes spaces and `&` for you.

12.4 Rate Limits, Retries and Timeouts

Servers protect themselves with rate limits. GitHub reports yours in `x-ratelimit-remaining` and `x-ratelimit-reset` (epoch seconds: the Timestamp Converter reads them). Go over and you get **429 Too Many Requests**, often with a `Retry-After` header: seconds to wait, or a date.

You get	Retry?
`429` **Too Many Requests**	Yes, after `Retry-After`: the server says when to come back
`500` `502` `503` `504`, **network errors, timeouts**	Yes, with backoff: hiccups usually pass
`400` `401` `403` `404`	No: fix the request or credentials (GitHub's 403 can mean a rate limit; check its headers)

Exponential backoff doubles the wait after each failure and **jitter** adds a random slice, so clients don't retry in lockstep. Cap the tries, and give each attempt an `AbortSignal.timeout(ms)`: a stuck request then fails with a `TimeoutError` instead of hanging.

helpers.js (add) LAB

```
const RETRY_STATUS = new Set([429, 500, 502, 503, 504]);

export async function fetchWithRetry(url, options = {}) {
  const { retries = 3, timeoutMs = 10_000, ...init } = options;
  for (let attempt = 1; ; attempt++) {
    let res;
    try {
      res = await fetch(url, { ...init, signal: AbortSignal.timeout(timeoutMs) });
      if (!RETRY_STATUS.has(res.status) || attempt > retries) return res;
    } catch (error) {
      if (attempt > retries) throw error;     // network error or TimeoutError
    }
    const retryAfter = Number(res?.headers.get('retry-after'));  // seconds
    const backoff = 2 ** attempt * 250 + Math.random() * 250;   // plus jitter
    const wait = retryAfter > 0 ? retryAfter * 1000 : backoff;
    const why = res?.status ?? 'an error';
    console.error(`attempt ${attempt} got ${why}, retrying in ${Math.round(wait)} ms`);
    await sleep(wait);                        // from node:timers/promises
  }
}
```

OUTPUT

```
$ node retry-demo.js
attempt 1 got 503, retrying in 1000 ms
attempt 2 got 503, retrying in 1000 ms
200 { ok: true, call: 3 }
```

`retry-demo.js` logs `fetchWithRetry('http://localhost:3000/api/flaky')`. Retries go to stderr, never into your data.

12.5 A Short Look at GraphQL

REST has many URLs with fixed answers. **GraphQL** has one: you `POST` a **query** naming exactly the fields you want, plus **variables**. GitHub's GraphQL API needs a token, even for public data.

graphql.js · needs GITHUB_TOKEN in .env (not run with one for this book)

```
const query = `query ($owner: String!, $name: String!) {
  repository(owner: $owner, name: $name) { stargazerCount } }`;
const res = await fetch('https://api.github.com/graphql', {
  method: 'POST',
  headers: { authorization: `Bearer ${process.env.GITHUB_TOKEN}`,
    'content-type': 'application/json', 'user-agent': 'js-course-bot/1.0' },
  body: JSON.stringify({ query, variables: { owner: 'nodejs', name: 'node' } }),
});
const body = await res.json();                // errors may come with a 200
if (!res.ok || body.errors) console.log(res.status, body.message ?? body.errors);
else console.log(body.data.repository);   // { stargazerCount: ... }
```

OUTPUT

```
$ node graphql.js
401 Bad credentials
```

EXERCISES

Try each one before you check the answer.

1. **WARM-UP** Use `getJson()` to print "Hiking Boot costs $149.99", then handle product 99 politely.
2. **PRACTICE** Write `whoami.js` : call `/api/me` with the token from `.env` . If it's missing or wrong, print "Check LAB_TOKEN in .env" and exit with code 1.
3. **CHALLENGE** With `paginate()` , count the lab's in-stock and sold-out products. Then print the 10 newest release tags of `nodejs/node` , five per page.

Answers: page 108

KEY TAKEAWAYS

- `fetch` only rejects when no response arrives: check `res.ok` every time.
- Send JSON with a method, a `content-type` and `JSON.stringify` ; keep tokens in `.env` .
- Follow the API's own pagination; an async generator turns pages into one loop.
- Retry 429, 5xx and timeouts with capped backoff and jitter, never a 4xx mistake.

Go deeper: APIs explained · HTTP headers · Rate limiting · API integration · JSON Formatter

13



PART 3 · BEYOND THE BROWSER

Scraping Web Pages with Cheerio

Collect data from web pages politely: check the rules, fetch HTML, pick out fields with CSS selectors, follow pages and store clean results in SQLite.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

13.1 Scrape responsibly

13.2 Give the Lab Shop a website

13.3 Fetch and parse with Cheerio

13.4 Follow pages and read structured data

13.5 Store results in SQLite

13.6 Test your parser with saved HTML

13.1 Scrape Responsibly

Web scraping means fetching pages with code and pulling out the data you need. Price trackers, research datasets and search engines all do it. So before you write a scraper, ask one question: **is there an API?** An official API or data export is faster, steadier and clearly allowed. Scrape when there isn't one.

Read robots.txt the way the standard says

Sites publish crawling rules at `/robots.txt`, a format standardized in 2022 as RFC 9309. Here's the file the Lab Shop will serve:

```
1 User-agent: *          # who the group below is for
2 Disallow: /admin       # paths that start with /admin
3 Crawl-delay: 1         # seconds between requests
```

1. **User-agent.** Groups match your bot's name, ignoring case; `*` covers everyone else.
2. **Disallow.** The longest matching rule wins, and `Allow` wins a tie.
3. **Crawl-delay.** Not part of RFC 9309, but a clear request. Honor it.

Figure 13.1 A robots.txt file. If fetching it returns a 4xx, there are no rules; a 5xx means assume everything is off limits. Cache it for at most 24 hours. It's a request, not a lock: the standard says it isn't access control. Test any path with the Robots.txt Tester.

Before you scrape

- [ ] Look for an official API or export first
- [ ] Read robots.txt and follow your group's rules
- [ ] Read the terms, especially on automated access
- [ ] Name your bot and a contact in the User-Agent
- [ ] Go slow: one request at a time, with a delay
- [ ] Collect only what you need, and no personal data without a lawful reason

Play fair

Public doesn't mean free for any use. Copyright, database rights, privacy laws such as the GDPR and a site's terms can apply to pages anyone can open. Take the facts you need at a polite pace, and leave personal data out unless you have a lawful basis to process it. Is web scraping legal? goes further. This isn't legal advice.

13.2 Give the Lab Shop a Website

Practice on a site you own. In `lab.js`, replace the comment `// Chapters 13 and 14 add website routes here.` with this block. It adds robots.txt, list pages at `/products?page=N` and a page per product.

lab.js (lines 80–104)

```
// ── The website to scrape (added in Chapter 13) ──
if (path === '/robots.txt') {
  res.writeHead(200, { 'content-type': 'text/plain' });
  return res.end('User-agent: *\nDisallow: /admin\nCrawl-delay: 1\n');
}
if (path === '/products') {
  if (!pageOk) return send(404, html('Not found', '<h1>No such page</h1>'));
  const items = pageOf(page).map(({ id, name, price }) =>
    `<li class="product"><a href="/products/${id}">${name}</a>` +
    ` <span class="price">${usd(price)}</span></li>`);
  const nextUrl = `/products?page=${page + 1}`;
  const next = page < PAGES ? `<a class="next" href="${nextUrl}">Next</a>` : '';
  return send(200, html(`Products, page ${page}`,
    `<h1>Lab Shop</h1><ul class="products">${items.join('')}</ul>${next}`));
}
const product = byId(path.match(/^\/products\/(\d+)$/)?.[1]);
if (product) {
  const { name, price, stock } = product;
  const ld = { '@context': 'https://schema.org', '@type': 'Product', name,
    offers: { '@type': 'Offer', price: price.toFixed(2), priceCurrency: 'USD',
      availability: `https://schema.org/${stock ? 'InStock' : 'OutOfStock'}` } };
  return send(200, html(name, `<h1>${name}</h1><p class="price">${usd(price)}</p>` +
    `<p class="stock">${stock ? `${stock} in stock` : 'Sold out'}</p>` +
    `<script type="application/ld+json">${JSON.stringify(ld)}</script>`));
}
```

Restart the lab, open `localhost:3000/products`, then check the rules with `robots-parser` from npm:

robots.js LAB

```
import parseRobots from 'robots-parser';
const url = 'http://localhost:3000/robots.txt';
const res = await fetch(url);
const robots = parseRobots(url, await res.text());
const bot = 'js-course-bot';
console.log(robots.isAllowed(
  'http://localhost:3000/admin', bot));
console.log(robots.getCrawlDelay(bot));
```

OUTPUT

```
$ node robots.js
false
1
```

No to `/admin`, one second between requests. Chapter 16 adds the 4xx and 5xx rules from 13.1.

13.3 Fetch and Parse with Cheerio

`fetch` gets you HTML as one long string. **Cheerio** (`npm install cheerio`, version 1.2) parses it into a tree you query with CSS selectors, the ones you'd use in a stylesheet. It doesn't run the page's JavaScript; that's Chapter 14.

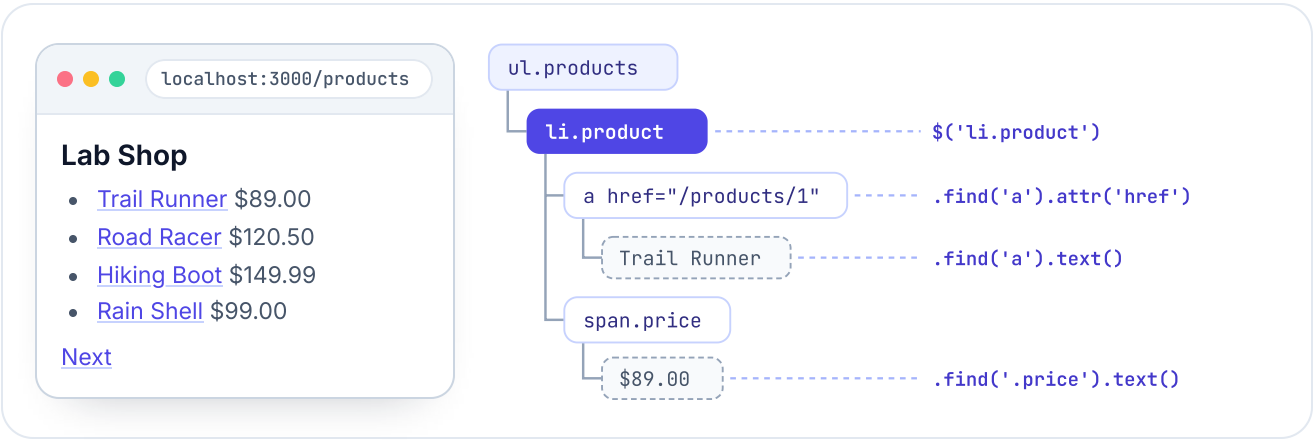


Figure 13.2 What the browser shows, and the tree Cheerio builds for one product. Each selector on the right reads one node.

Selectors match by tag (`h1`), class (`.price`), id (`#list`), attribute (`[type="application/ld+json"]`) and nesting (`li.product a`). Keep parsing in a function that takes HTML and returns data, with no network inside; 13.6 shows why.

parse.js

```
import * as cheerio from 'cheerio';
export function parseProducts(html) {
  const $ = cheerio.load(html);
  const products = $('li.product').map((i, el) => {
    const link = $(el).find('a');
    return {
      id: Number(link.attr('href').split('/').at(-1)),        // '/products/1'
      name: link.text().trim(),
      price: Number($(el).find('.price').text().replace(/[^\d.]/g, '')),
      href: link.attr('href'),
    };
  }).get();                                  // .get(): a plain array
  return { products, next: $('a.next').attr('href') };   // undefined on the last page
}
```

Coming from Python

Cheerio is Node's BeautifulSoup: `$('li.product')` is `soup.select('li.product')`, and `.text()` and `.attr('href')` are `.get_text()` and `tag.get('href')` (see the BeautifulSoup tutorial).

13.4 Follow Pages and Read Structured Data

A crawler repeats one loop: fetch a page, parse it, keep the data, find the next link, wait, go again. The lab asks for one second between requests, so the script sleeps that long.

crawl.js LAB

```
import { writeFile } from 'node:fs/promises';
import { setTimeout as sleep } from 'node:timers/promises';
import { HttpError } from './helpers.js';
import { parseProducts } from './parse.js';

const headers = { 'user-agent': 'js-course-bot/1.0' };
const rows = [];
let url = 'http://localhost:3000/products';
while (url) {
  const res = await fetch(url, { headers });
  if (!res.ok) throw new HttpError(res.status, url);
  const { products, next } = parseProducts(await res.text());
  rows.push(...products);
  console.log(`${products.length} products from ${url}`);
  url = next ? new URL(next, url).href : null;
  if (url) await sleep(1000);                  // the lab's Crawl-delay
}
await writeFile('products.json', JSON.stringify(rows, null, 2));
console.log(`saved ${rows.length} products to products.json`);
```

OUTPUT

```
$ node crawl.js
4 products from http://localhost:3000/products
4 products from http://localhost:3000/products?page=2
2 products from http://localhost:3000/products?page=3
saved 10 products to products.json
```

Product pages often carry **JSON-LD**, a block of structured data written for search engines. It's usually cleaner than the visible HTML. Cheerio 1.x also has `$.extract()`, which reads several fields in one call. Test JSON-LD on any page with the Schema Extractor.

json-ld.js LAB

```
import * as cheerio from 'cheerio';
const url = 'http://localhost:3000/products/2';
const res = await fetch(url);
const $ = cheerio.load(await res.text());
const ld = JSON.parse(
  $('script[type="application/ld+json"]').text());
console.log(ld.name, ld.offers.price);
console.log(ld.offers.availability);
console.log($.extract({ name: 'h1' }));
```

OUTPUT

```
$ node json-ld.js
Road Racer 120.50
https://schema.org/OutOfStock
{ name: 'Road Racer' }
```

13.5 Store Results in SQLite

A JSON file is fine for one run. For data you add to every day, use a database. Node 26 ships **SQLite** as `node:sqlite`: one file, no server, nothing to install.

save-db.js LAB

```
import { readFile } from 'node:fs/promises';
import { Database } from 'node:sqlite';       // Node 26.0 to 26.10: see below

const rows = JSON.parse(await readFile('products.json', 'utf8'));
const db = new Database('shop.db');
db.exec(`CREATE TABLE IF NOT EXISTS products (
  id INTEGER PRIMARY KEY, name TEXT NOT NULL, price REAL NOT NULL)`);
const upsert = db.prepare(`INSERT INTO products (id, name, price) VALUES (?, ?, ?)
  ON CONFLICT(id) DO UPDATE SET name = excluded.name, price = excluded.price`);
for (const { id, name, price } of rows) upsert.run(id, name, price);

const { n } = db.prepare('SELECT COUNT(*) AS n FROM products').get();
const cheap = db.prepare('SELECT name FROM products WHERE price < ? ORDER BY price');
console.log(`${n} products; under $50:`, cheap.all(50).map((r) => r.name).join(', '));
db.close();
```

OUTPUT

```
$ node save-db.js
10 products; under $50: Wool Socks, Water Bottle, Sun Hat, Head Lamp
$ node save-db.js
10 products; under $50: Wool Socks, Water Bottle, Sun Hat, Head Lamp
```

Run it twice: still 10. The **upsert** (`INSERT ... ON CONFLICT DO UPDATE`) updates rows it has seen instead of duplicating them.

> ⚠ **Common mistake**
>
> Building SQL by gluing strings, as in `` `VALUES ('${name}')` ``. A product called `Kid's Boot` breaks the query, and crafted text can rewrite it (SQL injection). Use `?` **placeholders** and pass the values to `run()`: SQLite fills them in safely.

The classes were renamed `Database` and `Statement` in Node 26.11; on earlier versions write `import { DatabaseSync as Database } from 'node:sqlite'` (that's how we tested this file, on Node 26.0).

13.6 Test Your Parser with Saved HTML

Sites change their HTML without telling you, and a broken parser returns empty or wrong data instead of crashing. Save a page once (`curl -o fixtures/products-1.html http://localhost:3000/products` , or `curl.exe` in PowerShell) and test `parseProducts` against it. Because it's pure, the test needs no network and runs in milliseconds. When the site changes, save a new fixture, watch the test fail, fix the selectors and see it pass.

parse.test.js

```
import { test } from 'node:test';
import assert from 'node:assert/strict';
import { readFile } from 'node:fs/promises';
import { parseProducts } from './parse.js';
const file = new URL('./fixtures/products-1.html', import.meta.url);
const { products, next } = parseProducts(await readFile(file, 'utf8'));
test('finds the four products on page 1', () => {
  assert.equal(products.length, 4);
  assert.deepEqual(products[0],
    { id: 1, name: 'Trail Runner', price: 89, href: '/products/1' });
});
test('finds the link to page 2', () => {
  assert.equal(next, '/products?page=2');
});
```

OUTPUT

```
$ node --test parse.test.js
✔ finds the four products on page 1 (0.9ms)
✔ finds the link to page 2 (0.1ms)
ℹ tests 2
... then suites 0, pass 2, fail 0 and four more summary lines
```

EXERCISES

Try each one before you check the answer.

1 WARM-UP Without a package, print the paths the lab's robots.txt disallows for `User-agent: *`.

2 PRACTICE Make `crawl.js` read each product's `availability` from JSON-LD and list the sold-out ones.

3 CHALLENGE Upsert `{ id, name, price, availability }` into SQLite twice (the count stays 10), then export products under $50 with `toCsv`.

Answers: page 109

KEY TAKEAWAYS

- Look for an API first, then read robots.txt and the terms; name your bot, go slow.
- Cheerio turns HTML into a tree you query with CSS selectors: `.find()`, `.text()`, `.attr()`.
- Follow next links with a delay, and prefer JSON-LD when a page has it.
- Store with upserts and placeholders, and test pure parsers against saved pages.

Go deeper: Web scraping 101 · Clean data · Node.js and proxies · The Scraping Blueprint · Use case

14



PART 3 · BEYOND THE BROWSER

Headless Browsers with Playwright

Drive a real browser from Node.js for pages that need JavaScript: wait for content, click, capture network data and take screenshots.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

14.1 When you need a browser
14.2 Your first Playwright script
14.3 Capture the data behind the page
14.4 Automate politely

14.1 When You Need a Browser

Some pages arrive empty and build their content with JavaScript. `fetch` doesn't run JavaScript, so it only sees the shell. See it in the lab: paste this route below Chapter 13's block.

lab.js (lines 106–113)

```
// ── A page built by browser JavaScript (added in Chapter 14) ──
if (path === '/app') {                    // the list only appears after JavaScript runs
  return send(200, html('Lab app', `<h1>Lab app</h1><ul id="list"></ul><script>
    fetch('/api/products').then((r) => r.json()).then((data) => {
      document.querySelector('#list').innerHTML = data.items
        .map((p) => '<li class="product">' + p.name + '</li>').join('');
    });</script>`));
}
```

fetch: localhost:3000/app

```
<h1>Lab app</h1>
<ul id="list"></ul>
<script>fetch('/api/products')...</script>
```

0 items

browser: localhost:3000/app

Lab app

- Trail Runner
- Road Racer
- Hiking Boot
- Rain Shell

4 items

Figure 14.1 The same URL, two views. Cheerio on the fetched HTML finds `#list li` 0 times; a browser runs the script, which loads page 1 of the API and fills the list.

cheerio-app.js LAB

```
import * as cheerio from 'cheerio';

const url = 'http://localhost:3000/app';
const res = await fetch(url);
const $ = cheerio.load(await res.text());
console.log('items:', $('#list li').length);
```

OUTPUT

```
$ node cheerio-app.js
items: 0
```

If the data is...	Use	Why
In the HTML the server sends	`fetch` + Cheerio	Fast and light: one request per page
Loaded from a JSON API	Call that API	Cleanest of all; find it in DevTools' Network tab
Built by JavaScript, or behind clicks	A browser	Runs scripts, scrolls, clicks and logs in
Screenshots, testing your own site	A browser	Only a browser renders the page

A browser costs far more time, memory and bandwidth per page, so reach for it last. Browser automation vs HTTP requests compares the two, and the Website Technology Checker shows what a site is built with.

14.2 Your First Playwright Script

Playwright drives real browsers from code. Run `npm install playwright`, then `npx playwright install chromium` once. It's headless unless you pass `{ headless: false }`.

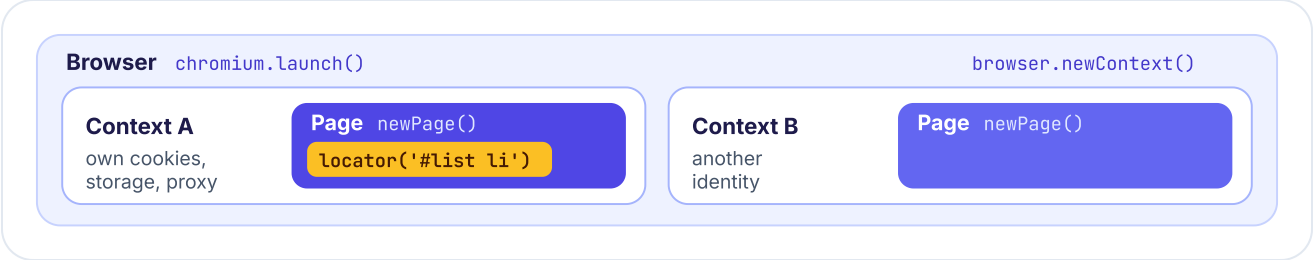


Figure 14.2 A browser holds **contexts** (separate sessions with their own cookies), contexts hold pages, and a **locator** finds elements on a page.

first-browser.js — LAB

```
import { chromium } from 'playwright';
const browser = await chromium.launch();            // headless by default
try {
  const page = await browser.newPage();
  const api = page.waitForResponse('**/api/products'); // listen first (14.3)
  await page.goto('http://localhost:3000/app');
  const items = page.locator('#list li');
  await items.first().waitFor();                     // wait for JavaScript
  console.log(await items.count(), await items.allTextContents());
  const data = await (await api).json();             // the data behind the page
  console.log((await api).status(), data.pages, data.items[0]);
} finally {
  await browser.close();                             // even after errors
}
```

OUTPUT

```
$ node first-browser.js
4 [ 'Trail Runner', 'Road Racer', 'Hiking Boot', 'Rain Shell' ]
200 3 { id: 1, name: 'Trail Runner', price: 89, stock: 12 }
```

> ⚠ **Common mistake**
>
> Forgetting `browser.close()`. An open browser keeps Node running, so the script never ends and you have to press `Ctrl` + `C`. Close it in `finally`, as above, whether the steps succeed or fail.

14.3 Capture the Data Behind the Page

The list on `/app` comes from `/api/products`, so the script also catches that response (start listening before `goto`) and reads its JSON: clean data, no HTML. You can often skip the browser and call that API with `fetch` (Chapter 12); if not, `page.route()` can block images and fonts.

14.4 Automate Politely

Good automation

- Testing and monitoring your own site
- Screenshots and PDFs of public pages
- Filling in your own forms and reports

Risky automation

- Solving or bypassing CAPTCHAs
- Running accounts you don't own
- Hammering a site because a browser "looks human"

Play fair

A browser doesn't make you invisible. Sites spot automation from browser fingerprints and behavior (see yours), and anti-bot systems are built to catch it. Chapter 13's rules still apply: robots.txt, the terms, a slow pace and no personal data without a lawful reason. A block is an answer, not a puzzle to solve. This isn't legal advice.

EXERCISES

Try each one before you check the answer.

1. **WARM-UP** Open `/app`, wait for the list, and save a full-page screenshot as `app.png`.
2. **PRACTICE** Make two contexts with different `userAgent` values, open `/api/echo` in each and print the `agent` the lab saw.
3. **CHALLENGE** Time `https://example.com` with and without blocking images, fonts and stylesheets, using `performance.now()`.

Answers: page 110

KEY TAKEAWAYS

- Use `fetch` and Cheerio when the HTML or an API has the data; a browser only when JavaScript builds it.
- Playwright goes browser, context, page, locator; close the browser in `finally`.
- Actions wait for elements; reading doesn't, so wait for content before you read it.
- The cleanest data is often the page's own API: capture it, or call it directly.

Go deeper: Headless browsing · Playwright scraping guide · Playwright vs Puppeteer · Screenshot tool

15



PART 3 · BEYOND THE BROWSER

Proxies, Concurrency and Scheduling

Route requests through proxies the right way, run many tasks without overloading anyone and put your scripts on a schedule.

PART 1
Fundamentals

PART 2
Modern JavaScript

PART 3
Beyond the Browser

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

YOU ARE HERE

IN THIS CHAPTER

15.1 What a proxy does for your code

15.2 Proxies with `fetch`

15.3 Rotation and sticky sessions

15.4 Concurrency without chaos

15.5 Run it on a schedule

15.1 What a Proxy Does for Your Code

A proxy is a server that makes requests for you. Your script sends each request to the proxy, the proxy forwards it, and the website sees the proxy's IP address instead of yours. You'll want one when a site shows different content by country, when you need to spread load across addresses, or when a site serves cloud-server traffic differently from home connections and you need to see what its visitors see.

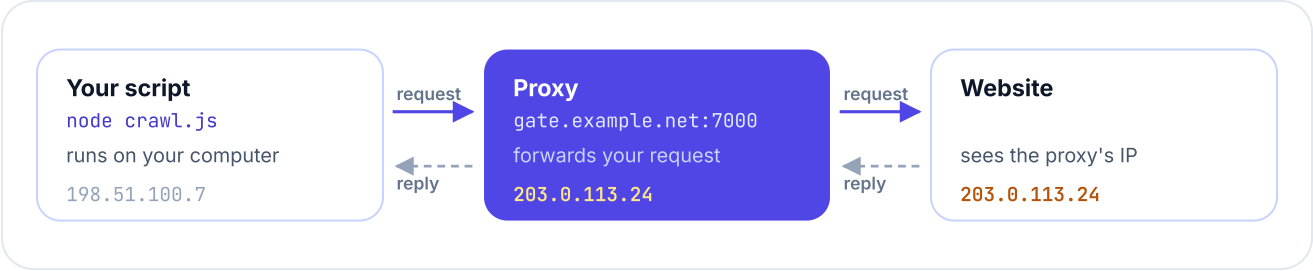


Figure 15.1 The site only ever sees the proxy's address. Addresses are from the ranges reserved for documentation.

Proxies differ in where their addresses come from: **datacenter** (cloud servers: fast, cheap, easy to spot), **residential** (home connections, shared with permission), **ISP** (datacenter speed, home-provider addresses) and **mobile** (carrier networks). Types of proxies compares them. Node.js gives you three ways to use one:

Option	Proxy types	Best for
`--use-env-proxy`	HTTP and HTTPS proxies, from `HTTPS_PROXY`	One proxy, no code changes
undici's `ProxyAgent`	HTTP and HTTPS; SOCKS5 (experimental)	Scripts that switch proxies
Playwright's `proxy`	HTTP, HTTPS and SOCKS5	Browsers, per launch or context

15.2 Proxies with `fetch`

The quickest route needs no package. Put the proxy in `.env` as `HTTPS_PROXY=http://USER:PASS@gate.example.net:7000` and start Node with `--use-env-proxy`: every `fetch` to an `https://` address then goes through it.

ip-env.js

```
// No imports: with --use-env-proxy, Node's own fetch reads HTTPS_PROXY
const res = await fetch('https://api.ipify.org?format=json');
console.log(await res.json());
```

```
$ node --use-env-proxy --env-file=.env ip-env.js
{ ip: '203.0.113.24' }
```

OUTPUT

Example output: you'll see your proxy's exit IP. Compare it with your own on My IP.

For more control (several proxies, one per request, SOCKS5), use **undici**, the library Node's `fetch` is built on (`npm install undici`). Each `ProxyAgent` is one proxy, passed to `fetch` as its `dispatcher` .

ip.js

```
import { fetch, ProxyAgent } from 'undici';

const dispatcher = new ProxyAgent(process.env.PROXY_URL);   // one agent per proxy
try {
  const res = await fetch('https://api.ipify.org?format=json', { dispatcher });
  console.log(await res.json());
} catch (error) {
  let why = error;
  while (why.cause) why = why.cause;          // the deepest cause says why
  console.log(error.message, '·', why.message);
}
```

OUTPUT

```
$ node --env-file=.env ip.js
{ ip: '203.0.113.24' }
$ node --env-file=.env ip.js
fetch failed · Proxy response (407) !== 200 when HTTP Tunneling
```

The second run has a typo in the password. "fetch failed" says nothing; the deepest `cause` shows the proxy's 407, Proxy Authentication Required. A proxy that's down gives `connect ECONNREFUSED` .

Three habits save hours. **Encode passwords:** `encodeURIComponent('p@ss')` gives `p%40ss` (the Proxy Format tool converts whole lists). **Write SOCKS5 as a URL:** `socks5://USER:PASS@gate.example.net:1080` works in `ProxyAgent` , experimentally. **Don't expect secrecy:** Node's docs say its proxy support routes traffic; it isn't an anonymity feature.

⚠ COMMON MISTAKE **Mixing Node's `fetch` with undici's agent**

INSTEAD OF

```
import { ProxyAgent } from 'undici';
const dispatcher = new ProxyAgent(url);
await fetch(page, { dispatcher });
// Node's fetch + the package's agent
```

WRITE

```
import { fetch, ProxyAgent } from 'undici';
const dispatcher = new ProxyAgent(url);
await fetch(page, { dispatcher });
// one fetch, one version
```

Node's `fetch` runs on its own built-in copy of undici. Mixed with the package's agent on Node 26.0, HTTPS responses lost every header and arrived still gzipped. Import `fetch` from `undici` too, or call undici's `install()` once (15.3) so the global `fetch` is the package's.

Tip

Test a new proxy in the Proxy Checker before you debug your code. If it fails there, the problem isn't your script.

15.3 Rotation and Sticky Sessions

Rotating means a new exit IP for each request: good for independent pages such as listings and search results. **Sticky** means one IP for a whole session: needed for logins, carts and multi-step flows, where a changing address looks like a stolen session (sticky vs rotating).



Figure 15.2 Six requests, two policies. Rotation spreads load; a sticky session keeps one identity until it expires or you end it.

Most providers give you one gateway address and pick rotating or sticky from your username or port. Each provider's syntax differs, so copy it from your dashboard instead of guessing. With a list of separate proxies, rotate them yourself: add `ProxyPool` to `helpers.js` (imports at the top), with undici's `install()` so every `fetch` in your program is the package's.

helpers.js (add)

```
import { install, ProxyAgent } from 'undici';
install();                                  // one fetch for the whole program

export class ProxyPool {
  #agents;
  #next = 0;
  #sticky = new Map();
  constructor(proxyUrls) {
    this.#agents = proxyUrls.map((url) => new ProxyAgent(url));
  }
  rotate() {                                // round robin: A, B, A, B...
    const agent = this.#agents[this.#next];
    this.#next = (this.#next + 1) % this.#agents.length;
    return agent;
  }
  sticky(key) {                             // same key, same proxy, every time
    if (!this.#sticky.has(key)) this.#sticky.set(key, this.rotate());
    return this.#sticky.get(key);
  }
}
```

Then `fetch(url, { dispatcher: pool.rotate() })` takes the next proxy in line, and `pool.sticky('cart-42')` hands back the same one every time you pass that key. **Rotation isn't a free pass:** a ban or a 429 is an answer, not something to rotate around.

Proxies in a browser

Playwright takes a proxy at launch or per context: `browser.newContext({ proxy: { server, username, password } })`, where `server` is a URL such as `http://gate.example.net:7000`. One context per identity keeps cookies and IP together (rotating proxies in Playwright).

Picking a provider

The nine networks below are ProxyHorizon's shortlist. Check the live directory or compare them side by side, and size your needs with the bandwidth calculator.

Provider	Residential from	IP pool	Countries	Rating
Bright Data	$5.04/GB	72M+	195	4.3
Decodo	$8.50/GB	115M+	195	4.4
Geonode	$4/GB	30M+	190	4.2
IPRoyal	$7/GB	64M+	195	4.4
NodeMaven	$3.50/GB	30M+	195	4.4
Oxylabs	$8/GB	102M+	195	4.4
PYProxy	$3.30/GB	90M+	190	4.3
SOAX	$6.60/GB	191M+	195	4.4
Webshare	$7/month	10M+	50	4.4

Source ProxyHorizon directory, November 2026: residential entry prices (Webshare's plan is monthly), advertised pool sizes and user ratings out of 5. Not sure which type you need? Try the proxy finder.

15.4 Concurrency Without Chaos

`Promise.all` over 1,000 URLs opens 1,000 requests at once: rude to the site and a fast way to get blocked. `mapLimit` caps the work in flight with a few workers that share one queue. But **concurrency is not rate**: five at a time can still mean fifty a second, so `pace` spaces out the starts.

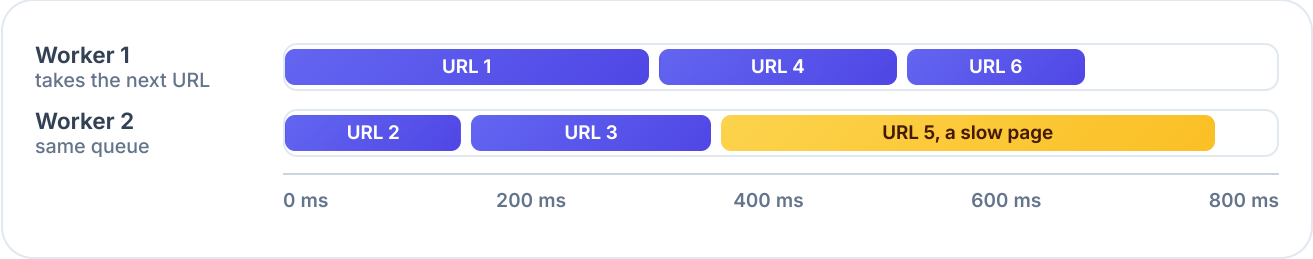


Figure 15.3 `mapLimit(urls, 2)`: each worker takes the next URL the moment it's free, so the slow URL 5 never holds up URL 6. A worked example, not a benchmark.

helpers.js (add)

```
export async function mapLimit(items, limit, fn) {   // at most `limit` at once
  const results = new Array(items.length);
  let index = 0;
  async function worker() {
    while (index < items.length) {
      const i = index++;                              // take the next item
      results[i] = await fn(items[i], i);
    }
  }
  await Promise.all(Array.from({ length: Math.min(limit, items.length) }, worker));
  return results;                                     // in the input's order
}
export function pace(ms) {                            // at most one start every ms
  let next = 0;
  return async () => {
    const now = Date.now();
    const at = Math.max(now, next);
    next = at + ms;
    await sleep(at - now);
  };
}
```

limits.js LAB

```
import { setTimeout as sleep } from 'node:timers/promises';
import { mapLimit, pace } from './helpers.js';

const ids = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const status = async (id) =>
  (await fetch(`http://localhost:3000/api/products/${id}`)).status;

const fast = await mapLimit(ids, 5, status);          // 5 at once, no gap
console.log('limit 5, no gap:    ', fast.join(' '));

await sleep(1500);                                    // let the lab's window reset
const wait = pace(250);                               // at most 4 starts a second
const polite = await mapLimit(ids, 2, async (id) => {
  await wait();
  return status(id);
});
console.log('limit 2, 250 ms gap:', polite.join(' '));
```

OUTPUT

```
$ node limits.js
limit 5, no gap:     200 200 200 200 200 429 429 429 429 429
limit 2, 250 ms gap: 200 200 200 200 200 200 200 200 200 200
```

The lab allows five API calls a second; if a run straddles a second, the 429s move.

15.5 Run It on a Schedule

A `setInterval` loop lasts only as long as its terminal. For real schedules, use **cron** on an always-on machine (Task Scheduler on Windows), `node-cron` or `croner` inside an app that runs all day, or **GitHub Actions** in the cloud (Chapter 16 sets it up). They share one five-field format:



1. **Minute.** 30
2. **Hour.** 7, so 07:30
3. **Day of month.** Any
4. **Month.** Any
5. **Day of week.** 1–5, Monday to Friday

Figure 15.4 A cron line for weekdays at 07:30; `*/30 * * * *` means every 30 minutes. GitHub Actions reads it in UTC unless you add a time zone such as `Europe/London`, runs schedules at most every 5 minutes, and can start them late when it's busy, especially on the hour.

EXERCISES

Try each one before you check the answer.

1. **WARM-UP** Turn `gate.example.net:7000:student:p@ss` into a proxy URL with `encodeURIComponent`. Check it with the Proxy Format tool.
2. **PRACTICE** Run `ip.js` through a practice proxy (`npx proxy -p 8001`), then stop the proxy and run it again. Explain both results.
3. **CHALLENGE** Fetch the ten lab products two at a time, 250 ms apart, through a two-proxy `ProxyPool`, counting each proxy's requests. Then write a cron line and time zone for weekdays at 07:30 where you live.

Answers: page 111

KEY TAKEAWAYS

- A proxy changes the IP a site sees, not the rules: robots.txt, terms and rate limits still apply.
- `--use-env-proxy` for one proxy, `ProxyAgent` for control; read the deepest `cause`.
- Rotate for independent pages, stay sticky for sessions, and never guess a provider's syntax.
- Concurrency is not rate: cap work with `mapLimit`, space out starts with `pace`.

Go deeper: Why proxies · Proxy speed tests · Choosing a provider · Ultimate Proxy Handbook

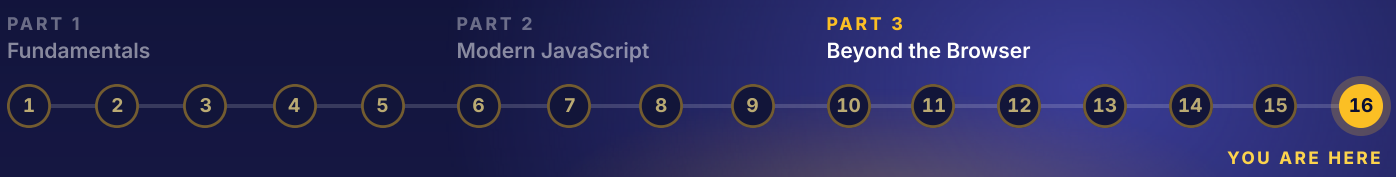
16



PART 3 · BEYOND THE BROWSER

Three Projects to Build and Ship

Put it all together: an API client CLI, a polite scraper with proxy rotation and a scheduled automation bot, then ship them responsibly.



IN THIS CHAPTER

16.1 Project 1: repo-pulse, an API client CLI
16.2 Project 2: shop-crawler, a polite scraper
16.3 Project 3: site-watch, a scheduled bot
16.4 Ship it responsibly

Three projects built from parts you already have. Type them in, run them against the Lab Shop or your own site, then extend them with the challenges at the end.

repo-pulse
A command-line tool that checks GitHub repos through their API.

shop-crawler
A polite scraper with robots.txt, pacing, proxies and SQLite.

site-watch
A bot that checks your site each weekday and posts a digest.

16.1 Project 1: `repo-pulse`, an API Client CLI

`repo-pulse` prints stars, open issues, latest release and last push for each `owner/name` you give it, from GitHub's `/repos/{owner}/{repo}` and `/releases/latest` (a 404 there means no releases yet). It reuses `fetchWithRetry` (12.4), `mapLimit` (15.4) and `HttpError` (Chapter 6).

repo-pulse.js

```
#!/usr/bin/env node
import { parseArgs, styleText } from 'node:util';
import { fetchWithRetry, mapLimit, HttpError } from './helpers.js';
const { values, positionals: repos } = parseArgs({
  options: { json: { type: 'boolean' } }, allowPositionals: true });
if (repos.length === 0) {
  console.error('usage: repo-pulse owner/repo [owner/repo...] [--json]');
  process.exit(2);
}
const headers = { 'user-agent': 'repo-pulse/1.0' };
const token = process.env.GITHUB_TOKEN;              // optional (see below)
if (token) headers.authorization = `Bearer ${token}`;
let left;                                            // calls left this hour
async function github(path) {                        // the data, or null for a 404
  const res = await fetchWithRetry(`https://api.github.com${path}`, { headers });
  left = res.headers.get('x-ratelimit-remaining');
  if (res.status === 404) return null;
  if (!res.ok) throw new HttpError(res.status, path);
  return res.json();
}
const rows = await mapLimit(repos, 3, async (name) => {
  const repo = await github(`/repos/${name}`);
  if (!repo) return { repo: name, stars: 'not found' };
  const release = await github(`/repos/${name}/releases/latest`);  // 404: none
  return { repo: repo.full_name, stars: repo.stargazers_count,
    issues: repo.open_issues_count,                  // counts pull requests too
    release: release?.tag_name ?? 'none', pushed: repo.pushed_at.slice(0, 10) };
});
if (values.json) console.log(JSON.stringify(rows, null, 2));
else console.table(rows);
console.error(styleText('dim', `calls left: ${left}`, { stream: process.stderr }));
```

```
$ node repo-pulse.js cheeriojs/cheerio microsoft/playwright                TERMINAL
```

(index)	repo	stars	issues	release	pushed
0	'cheeriojs/cheerio'	30552	82	'v1.2.0'	'2026-11-20'
1	'microsoft/playwright'	100712	45	'v1.65.1'	'2026-11-21'

```
calls left: 36
```

Your numbers will differ (this ran on 21 November 2026). The last line goes to stderr, so `--json > repos.json` saves clean JSON. Without a token GitHub allows 60 calls an hour per IP, two per repo; put a personal access token in `.env` as `GITHUB_TOKEN` for 5,000. A `"bin"` entry and `npm link` (Chapter 10) make it a real command.

16.2 Project 2: `shop-crawler`, a Polite Scraper with Proxy Rotation

`shop-crawler` walks the Lab Shop like a careful bot: rules first, then list pages, then products, one request a second. Point it only at sites you own or may crawl.



Figure 16.1 The pipeline. Every request goes through one function, `get()`, so politeness can't be skipped.

It saves through `store.js`, 13.5's upsert wrapped in a function:

store.js

```
import { Database } from 'node:sqlite';    // Node 26.0 to 26.10: see 13.5
export function saveProducts(file, rows) {      // 13.5's upsert, then a count
  const db = new Database(file);
  db.exec(`CREATE TABLE IF NOT EXISTS products (id INTEGER PRIMARY KEY,
    name TEXT, price REAL, availability TEXT)`);
  const upsert = db.prepare(`INSERT INTO products VALUES (?, ?, ?, ?)
    ON CONFLICT(id) DO UPDATE SET name = excluded.name, price = excluded.price,
    availability = excluded.availability`);
  for (const p of rows) upsert.run(p.id, p.name, p.price, p.availability);
  const { n } = db.prepare('SELECT COUNT(*) AS n FROM products').get();
  db.close();
  return n;
}
```

shop-crawler.js

```
import { mkdir, writeFile } from 'node:fs/promises';
import * as cheerio from 'cheerio';
import robotsParser from 'robots-parser';
import { fetchWithRetry, mapLimit, pace, ProxyPool, toCsv } from './helpers.js';
import { parseProducts } from './parse.js';               // Chapter 13
import { saveProducts } from './store.js';                // the previous page
const START = process.argv[2] ?? 'http://localhost:3000/products';
const AGENT = 'shop-crawler/1.0 (+https://example.com/bot)';
const proxies = process.env.PROXY_URLS?.split(',') ?? [];
const pool = proxies.length ? new ProxyPool(proxies) : null;
const t0 = performance.now();
let pages = 0;
const robotsUrl = new URL('/robots.txt', START).href;     // 0. the rules (13.1)
const r = await fetchWithRetry(robotsUrl, { headers: { 'user-agent': AGENT } });
if (r.status >= 500) throw new Error('robots.txt is failing: not crawling today');
const robots = robotsParser(robotsUrl, r.ok ? await r.text() : '');   // 4xx: no rules
const wait = pace((robots.getCrawlDelay(AGENT) ?? 1) * 1000);         // 1 s by default
async function get(url) {                                 // every request goes here
  if (!robots.isAllowed(url, AGENT)) throw new Error(`robots.txt says no: ${url}`);
  await wait();
  pages += 1;
  const res = await fetchWithRetry(url,
    { headers: { 'user-agent': AGENT }, dispatcher: pool?.rotate() });
  if (!res.ok) throw new Error(`HTTP ${res.status} for ${url}`);
  return res.text();
}
const found = [];                                         // 1. the list pages
for (let url = START; url; ) {
  const { products, next } = parseProducts(await get(url));
  found.push(...products);
  url = next && new URL(next, url).href;
}
const rows = await mapLimit(found, 2, async ({ id, href }) => {   // 2. detail pages
  try {
    const $ = cheerio.load(await get(new URL(href, START).href));
    const { name, offers } = JSON.parse($('script[type="application/ld+json"]').text());
    const availability = offers.availability.split('/').at(-1);
    return { id, name, price: Number(offers.price), availability };
  } catch (error) {
    console.error(`skipped ${href}: ${error.message}`);
  }
});
const good = rows.filter(Boolean);                        // 3. save and report
await mkdir('data', { recursive: true });
const total = saveProducts('data/shop.db', good);
await writeFile('data/products.csv', toCsv(good));
const secs = ((performance.now() - t0) / 1000).toFixed(1);
console.log(`${pages} pages, ${good.length} products (${total} saved), ${secs} s`);
```

```
$ node shop-crawler.js                                    TERMINAL
13 pages, 10 products (10 saved), 12.0 s
$ node --env-file=.env shop-crawler.js
13 pages, 10 products (10 saved), 12.0 s
```

Thirteen pages at one a second. The second run sends each request through the next of the two practice proxies in `PROXY_URLS` (A, B, A, B). Still 10 saved: the upsert updates rows.

> ⚠ **Common mistake**
>
> Pointing remote proxies at `localhost`. A proxy on the internet looks for its own localhost, not yours, so test rotation with practice proxies. For a real site you're allowed to crawl, pass its list URL as the first argument and your provider's URLs in `PROXY_URLS`.

16.3 Project 3: `site-watch`, a Scheduled Automation Bot

Each weekday morning, `site-watch` checks pages on **your own** site (status, response time, title, meta description, canonical and `noindex`) and posts a digest to a chat webhook: a private URL that turns a POST into a message. Slack's incoming webhooks take `{ "text": ... }`; Discord's take `{ "content": ... }`, up to 2,000 characters. GitHub Actions runs it from this file, with the webhook URL stored as a repository secret and the page list as a variable:

.github/workflows/site-watch.yml

```yaml
name: site-watch
on:
  schedule:
    - cron: '30 7 * * 1-5'           # weekdays at 07:30...
      timezone: 'Europe/London'      # ...London time (UTC if you leave it out)
  workflow_dispatch:                 # adds a "Run workflow" button for tests
jobs:
  watch:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-node@v7
        with:
          node-version: 26
      - run: npm ci
      - run: node site-watch.js
        env:
          WATCH_URLS: ${{ vars.WATCH_URLS }}
          WEBHOOK_URL: ${{ secrets.WEBHOOK_URL }}
```

Commit `package-lock.json` for `npm ci`. Schedules run on the default branch only; a public repo loses them after 60 days without activity.

site-watch.js LAB

```
import * as cheerio from 'cheerio';
import { mapLimit } from './helpers.js';

async function check(url) {                          // one digest line per page
  const t0 = performance.now();
  try {
    const res = await fetch(url, { headers: { 'user-agent': 'site-watch/1.0' },
      signal: AbortSignal.timeout(15_000) });
    const ms = Math.round(performance.now() - t0);
    const $ = cheerio.load(await res.text());
    const problems = [
      res.status !== 200 && `status ${res.status}`,
      ms > 2000 && 'slow',
      !$('title').text().trim() && 'no title',
      !$('meta[name="description"]').attr('content') && 'no description',
      !$('link[rel="canonical"]').attr('href') && 'no canonical',
      $('meta[name="robots"]').attr('content')?.includes('noindex') && 'noindex',
    ].filter(Boolean);
    return `${problems.length ? 'FIX' : 'OK '} ${url} ${ms} ms ${problems.join(', ')}`;
  } catch (error) {
    return `FIX ${url} unreachable (${error.name})`;
  }
}
const lines = await mapLimit(process.env.WATCH_URLS.split(','), 3, check);
const bad = lines.filter((line) => line.startsWith('FIX')).length;
const text = [`${bad} of ${lines.length} pages need attention`, ...lines].join('\n');
console.log(text);
const hook = process.env.WEBHOOK_URL;                // a secret: keep it in env
if (hook) {                                        // Slack takes text; Discord, content
  const body = hook.includes('discord') ? { content: text } : { text };
  const res = await fetch(hook, { method: 'POST', body: JSON.stringify(body),
    headers: { 'content-type': 'application/json' } });
  if (!res.ok) throw new Error(`webhook answered ${res.status}`);
}
if (bad > 0) process.exitCode = 1;                   // a failed run is an alert too
```

OUTPUT

```
$ node --env-file=.env site-watch.js
3 of 3 pages need attention
FIX http://localhost:3000/ 11 ms no description, no canonical
FIX http://localhost:3000/products 9 ms no description, no canonical
FIX http://localhost:3000/old-page 10 ms status 404, no title, no description, no canonical
```

A local test: `.env` lists three lab pages and sets `WEBHOOK_URL` to the lab's `/api/echo`, a stand-in for Slack. Lab pages lack descriptions and canonicals, so all fail and the run exits with code 1. The workflow and real webhooks weren't run for this book.

Spot-check pages with the Meta Tag, Canonical and Indexability checkers, or read up on SEO monitoring.

16.4 Ship It Responsibly

A laptop with cron is fine for a personal job. Anything that must run on time belongs on a small VPS with cron or systemd (proxies on a VPS) or on GitHub Actions, which needs no server at all. Wherever it runs, go through this list first:

- [ ] Secrets in env or GitHub secrets, never code
- [ ] `.env` listed in `.gitignore`
- [ ] Logs as JSON lines; errors to stderr
- [ ] A non-zero exit code on failure
- [ ] A timeout per request, a page limit per run
- [ ] A kill switch after repeated 403s or 429s
- [ ] Only the data you need, the rest deleted
- [ ] robots.txt, terms and rate limits checked

For a seat belt, Node's permission model: `node --permission --allow-fs-read=. --allow-fs-write=./data --allow-net shop-crawler.js` lets the crawler read its folder, write only to `data/` (create it first) and use the network. Anything else fails with `ERR_ACCESS_DENIED`. Node's docs call it a seat belt for code you trust, not a sandbox, and Node 26.0 still marks `--allow-net` experimental.

EXTENSION CHALLENGES — One per project. Answers are in the appendix.

1	WARM-UP	`repo-pulse`: add `--sort stars` and `--csv`, reusing `toCsv` from Chapter 10.
2	PRACTICE	`shop-crawler`: record each run in a `runs` table, and stop the crawl after three 429s in a row.
3	CHALLENGE	`site-watch`: post only what changed since yesterday, keeping yesterday's results in a JSON file (stateless vs stateful).

Answers: page 111

KEY TAKEAWAYS

- Real tools are small parts you already know, joined with care.
- Build politeness in: robots.txt first, one request at a time, a named User-Agent.
- Secrets stay in env and GitHub secrets; failures show up as non-zero exit codes.
- Test against the Lab Shop and practice proxies before touching a real site.

Where to next: The Scraping Blueprint · Ultimate Proxy Handbook · Proxies for AI Agents · Free tools

APPENDIX

Exercise Answers

Try each exercise before you look. If your code differs from the answer, run both: there's usually more than one right way, and the output tells you whether yours works. Lab Shop answers need `node lab.js` running in another terminal.

01 How JavaScript Runs

WARM-UP `node -v` prints `v26.` and a patch number, such as `v26.13.1`. `npm -v` prints an `11.` version (Node.js 26.13.1 comes with npm 11.20.0). `node -p "process.platform"` prints `darwin` on a Mac, `win32` on Windows (64-bit included) and `linux` on Linux.

PRACTICE One `console.log` per line. Your name, year and version will differ.

about-me.js

```
console.log('Ada Lovelace');              // Ada Lovelace
console.log(new Date().getFullYear());    // 2026
console.log(process.version);             // v26.0.0
```

CHALLENGE Three mistakes, three error types, in the words Node.js 26 uses:

The mistake	The message	The type
`consol.log('hi');`	`consol is not defined`	`ReferenceError`
`console.log('hi);`	`Invalid or unexpected token`	`SyntaxError`
`const name = 'Ada'; name();`	`name is not a function`	`TypeError`

02 Values, Variables and Operators

WARM-UP `'object'`, `'object'`, `false`, `'53'` and `15`. `typeof null` is the famous quirk; arrays really are objects (test for one with `Array.isArray`); `+` joins text when either side is a string, while `*` always does math.

PRACTICE Make the formatter once, then use it for every price:

format.js

```
const usd = new Intl.NumberFormat('en-US', { style: 'currency', currency: 'USD' });
console.log(usd.format(1234.5));   // $1,234.50
```

CHALLENGE Trim first, or the outer spaces turn into dashes too. For runs of spaces inside a name, such as `'Trail Runner'`, replace the regular expression `/\s+/g` instead of `' '` (Chapter 8).

slug.js

```
const name = '  Trail Runner  ';
console.log(name.trim().toLowerCase().replaceAll(' ', '-'));   // trail-runner
```

EXERCISE ANSWERS, CONTINUED

03 Control Flow and Functions

WARM-UP Test 15 first: a multiple of 15 is also a multiple of 3. It prints 1, 2, Fizz, 4, Buzz and so on, ending with 14 and FizzBuzz, one per line.

fizzbuzz.js

```
for (let i = 1; i <= 15; i++) {
  if (i % 15 === 0) console.log('FizzBuzz');
  else if (i % 3 === 0) console.log('Fizz');
  else if (i % 5 === 0) console.log('Buzz');
  else console.log(i);
}
```

PRACTICE A guard clause for other countries, then the rule from Figure 3.3:

shipping.js

```
function shippingCost(total, country = 'US') {
  if (country !== 'US') return 14.99;
  if (total >= 50) return 0;
  return 4.99;
}
console.log(shippingCost(49.99), shippingCost(50), shippingCost(80, 'CA'));
// 4.99 0 14.99
```

CHALLENGE Return from inside the loop the moment a try succeeds. Chapter 12's `fetchWithRetry` is the same idea, with a wait between tries.

try-times.js

```
function tryTimes(task, times) {
  for (let attempt = 1; attempt <= times; attempt++) {
    if (task()) return attempt;
  }
  return -1;
}
let calls = 0;
console.log(tryTimes(() => ++calls === 3, 5));  // 3: it succeeds on call 3
console.log(tryTimes(() => false, 4));          // -1: it never succeeds
```

04 Arrays, Objects and Array Methods

WARM-UP Keep, then reshape. The other two answers start with the same import.

cheap.js

```
import { products } from './products.js';
const cheap = products.filter((p) => p.price < 50).map((p) => p.name);
console.log(cheap);   // [ 'Wool Socks', 'Head Lamp', 'Water Bottle', 'Sun Hat' ]
```

EXERCISE ANSWERS, CONTINUED

04 Arrays, Objects and Array Methods

continued

PRACTICE Start the sum at 0 and add `price * stock` for each product:

value.js

```
const value = products.reduce((sum, p) => sum + p.price * p.stock, 0);
const usd = new Intl.NumberFormat('en-US', { style: 'currency', currency: 'USD' });
console.log(value, usd.format(value));   // 4588.96 $4,588.96
```

CHALLENGE Group on a ternary, then sort a copy of the sold-out names:

groups.js

```
const status = (p) => (p.stock > 0 ? 'inStock' : 'soldOut');
const groups = Object.groupBy(products, status);
console.log(groups.inStock.length, groups.soldOut.length);   // 8 2
console.log(groups.soldOut.map((p) => p.name).toSorted());  // [ 'Head Lamp', 'Road Racer' ]
```

05 Errors, Debugging and Testing

WARM-UP `ReferenceError` (a name that doesn't exist), `SyntaxError` (a typo found before anything runs) and `TypeError` (`find` found nothing, so there's no object to read `price` from).

PRACTICE Strip the `$` and the commas, convert, and refuse anything that isn't a number. `Number('')` is `0`, so check for empty text too. Export the function for the tests, then try it from a second file, which prints `1299` and `Sorry, Not a price: N/A`.

price.js

```
export function parsePrice(text) {
  const clean = text.replaceAll('$', '')
    .replaceAll(',', '').trim();
  const number = Number(clean);
  if (clean === '' || Number.isNaN(number)) {
    throw new RangeError(`Not a price: ${text}`);
  }
  return number;
}
```

use-price.js

```
import { parsePrice } from './price.js';
for (const text of ['$1,299.00', 'N/A']) {
  try {
    console.log(parsePrice(text));
  } catch (error) {
    console.log(`Sorry, ${error.message}`);
  }
}
```

CHALLENGE One test per path. `node --test` finds the file by its `.test.js` name, prints a ✔ for each test, then the totals: tests 2, pass 2, fail 0.

price.test.js

```
import { test } from 'node:test';
import assert from 'node:assert/strict';
import { parsePrice } from './price.js';
test('reads a formatted price', () => assert.equal(parsePrice('$1,299.00'), 1299));
test('rejects N/A', () => assert.throws(() => parsePrice('N/A'), RangeError));
```

06 Scope, Closures and Classes

WARM-UP The `var` loop prints `3 3 3` and the `let` loop prints `0 1 2`. The callbacks run after the loop has finished: `var` gives all three one shared `i`, by then 3, while `let` gives every lap its own.

EXERCISE ANSWERS, CONTINUED

06 Scope, Closures and Classes

continued

PRACTICE Return an object whose two methods share one closed-over count. Nothing else can reach it, so `counter.count` is `undefined`.

counter.js

```
function makeCounter() {
  let count = 0;                      // private: only the methods see it
  return {
    increment() { count += 1; return count; },
    current() { return count; },
  };
}
const counter = makeCounter();
counter.increment();
counter.increment();
console.log(counter.current(), counter.count);   // 2 undefined
```

CHALLENGE Keep the items private, return `this` so calls chain, and add a `toJSON` method, which `JSON.stringify` calls for you:

cart.js

```
import { products } from './products.js';
class Cart {
  #items = [];
  add(product, qty = 1) {
    this.#items.push({ ...product, qty });
    return this;                          // so calls can chain
  }
  get total() {
    return this.#items.reduce((sum, i) => sum + i.price * i.qty, 0);
  }
  toJSON() { return this.#items; }        // JSON.stringify calls this
  static fromJSON(json) {
    const cart = new Cart();
    for (const { qty, ...product } of JSON.parse(json)) cart.add(product, qty);
    return cart;
  }
}
const cart = new Cart().add(products[0], 2).add(products[9]);
const copy = Cart.fromJSON(JSON.stringify(cart));
console.log(cart.total, copy.total);     // 202 202
```

07 Asynchronous JavaScript

WARM-UP The plain `console.log` first, then every microtask in the order you queued it, then the timer:

predict.js

```
setTimeout(() => console.log('timeout'), 0);
Promise.resolve().then(() => console.log('then'));
queueMicrotask(() => console.log('microtask'));
console.log('sync');
```

OUTPUT

```
$ node predict.js
sync
then
microtask
timeout
```

EXERCISE ANSWERS, CONTINUED

07 Asynchronous JavaScript

continued

PRACTICE Wrap `setTimeout` in a promise, then `await` it inside a `for...of`. It prints 3, 2, 1 and `Liftoff!`, one second apart.

countdown.js

```
const sleep = (ms) => new Promise((resolve) => setTimeout(resolve, ms));
for (const n of [3, 2, 1]) {
  console.log(n);
  await sleep(1000);
}
console.log('Liftoff!');
```

CHALLENGE Count the fulfilled results: `node jobs.js` prints `3 ok, 2 failed`. Node's `sleep` takes a signal, so a job still running after one second is aborted: `node jobs.js 1500` prints `2 ok, 3 failed`, then `AbortError TimeoutError`, because the timeout is the abort's `cause`.

jobs.js

```
import { setTimeout as sleep } from 'node:timers/promises';
async function job(id, ms, fails = false) {
  await sleep(ms, undefined, { signal: AbortSignal.timeout(1000) });
  if (fails) throw new Error(`job ${id} failed`);
  return `job ${id} done`;
}
const slow = Number(process.argv[2] ?? 250);   // try 1500
const results = await Promise.allSettled([
  job(1, 200), job(2, 300, true), job(3, 100), job(4, 400, true), job(5, slow),
]);
const ok = results.filter((r) => r.status === 'fulfilled').length;
console.log(`${ok} ok, ${results.length - ok} failed`);
const last = results.at(-1);
if (last.status === 'rejected') {
  console.log(last.reason.name, last.reason.cause?.name);
}
```

08 Iterators, Generators, JSON and Regex

WARM-UP Count up by `step` instead of 1:

range.js

```
function* range(start, end, step = 1) {
  for (let n = start; n < end; n += step) yield n;
}
console.log([...range(0, 10, 3)]);   // [ 0, 3, 6, 9 ]
```

EXERCISE ANSWERS, CONTINUED

08 Iterators, Generators, JSON and Regex

continued

PRACTICE `stringify` turns the `Date` into a string; a reviver turns it back. A fixed date keeps the output the same for everyone.

saved.js

```
const text = JSON.stringify({ savedAt: new Date('2026-11-21T07:30:00Z') });
console.log(text);                  // {"savedAt":"2026-11-21T07:30:00.000Z"}
const back = JSON.parse(text, (key, value) =>
  key === 'savedAt' ? new Date(value) : value);
console.log(back.savedAt instanceof Date, back.savedAt.getUTCHours());   // true 7
```

CHALLENGE Map each match's groups to an object, and drop the commas before `Number`:

amounts.js

```
const text = 'Prices: USD 1,299.00 and EUR 89.50';
const price = /(?<currency>[A-Z]{3})\s(?<amount>[\d,]+\.\d{2})/g;
const prices = [...text.matchAll(price)].map(({ groups }) => ({
  currency: groups.currency,
  amount: Number(groups.amount.replaceAll(',', '')),
}));
console.log(prices);
// [ { currency: 'USD', amount: 1299 }, { currency: 'EUR', amount: 89.5 } ]
```

09 Modules, Packages and TypeScript

WARM-UP Export the function from one file and import it by name, keeping the `.js`. If Node stops with `SyntaxError: Cannot use import statement outside a module`, your `package.json` says `"type": "commonjs"`: run `npm pkg set type=module`.

money.js

```
export const usd = (n) => `$${n.toFixed(2)}`;
```

app.js

```
import { usd } from './money.js';
console.log(usd(89));   // $89.00
```

PRACTICE Six commands set it up, and the `tsconfig.json` beside them (the one Chapter 9 points to) configures `tsc`. Add an `index.ts` that imports cheerio, since `tsc` checks `.ts` files and stops with `error TS18003: No inputs were found` when there are none. Then `node --run start` runs it and `node --run check` checks it.

Terminal

```
npm init -y
npm pkg set type=module
npm install cheerio
npm install -D typescript @types/node
npm pkg set scripts.start="node index.ts"
npm pkg set scripts.check="tsc"
```

tsconfig.json

```
{
  "compilerOptions": {
    "module": "nodenext",
    "noEmit": true,
    "allowImportingTsExtensions": true,
    "erasableSyntaxOnly": true,
    "verbatimModuleSyntax": true,
    "types": ["node"]
  }
}
```

EXERCISE ANSWERS, CONTINUED

09 Modules, Packages and TypeScript

continued

CHALLENGE Type the product and the items, and keep everything else as in Chapter 6. `node cart.ts` runs it without looking at the types and prints `202 202`; `npx tsc` catches the string price without running anything: `cart.ts:25:30 - error TS2322: Type 'string' is not assignable to type 'number'.`

cart.ts

```ts
type Product = { name: string; price: number };
type Item = { product: Product; qty: number };

class Cart {
  #items: Item[] = [];
  add(product: Product, qty = 1): this {
    this.#items.push({ product, qty });
    return this;
  }
  get total(): number {
    return this.#items.reduce((sum, i) => sum + i.product.price * i.qty, 0);
  }
  toJSON(): Item[] { return this.#items; }
  static fromJSON(json: string): Cart {
    const cart = new Cart();
    const items = JSON.parse(json) as Item[];
    for (const { product, qty } of items) cart.add(product, qty);
    return cart;
  }
}
const cart = new Cart()
  .add({ name: 'Trail Runner', price: 89 }, 2)
  .add({ name: 'Sun Hat', price: 24 });
console.log(cart.total, Cart.fromJSON(JSON.stringify(cart)).total);
cart.add({ name: 'Day Pack', price: '64' });   // the mistake tsc should catch
```

10 Node.js Core and Command-Line Tools

WARM-UP `process.cwd()` follows you; `import.meta.dirname` follows the script. Run it from the course folder, then from the folder above:

where.js

```ts
console.log(process.cwd());
console.log(import.meta.dirname);
console.log(process.argv.slice(2));
```

OUTPUT

```
$ node where.js a b
/Users/you/js-course
/Users/you/js-course
[ 'a', 'b' ]
$ cd .. && node js-course/where.js a b
/Users/you
/Users/you/js-course
[ 'a', 'b' ]
```

EXERCISE ANSWERS, CONTINUED

10 Node.js Core and Command-Line Tools

continued

PRACTICE Move `toCsv` into `helpers.js` with `export` (Chapters 12 to 16 keep adding to that file), then read, convert and write. No `products.json` yet? Save Chapter 4's list with `JSON.stringify(products, null, 2)`. The CSV starts `id,name,price,stock`, then `1,Trail Runner,89,12`.

to-csv.js

```
import { readFile, writeFile } from 'node:fs/promises';
import { toCsv } from './helpers.js';
const rows = JSON.parse(await readFile('products.json', 'utf8'));
await writeFile('products.csv', toCsv(rows));
console.log(`${rows.length} rows written to products.csv`);   // 10 rows written ...
```

CHALLENGE Try the `user:pass@host:port` form first, since a `host:port:user:pass` password may hold an `@`. Add `parse()` above the loop, then check every part and the port's range. Given both formats, a port of 70000 and a line with no login, it prints the two URLs, a red `Skipped:` line for each bad one, and `echo $?` shows `1`.

proxyfmt.js (changes)

```
const AT = /^(?<user>[^:]+):(?<pass>.+)@(?<host>[^:@]+):(?<port>\d+)$/;
function parse(line) {                          // either format -> its parts
  const match = line.match(AT);
  if (match) return match.groups;
  const [host, port, user, ...rest] = line.split(':');
  return { host, port, user, pass: rest.join(':') };
}
// inside the loop, in place of the split and the old check:
const { host, port, user, pass } = parse(line.trim());
const n = Number(port);
if (!host || !user || !pass || !Number.isInteger(n) || n < 1 || n > 65535) {
  console.error(styleText('red', `Skipped: ${line}`));
  process.exitCode = 1;
  continue;
}
```

11 Build a Small API Server

edits to lab.js

WARM-UP Add a route after the 429 check, so health checks count toward the limit like any other call. `curl localhost:3000/api/health` then prints `{"ok":true,"uptime":0.975366292}`; `uptime` is in seconds and will differ.

lab.js (add)

```
if (path === '/api/health') return send(200, { ok: true, uptime: process.uptime() });
```

EXERCISE ANSWERS, CONTINUED

11 Build a Small API Server

continued

PRACTICE Filter first, then count the pages of what's left, and keep `q` in the `Link` header. `curl 'localhost:3000/api/products?q=HAT'` returns only the Sun Hat; `?q=o` finds five products on two pages, with `link: </api/products?q=o&page=2>; rel="next"`. The quotes stop zsh from reading the `?`.

lab.js (replace the /api/products route)

```
if (path === '/api/products') {
  const q = (url.searchParams.get('q') ?? '').toLowerCase();
  const found = PRODUCTS.filter((p) => p.name.toLowerCase().includes(q));
  const pages = Math.max(1, Math.ceil(found.length / PER_PAGE));
  if (!Number.isInteger(page) || page < 1 || page > pages) {
    return send(404, { error: 'No such page' });
  }
  const items = found.slice((page - 1) * PER_PAGE, page * PER_PAGE);
  const query = q ? `q=${encodeURIComponent(q)}&` : '';
  const next = `</api/products?${query}page=${page + 1}>; rel="next"`;
  return send(200, { page, pages, items }, page < pages ? { link: next } : {});
}
```

CHALLENGE Make the limiter return the count instead of a yes or no, then set both headers before the 429 check; `setHeader` values merge with the ones `writeHead` adds later. Six quick calls show remaining `4 3 2 1 0`, then a 429 with `0`.

lab.js (changes)

```
function callsThisSecond(address) {         // was tooMany(): now returns the count
  const second = Math.floor(Date.now() / 1000);
  const entry = hits.get(address);
  if (entry?.second === second) entry.count += 1;
  else hits.set(address, { second, count: 1 });
  return hits.get(address).count;
}
// in the handler, in place of the old 429 check:
if (path.startsWith('/api/')) {
  const calls = callsThisSecond(req.socket.remoteAddress);
  res.setHeader('x-ratelimit-limit', 5);
  res.setHeader('x-ratelimit-remaining', Math.max(0, 5 - calls));
  if (calls > 5) {
    return send(429, { error: 'Too many requests' }, { 'retry-after': '1' });
  }
}
```

TERMINAL

```
$ for i in 1 2 3 4 5 6; do curl -s -o /dev/null \
  -w '%{http_code} %header{x-ratelimit-remaining}\n' localhost:3000/api/products/1; done
200 4
... 200 3, 200 2, 200 1, 200 0, then:
429 0
```

EXERCISE ANSWERS, CONTINUED

12 Calling REST and GraphQL APIs

WARM-UP `getJson` throws an `HttpError` for any bad status, so catch it and check which one:

boot-price.js — LAB

```
import { getJson, HttpError } from './helpers.js';
const base = 'http://localhost:3000/api/products';
const boot = await getJson(`${base}/3`);
console.log(`${boot.name} costs $${boot.price}`);   // Hiking Boot costs $149.99
try {
  await getJson(`${base}/99`);
} catch (error) {
  if (error instanceof HttpError && error.status === 404) {
    console.log('There is no product 99.');
  } else {
    throw error;                       // anything else is a real problem
  }
}
```

PRACTICE Send the token, and treat a 401 as a setup problem: a message on stderr and a failing exit code. With `LAB_TOKEN=lab-token`, `node --env-file=.env whoami.js` prints `200 { user: 'student', plan: 'free' }`; with a wrong token it prints the hint, and `echo $?` shows `1`.

whoami.js — LAB

```
const res = await fetch('http://localhost:3000/api/me', {
  headers: { authorization: `Bearer ${process.env.LAB_TOKEN}` },
});
if (res.status === 401) {
  console.error('Check LAB_TOKEN in .env');
  process.exitCode = 1;
} else {
  console.log(res.status, await res.json());
}
```

CHALLENGE Count inside the loop; `paginate` fetches each page only when the loop needs it, and the lab gives `{ inStock: 8, soldOut: 2 }`. GitHub answers with an array per page and a `Link` header, so the same `paginate` works there; `break` after ten stops it before a third request. The tags, newest first, change every few weeks. Without a token GitHub allows 60 requests an hour, and this uses two.

stock-count.js

```
import { paginate } from './helpers.js';
let inStock = 0;
let soldOut = 0;
const url = 'http://localhost:3000/api/products';
for await (const p of paginate(url)) {
  if (p.stock > 0) inStock += 1;
  else soldOut += 1;
}
console.log({ inStock, soldOut });
```

tags.js

```
import { paginate } from './helpers.js';
const url = 'https://api.github.com/repos/'
  + 'nodejs/node/releases?per_page=5';
const headers =
  { 'user-agent': 'js-course-bot/1.0' };
const releases = paginate(url, { headers });
const tags = [];
for await (const r of releases) {
  tags.push(r.tag_name);
  if (tags.length === 10) break;   // two pages
}
console.log(tags.join(' '));
```

EXERCISE ANSWERS, CONTINUED

13 Scraping Web Pages with Cheerio

WARM-UP Keep the lines that start with `Disallow:` and cut off the label. This shortcut ignores groups, which works here because the lab's file has only the `User-agent: *` group; for real sites, use `robots-parser` (13.2).

disallowed.js — LAB

```
const res = await fetch('http://localhost:3000/robots.txt');
const lines = (await res.text()).split('\n');
const rules = lines.filter((l) => l.startsWith('Disallow:'));
console.log(rules.map((l) => l.slice(9).trim()));   // [ '/admin' ]
```

PRACTICE After the list pages, visit each product page at the same pace and read its JSON-LD. Add `import * as cheerio from 'cheerio';` to `crawl.js`, then:

crawl.js (add at the end)

```
const details = [];
for (const { id, href } of rows) {
  await sleep(1000);                                  // still one request a second
  const res = await fetch(new URL(href, 'http://localhost:3000'), { headers });
  const $ = cheerio.load(await res.text());
  const ld = JSON.parse($('script[type="application/ld+json"]').text());
  const availability = ld.offers.availability.split('/').at(-1);   // 'InStock'
  const price = Number(ld.offers.price);
  details.push({ id, name: ld.name, price, availability });
}
const soldOut = details.filter((d) => d.availability === 'OutOfStock');
console.log('Sold out:', soldOut.map((d) => d.name));   // [ 'Road Racer', 'Head Lamp' ]
```

CHALLENGE Store `details` with 13.5's upsert, in a new file, since `shop.db` already has a three-column `products` table. Import `Database` from `node:sqlite` and `toCsv` from `helpers.js`. Every run prints `10 rows in stock.db; 4 under $50 in cheap.csv`: the upsert updates the same ten rows, and the CSV lists Wool Socks, Head Lamp, Water Bottle and Sun Hat. 16.2's `store.js` is the same code in a function.

crawl.js (add at the end)

```
const db = new Database('stock.db');
db.exec(`CREATE TABLE IF NOT EXISTS products (id INTEGER PRIMARY KEY,
  name TEXT, price REAL, availability TEXT)`);
const upsert = db.prepare(`INSERT INTO products VALUES (?, ?, ?, ?)
  ON CONFLICT(id) DO UPDATE SET name = excluded.name,
  price = excluded.price, availability = excluded.availability`);
for (const d of details) upsert.run(d.id, d.name, d.price, d.availability);
const { n } = db.prepare('SELECT COUNT(*) AS n FROM products').get();
const cheap = db.prepare('SELECT * FROM products WHERE price < 50').all();
await writeFile('cheap.csv', toCsv(cheap));
console.log(`${n} rows in stock.db; ${cheap.length} under $50 in cheap.csv`);
db.close();
```

EXERCISE ANSWERS, CONTINUED

14 Headless Browsers with Playwright

WARM-UP All three answers go inside the `launch`, `try` and `finally` of `first-browser.js` (14.2). Wait for the first item, so the screenshot shows the list JavaScript built:

shot.js — LAB

```
const page = await browser.newPage();
await page.goto('http://localhost:3000/app');
await page.locator('#list li').first().waitFor();   // JavaScript filled the list
await page.screenshot({ path: 'app.png', fullPage: true });
```

PRACTICE Each context is its own identity. `page.goto` returns the response, so read the lab's JSON straight from it. It prints `js-course-bot/1.0 (A)`, then `(B)`.

agents.js — LAB

```
for (const userAgent of ['js-course-bot/1.0 (A)', 'js-course-bot/1.0 (B)']) {
  const context = await browser.newContext({ userAgent });   // its own identity
  const page = await context.newPage();
  const res = await page.goto('http://localhost:3000/api/echo');
  console.log((await res.json()).agent);                     // what the lab saw
  await context.close();
}
```

CHALLENGE Send every request through a check and abort the heavy types, with a fresh page per run so the timings stay apart. One run printed 119 ms, then 95 ms blocked. Yours will differ: example.com has almost nothing to block, so try a heavier page you're allowed to load.

block.js

```
const HEAVY = ['image', 'font', 'stylesheet'];
async function timeLoad(url, block) {
  const page = await browser.newPage();
  if (block) {
    await page.route('**/*', (route) => {
      const heavy = HEAVY.includes(route.request().resourceType());
      return heavy ? route.abort() : route.continue();
    });
  }
  const t0 = performance.now();
  await page.goto(url);
  const ms = Math.round(performance.now() - t0);
  await page.close();
  return ms;
}
const url = 'https://example.com';
console.log('everything:', await timeLoad(url, false), 'ms');
console.log('blocked:   ', await timeLoad(url, true), 'ms');
```

EXERCISE ANSWERS, CONTINUED

15 Proxies, Concurrency and Scheduling

WARM-UP Split on the colons and encode the password, then compare with the Proxy Format tool. A password holding a colon needs Chapter 10's `rest.join(':')`.

proxy-url.js

```
const line = 'gate.example.net:7000:student:p@ss';
const [host, port, user, pass] = line.split(':');
console.log(`http://${user}:${encodeURIComponent(pass)}@${host}:${port}`);
// http://student:p%40ss@gate.example.net:7000
```

PRACTICE Start a practice proxy with `npx proxy -p 8001` and put `PROXY_URL=http://127.0.0.1:8001` in `.env`. `ip.js` then prints your own public IP, because the practice proxy runs on your computer and leaves through your own connection. Stop it with `Ctrl`+`C` and run again: `fetch failed · connect ECONNREFUSED 127.0.0.1:8001`, since nothing listens there now. Stop practice proxies when you're done: while one runs, other devices on your network can use it.

CHALLENGE Start a second practice proxy on port 8002, and count requests in a `Map` keyed by the dispatcher `rotate()` hands out; `helpers.js` calls undici's `install()`, so the global `fetch` accepts it. For the schedule, write `30 7 * * 1-5` with your own IANA time zone, such as `timezone: 'America/New_York'` or `'Asia/Kolkata'`, beside the `cron` line of 16.3's workflow.

pool-count.js — LAB

```
import { mapLimit, pace, ProxyPool } from './helpers.js';
const pool = new ProxyPool(['http://127.0.0.1:8001', 'http://127.0.0.1:8002']);
const perProxy = new Map();                  // dispatcher -> requests sent
const wait = pace(250);
const ids = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const statuses = await mapLimit(ids, 2, async (id) => {
  await wait();
  const dispatcher = pool.rotate();
  perProxy.set(dispatcher, (perProxy.get(dispatcher) ?? 0) + 1);
  const url = `http://localhost:3000/api/products/${id}`;
  return (await fetch(url, { dispatcher })).status;
});
console.log(statuses.join(' '));             // 200 ten times
console.log('requests per proxy:', [...perProxy.values()]);   // [ 5, 5 ]
```

16 Three Projects to Build and Ship

extension challenges

WARM-UP `repo-pulse`: import `toCsv`, add two options, sort with "not found" rows last, and print a third way; the rest stays as in 16.1. `--sort stars --csv` prints the header, then a row per repo. The CSV goes to stdout and the rate-limit line to stderr, so `> repos.csv` saves a clean file.

repo-pulse.js (changes)

```
const { values, positionals: repos } = parseArgs({ allowPositionals: true, options: {
  json: { type: 'boolean' }, csv: { type: 'boolean' }, sort: { type: 'string' } } });
// ...then, after the rows are built:
const n = (row) => (typeof row[values.sort] === 'number' ? row[values.sort] : -1);
if (values.sort) rows.sort((a, b) => n(b) - n(a));   // biggest first, "not found" last
if (values.csv) process.stdout.write(toCsv(rows));
else if (values.json) console.log(JSON.stringify(rows, null, 2));
else console.table(rows);
```

EXERCISE ANSWERS, CONTINUED

16 Three Projects to Build and Ship

continued

PRACTICE `shop-crawler`: one `runs` row per crawl, and a kill switch in `get()`. `fetchWithRetry` already retries a 429 three times, so one that reaches `get()` means "slow down". Throwing once wouldn't stop the crawl, because each product's `try` / `catch` swallows the error; a flag that makes every later `get()` throw does. At the end, call `saveRun('data/shop.db', { started, pages, products: good.length, seconds: Number(secs), stopped })`. Against a test server that answered every product page with 429, the third one stopped the crawl.

store.js (add)

```
export function saveRun(file, run) {
  const db = new Database(file);
  db.exec(`CREATE TABLE IF NOT EXISTS runs
    (started TEXT, pages INTEGER,
    products INTEGER, seconds REAL,
    stopped INTEGER)`);
  db.prepare(`INSERT INTO runs
    VALUES (?, ?, ?, ?, ?)`)
    .run(run.started, run.pages, run.products,
      run.seconds, run.stopped ? 1 : 0);
  db.close();
}
```

shop-crawler.js (changes)

```
import { saveProducts, saveRun }
  from './store.js';
const started = new Date().toISOString();
let in429 = 0;          // 429s in a row
let stopped = false;    // the kill switch
// the first line of get():
if (stopped) throw new Error(`stopped: ${url}`);
// right after fetchWithRetry answers:
in429 = res.status === 429 ? in429 + 1 : 0;
if (in429 >= 3) stopped = true;
```

CHALLENGE `site-watch`: compare each page's problems with yesterday's, never the timings, which change every run. Make `check()` return `{ url, problems: problems.join(', ') }`, with `problems: `unreachable (${error.name})`` in its `catch`, then replace everything from `const lines` to the end of the file. The first run reports every page, the baseline; the next prints `No changes since yesterday` until a page changes. The digest is the alert now, so the run no longer fails while known problems remain.

site-watch.js (changes)

```
import { readFile, writeFile } from 'node:fs/promises';
// ...check() as above, then:
const results = await mapLimit(process.env.WATCH_URLS.split(','), 3, check);
const today = Object.fromEntries(results.map((r) => [r.url, r.problems]));
const yesterday = await readFile('yesterday.json', 'utf8')
  .then(JSON.parse, () => ({}));            // first run: nothing to compare
const changed = results.filter((r) => yesterday[r.url] !== r.problems);
await writeFile('yesterday.json', JSON.stringify(today, null, 2));
if (changed.length === 0) {
  console.log('No changes since yesterday');
} else {
  const lines = changed.map((r) =>
    `${r.problems ? 'FIX' : 'OK '} ${r.url} ${r.problems || 'all good'}`);
  const n = `${changed.length} of ${results.length}`;
  const text = [`${n} pages changed since yesterday`, ...lines].join('\n');
  console.log(text);
  // ...post text to WEBHOOK_URL exactly as before
}
```

On GitHub Actions each run starts from a fresh checkout, so `yesterday.json` vanishes unless you keep it: commit it back from the workflow, or save it with `actions/cache@v6`, using a key that's new on every run and a `restore-keys` prefix that brings back the latest copy. The stateless vs stateful scraping post covers when state like this is worth keeping.

KEEP IT OPEN WHILE YOU CODE

JavaScript Syntax Cheat Sheet

{ } THE BASICS

Values and variables Chapter 2

```
const price = 89;          // can't be reassigned
let qty = 1; qty += 1;     // can change
typeof 'hi'                // 'string'
typeof null                // 'object' (a quirk)
`${qty} items`             // template literal
// falsy: false 0 -0 0n '' null undefined NaN
```

Operators Chapter 2

```
a === b   a !== b          // never == or !=
a && b    a || b    !a     // and, or, not
stock ?? 0                 // only null, undefined
name || 'friend'           // any falsy value
user?.address?.city        // undefined, no crash
stock > 0 ? 'In' : 'Out'   // pick one of two
7 % 3   2 ** 10   i++      // 1, 1024, add 1
```

Control flow Chapter 3

```
if (a) {...} else if (b) {...} else {...}
switch (day) { case 'Sat': ...; break; }
for (let i = 0; i < 3; i++) {...}
for (const item of list) {...}
while (more) {...}         // break, continue
```

Functions Chapter 3

```
function add(a, b = 1, ...rest) {
  return a + b;            // default, rest
}
const double = (n) => n * 2;
const toItem = (name) => ({ name });  // object
list.forEach((x) => console.log(x));
```

Classes Chapter 6

```
class Cart extends Base {
  #items = [];             // private field
  constructor() { super(); }
  get size() { return this.#items.length; }
  static from(json) { return new Cart(); }
}
cart instanceof Cart       // true
```

DATA, ASYNC AND MODULES

Array methods Chapter 4

```
// return something new, the original stays
map  filter  find  findLast  some  every
slice  reduce  includes  toSorted  toReversed
list.reduce((sum, x) => sum + x, 0)
list.toSorted((a, b) => a - b)   // numbers
// change the array itself
push  pop  shift  unshift  splice  sort  reverse
```

Objects, destructuring and spread Chapter 4

```
const { name, price: cost = 0 } = item;
const [first, ...rest] = list;
const copy = { ...item, price: 79 }; // shallow
const deep = structuredClone(item);
Object.keys(o)   Object.entries(o)
Object.groupBy(list, (p) => p.type)
new Map([[1, 'a']])   [...new Set(list)]
```

Async Chapter 7

```
const data = await getJson(url);
await Promise.all(urls.map(get))  // fail fast
await Promise.allSettled(jobs)    // every result
fetch(url, { signal: AbortSignal.timeout(5_000) })
import { setTimeout as sleep }
  from 'node:timers/promises';
await sleep(1000);                // wait 1 s
```

Errors Chapter 5

```
try { risky(); } catch (error) { ... }
finally { cleanUp(); }            // always runs
throw new RangeError(`Bad: ${text}`);
throw new Error('Row 7', { cause: error });
error instanceof HttpError        // which kind?
```

Modules Chapter 9

```
export function usd(n) { ... }
export default withTax;
import withTax, { usd } from './money.js';
import * as money from './money.js';
import data from './d.json' with { type: 'json' };
import.meta.dirname         // this file's folder
```

PATTERNS AND PARTS

Regular Expressions Cheat Sheet

Chapter 8 teaches the basics. Test a pattern in the REPL with `/pattern/flags.test('text')` before you trust it on real data.

Token	Matches
CHARACTERS AND CLASSES	
`.`	Any character but a line break (any at all with `s`)
`\d \D`	A digit; anything else
`\w \W`	A letter, digit or `_`; anything else
`\s \S`	Whitespace; anything else
`[abc]`	One of a, b or c
`[^abc]`	Anything but a, b or c
`[a-z0-9]`	One character from a range
`\. \$ \(`	The character itself, escaped
HOW MANY	
`* + ?`	0 or more; 1 or more; 0 or 1
`{3} {2,}`	Exactly 3; 2 or more
`{2,4}`	2 to 4
`*? +?`	As few as possible (lazy)
WHERE	
`^ $`	Start and end (of each line with `m`)
`\b`	A word boundary

Token	Matches
GROUPS AND CHOICES	
`(abc)`	A group you can read back: `match[1]`, `$1`
`(?<id>abc)`	A named group: `match.groups.id`, `$<id>`
`(?:abc)`	A group that doesn't capture
`a|b`	a or b
`(?=x) (?!x)`	Followed by x; not followed by x
`(?⩽x) (?<!x)`	After x; not after x
FLAGS, AFTER THE CLOSING SLASH	
`g i m`	Every match; ignore case; `^` and `$` per line
`s u v`	`.` matches line breaks; Unicode; Unicode sets
`d y`	Match positions in `indices`; sticky
METHODS	
`re.test(s)`	`true` or `false`
`s.match(re)`	The first match, or every match with `g`
`s.matchAll(re)`	Each match with its groups (needs `g`)
`s.replace(re, x)`	Swap matches for `x`, a string or a function
`s.split(re)`	Split on a pattern, such as `/\s+/`
`RegExp.escape(s)`	Makes text safe inside a pattern (ES2025)

Patterns From This Book

Pattern	What it finds	Where
`/^\/api\/products\/(\d+)$/`	The id at the end of a Lab Shop API path	11.4
`/<([^>]+)>;\s*rel="next"/`	The next page's URL in a `Link` header	12.3
`/[^\d.]/g`	Everything but digits and dots, to clean `$89.00` into `89.00`	13.3
`/[",\r\n]/`	The characters that make a CSV cell need quotes	10.2

COMMANDS AND PATTERNS

Node.js Toolbox Cheat Sheet

RUN, INSTALL, CONFIGURE

Run Node Chapters 1, 5, 9, 10

```
node app.js                  // run a file
node app.ts                  // types stripped
node --watch app.js          // re-run on save
node --env-file=.env app.js // load settings
node --test                  // every *.test.js
node --run start             // a package script
node -p "process.version"    // print one value
node --inspect-brk app.js    // debug in Chrome
```

Proxies and permissions Chapters 15, 16

```
node --use-env-proxy app.js // fetch via HTTPS_PROXY
node --permission --allow-fs-read=. \
  --allow-fs-write=./data --allow-net app.js
```

npm Chapter 9

```
npm init -y; npm pkg set type=module
npm install cheerio          // a dependency
npm install -D typescript    // a dev tool
npm uninstall cheerio
npm ci                       // exactly the lockfile
npm audit                    // known problems
npx tsc                      // a package's command
npm link                     // "bin" as a command
```

Files and the process Chapter 10

```
import { readFile, writeFile }
  from 'node:fs/promises';
join(import.meta.dirname, 'data.json')
process.argv.slice(2)        // your arguments
process.env.TOKEN            // a setting
process.exitCode = 1         // fail, but finish
console.error('oops')        // to stderr
```

tsconfig.json Chapter 9

```
{ "compilerOptions": {
    "module": "nodenext", "noEmit": true,
    "allowImportingTsExtensions": true,
    "erasableSyntaxOnly": true,
    "verbatimModuleSyntax": true,
    "types": ["node"] } }
```

TALK TO THE WEB

fetch Chapter 12

```
const res = await fetch(url, {
  method: 'POST',
  headers: { 'content-type': 'application/json',
    'user-agent': 'my-bot/1.0' },
  body: JSON.stringify(data),
  signal: AbortSignal.timeout(10_000),
});
if (!res.ok) throw new HttpError(res.status, url);
const body = await res.json();  // or res.text()
```

Cheerio and Playwright Chapters 13, 14

```
const $ = cheerio.load(html);
$('li.product a').text()   $('a').attr('href')
$('li').map((i, el) => $(el).text()).get()
const browser = await chromium.launch();
const page = await browser.newPage();
await page.goto(url);
await page.locator('#list li').allTextContents()
await browser.close();     // in finally
```

Proxy URLs Chapter 15

```
http://USER:PASS@gate.example.net:7000
socks5://USER:PASS@gate.example.net:1080
encodeURIComponent('p@ss')    // 'p%40ss'
import { fetch, ProxyAgent } from 'undici';
fetch(url, { dispatcher: new ProxyAgent(u) })
browser.newContext({ proxy: { server,
  username, password } })
```

cron Chapter 15

```
// minute hour day-of-month month day-of-week
30 7 * * 1-5     // weekdays at 07:30
*/30 * * * *     // every 30 minutes
0 */6 * * *      // every 6 hours
0 9 1 * *        // 09:00 on the 1st
// GitHub Actions: UTC unless you set timezone
```

REFERENCE

HTTP Status Codes and Headers

Code	Name	What it means	Your script
2XX · SUCCESS			
200	**OK**	It worked; the body has the data	Use it
201	**Created**	A POST made something new	Use it
204	**No Content**	It worked, and there's no body	Skip `res.json()`
3XX · LOOK ELSEWHERE			
301, 308	**Moved Permanently**	The resource has a new URL for good	`fetch` follows it
302, 307	**Found, Temporary Redirect**	Look elsewhere this time	`fetch` follows it
304	**Not Modified**	Your cached copy is still current	Use the cache
4XX · THE REQUEST'S FAULT			
400	**Bad Request**	The request is malformed	Fix it; don't retry
401	**Unauthorized**	Credentials missing or wrong	Fix the token
403	**Forbidden**	Not allowed (on GitHub, maybe a rate limit)	Check the headers
404	**Not Found**	No such resource	Don't retry
407	**Proxy Auth Required**	The proxy rejected your login	Fix the proxy URL
429	**Too Many Requests**	You're over a rate limit	Wait for `Retry-After`
5XX · THE SERVER'S FAULT			
500	**Internal Server Error**	The server broke	Retry with backoff
502	**Bad Gateway**	A gateway got a bad answer upstream	Retry with backoff
503	**Service Unavailable**	Overloaded or down, often with `Retry-After`	Retry with backoff
504	**Gateway Timeout**	A gateway gave up waiting upstream	Retry with backoff

Headers You'll Meet

Header	What it carries	Sent with
`content-type`	What the body is, such as `application/json` or `text/html`	Both
`user-agent`	Who's asking; name your bot, as in `js-course-bot/1.0`	Request
`authorization`	`Bearer TOKEN` or `Basic` and base64 `user:pass`	Request
`link`	Where the next page is: `<...>; rel="next"`	Response
`retry-after`	Seconds to wait (or a date) after a 429 or 503	Response
`x-ratelimit-remaining`	Calls left in the current window, as GitHub sends it	Response

YOUR PRACTICE LAB

Lab Shop Routes and Helpers

Every Lab Shop Route

Route	What it does	Added	Used in
`GET /api/products?page=N`	4 a page, 3 pages, a `Link` to the next; a bad page: 404	Ch 11	12, 15, 16
`GET /api/products/:id`	One product, or 404 `{ "error": "Not found" }`	Ch 11	12, 15
`GET /api/me`	200 with `Bearer lab-token`, otherwise 401	Ch 11	12
`ANY /api/echo`	Echoes your method, body and three key headers	Ch 11	12, 14, 16
`GET /api/slow`	Answers `{ "ok": true }` after 3 seconds	Ch 11	12
`GET /api/flaky`	503 with `Retry-After: 1` twice, then 200, repeating	Ch 11	12
Any `/api/` route	The 6th call in one second gets 429, `Retry-After: 1`	Ch 11	12, 15
`GET /`	"Lab Shop is running"	Ch 11	11
Anything else	404 `{ "error": "No route for /path" }`	Ch 11	11
`GET /robots.txt`	`Disallow: /admin` and `Crawl-delay: 1` for everyone	Ch 13	13, 16
`GET /products?page=N`	HTML list: `li.product` links, prices, `a.next`	Ch 13	13, 16
`GET /products/:id`	HTML: `h1`, `p.price`, `p.stock` and JSON-LD	Ch 13	13, 16
`GET /app`	An empty `ul#list` that browser JavaScript fills	Ch 14	14

Where Each Helper Is Taught

The reusable pieces in your `helpers.js` (and two neighbors), with the section that builds each one.

Helper	What it does	Built in	Used again in
`usd(n)`	Formats a price as `$89.00`	3.4	Ch 9, 11
`sleep(ms)`	The promise `setTimeout` from `node:timers/promises`	7.4	Ch 8, 12, 13, 15
`HttpError`	An error that carries `status` and `url`	6.4	Ch 12, 13, 16
`toCsv(rows)`	Rows to CSV, quoted the RFC 4180 way	10.2	Ch 13, 16
`getJson(url)`	`fetch`, check `res.ok`, parse the JSON	12.1	Ch 12
`paginate(url)`	Follows `rel="next"` links, one item at a time	12.3	Ch 12
`fetchWithRetry()`	Timeouts, capped retries, backoff, `Retry-After`	12.4	Ch 16
`parseProducts(html)`	Reads the lab's list pages with Cheerio (`parse.js`)	13.3	Ch 13, 16
`ProxyPool`	`rotate()` and `sticky(key)` over undici agents	15.3	Ch 15, 16
`mapLimit()`	At most `limit` tasks in flight, results in order	15.4	Ch 15, 16
`pace(ms)`	At most one start every `ms` milliseconds	15.4	Ch 15, 16
`saveProducts()`	SQLite upsert, then a row count (`store.js`)	16.2	Ch 16

PRINT AND TICK

Checklists

Five lists from the chapters, in one place. Print this page, or keep it beside your editor.

Before you scrape

13.1

- [ ] Look for an official API or export first
- [ ] Read robots.txt and follow your group's rules
- [ ] Read the terms, especially on automated access
- [ ] Name your bot and a contact in the User-Agent
- [ ] Go slow: one request at a time, with a delay
- [ ] Collect only what you need, and no personal data without a lawful reason

Script code review

Chapters 12 to 16

- [ ] A timeout on every request
- [ ] `res.ok` checked before the body is used
- [ ] Retries capped, with backoff and jitter
- [ ] Secrets in env, never in the code
- [ ] A User-Agent that names your bot
- [ ] Concurrency and rate both capped
- [ ] Errors to stderr and a non-zero exit code
- [ ] Parsers tested against saved pages

Ship it responsibly

16.4

- [ ] Secrets in env or GitHub secrets, never code
- [ ] Logs as JSON lines; errors to stderr
- [ ] A timeout per request, a page limit per run
- [ ] Only the data you need, the rest deleted
- [ ] `.env` listed in `.gitignore`
- [ ] A non-zero exit code on failure
- [ ] A kill switch after repeated 403s or 429s
- [ ] robots.txt, terms and rate limits checked

Before you install a package

9.4

- [ ] Is it built in? Node 26 has `fetch`, tests, SQLite and argument parsing
- [ ] Is the name exactly right? Copy it from the docs
- [ ] Is it maintained? Recent releases, answered issues
- [ ] Is it widely used? Check weekly downloads
- [ ] Does the license fit? MIT and Apache-2.0 suit most
- [ ] Unsure? `--ignore-scripts`, then `npm audit`

When you're stuck

5.3

- [ ] Read the whole error, top to bottom
- [ ] Find the first frame in your own file
- [ ] Log the values just before the bad line
- [ ] Halve it: comment out half, does it still break?
- [ ] Shrink it to the smallest example that fails
- [ ] Explain it out loud, line by line

WORDS YOU'LL MEET

Glossary

Every term the book defines, in one list. Linked terms have a longer entry in the ProxyHorizon Glossary.

407 Proxy Authentication Required. What a proxy returns when your username or password is missing or wrong (proxy authentication).

AbortSignal. An object that tells an operation to stop; `AbortSignal.timeout(ms)` fires by itself after `ms`.

Accumulator. The running value `reduce` carries through an array, such as a sum or a count.

API. Application programming interface: URLs a program calls to read or change data, usually answering in JSON.

Array. An ordered list of values in square brackets, indexed from 0.

Arrow function. A shorter function syntax, `(x) => x * 2`, that also takes `this` from where it's written.

Assertion. A check in a test, such as `assert.equal(actual, expected)`, that fails loudly when the result is wrong.

`async` and `await`. `async` makes a function return a promise; `await` pauses that function until a promise settles.

Asynchronous code. Code that starts slow work, such as a network call, moves on, and handles the result later.

Bearer token. A secret sent as `authorization: Bearer TOKEN` to prove who you are to an API.

Browser context. In Playwright, an isolated session inside one browser, with its own cookies and, optionally, its own proxy.

Call stack. Where JavaScript keeps the function calls it's running right now, newest on top, one at a time.

Callback. A function you pass to another function so it can call it later or for each item.

Cheerio. A Node.js library that parses HTML into a tree you query with CSS selectors; it doesn't run scripts.

Class. A blueprint for objects that share data and behavior, built with `new` and set up by a `constructor`.

Closure. A function plus the variables it could see when it was created, kept alive after the outer function ends.

CommonJS. Node's older module format, using `require()` and `module.exports`; still common in older code.

Concurrency. How many tasks run at once. Different from rate, which is how many start per second.

Crawl-delay. A non-standard robots.txt line asking crawlers to wait that many seconds between requests.

Cron. A five-field schedule (minute, hour, day of month, month, day of week), such as `30 7 * * 1-5` for weekdays at 07:30.

CSS selector. A pattern such as `li.product a` that matches elements by tag, class, id, attribute or nesting.

curl. A command-line HTTP client on macOS, Windows 10 and later, and most Linux systems.

Dependency. A package your project needs, listed in `package.json`; dev-only tools go in `devDependencies`.

Destructuring. Unpacking values from an object or array into variables in one line, with optional renames and defaults.

DOM. The Document Object Model: the tree of elements a browser builds from HTML, which scripts read and change.

Endpoint. One URL of an API that answers a particular kind of request, such as `/api/products/3`.

Engine (V8). The part of a runtime that reads and executes JavaScript. Chrome, Edge and Node.js all use V8.

Environment variable. A named setting a program reads from `process.env`, often loaded from a `.env` file with `--env-file`.

ES module (ESM). JavaScript's standard module format, with `import` and `export`; this course uses it everywhere.

Event loop. The loop in Node.js that finishes the current code, then runs every microtask, then the next task.

Exception. An error raised with `throw` that stops the code until a `try` / `catch` handles it.

Exit code. The number a process ends with: 0 for success, anything else for failure; set it with `process.exitCode`.

Exponential backoff. Waiting longer after each failed attempt (0.5 s, 1 s, 2 s…), usually capped and with jitter added.

Fixture. A saved copy of input data, such as an HTML page, that tests read instead of the live network.

Function. A named, reusable piece of code that takes inputs and can hand back a result with `return`.

Generator. A `function*` that yields values one at a time and pauses between them, so sequences can be lazy or endless.

Getter. A class method that looks like a property when you read it, such as `product.inStock`.

GitHub Actions. GitHub's automation service; a `schedule` trigger runs a workflow on a cron line, in UTC by default.

GraphQL. An API style with one endpoint: you POST a query naming the fields you want; errors can come with a 200.

Guard clause. An early `return` at the top of a function that handles bad input before the main logic.

GLOSSARY, CONTINUED

Headless browser. A real browser running without a window, driven by code.

HTTP header. A `name: value` line sent with a request or response, such as `content-type` or `user-agent`.

HTTP method. The verb of a request: GET reads, POST creates or sends, PUT and PATCH replace or change, DELETE removes.

Iterable. Anything `for...of` can loop over, such as arrays, strings, Maps and Sets.

Jitter. A small random amount added to a retry wait, so many clients don't retry at the same moment.

JSON. JavaScript Object Notation: text shaped like objects and arrays, read with `JSON.parse`.

JSON-LD. Structured data in a `script` tag, written for search engines and often cleaner than the HTML.

Link header. A response header, `Link: <...>; rel="next"`, that says where the next page is.

localhost. The name a computer uses for itself, so `localhost:3000` never leaves your machine.

Locator. Playwright's way to find elements on a page; actions on locators wait for the element by themselves.

Lockfile. `package-lock.json`: the exact version of every package, so `npm ci` rebuilds the same tree.

LTS. Long-term support: a Node.js line that gets fixes for years (Node.js 26 until April 2029).

`Map` and `Set`. A Map holds key-value pairs with any type of key; a Set holds each value once and can compare sets.

Method. A function stored as a property of an object, such as `products.map`.

Microtask. A queued callback, such as a promise's `then`, that runs right after the current code, before any timer.

Module. A file with its own scope that shares only what it exports; other files import it.

Mutate. To change an array or object in place, as `push`, `splice` and `sort` do.

Named group. A regular-expression group written `(?<name>...)`, whose match you read by name from `match.groups`.

NDJSON. Newline-delimited JSON: one JSON object per line, safe to append to and easy to stream.

Node.js. The runtime that runs JavaScript outside the browser. This book uses Node.js 26 LTS.

npm. Node's package registry, and the command that installs packages into `node_modules`.

Nullish coalescing (`??`). A fallback only for `null` or `undefined`; `||` also replaces `0` and `''`.

Object. A record of named values, called properties, in curly braces.

Operator. A symbol that combines values into a new one, such as `+`, `===`, `&&` or `??`.

Optional chaining (`?.`). Returns `undefined` instead of crashing when part of a property path is missing.

Package. Reusable code published to the npm registry and installed with `npm install`.

Pagination. Splitting a long list into pages, by number, offset, cursor or a `Link` header.

Parameter and argument. A parameter is an input's name inside a function; an argument is the value you pass in.

Permission model. Node's `--permission` flag: no file or network access unless you grant it, as with `--allow-net`.

Placeholder (SQL). A `?` in a prepared statement that the database fills in safely, which prevents SQL injection.

Playwright. A library that drives Chromium, Firefox and WebKit from code, for pages that need JavaScript.

Port. A numbered door on a computer that lets many servers share one machine; the Lab Shop uses 3000, HTTPS sites 443.

Primitive. A simple value type: string, number, bigint, boolean, undefined, null or symbol.

Private field (`#`). A class property marked with `#` that no code outside the class can read or change.

Promise. An object for a result that isn't ready yet; it settles once, with a value or an error.

Promise combinator. Turns many promises into one: `all`, `allSettled`, `race` and `any` treat failure differently.

Proxy. A server that forwards your requests, so the website sees the proxy's IP address instead of yours.

`ProxyAgent` (dispatcher). undici's object for one proxy, passed to `fetch` as its `dispatcher` option.

Pure function. A function whose output depends only on its inputs and that changes nothing outside itself.

Query string. The part of a URL after `?`, such as `?page=2`; `URLSearchParams` reads and builds it.

Queue. A list of work waiting its turn, taken from the front; `mapLimit`'s workers share one.

Rate limit. A cap on how many requests a client may make in a time window; going over usually returns 429.

GLOSSARY, CONTINUED

Regular expression. A pattern for text, such as `/\d+/` for a run of digits, used to test, find, extract or replace.

REPL. Read, evaluate, print, loop: the interactive prompt you get by typing `node` with no file.

REST. An API style with many URLs (endpoints), each returning a fixed shape of data for a method such as GET or POST.

Retry-After. A header on 429 and 503 responses: how many seconds to wait, or until when, before trying again.

robots.txt. A file at a site's root (RFC 9309) saying which paths each crawler may visit: a request, not a lock.

Rotating proxy. A new exit IP for each request, from a provider's gateway or your own round-robin pool.

Route. A method and path a server answers, such as `GET /api/products/3`, with the code that builds the response.

Runtime. The program that runs your JavaScript and hands it tools: the browser is one, Node.js is another.

Scope. The part of the code that can see a name: module, function or block.

Secret (CI). An encrypted value, such as a token, stored in GitHub and passed to a workflow as an environment variable.

Semantic versioning (semver). Versions as major.minor.patch; `^` allows minor and patch updates, `~` patches only.

Shebang. The first line `#!/usr/bin/env node`, which tells macOS and Linux to run a script file with Node.

SOCKS5. A lower-level proxy protocol, written `socks5://user:pass@host:port`.

Spread syntax (`...`). Copies the items or properties of an array or object into a new one; the copy is shallow.

SQL injection. When text glued into a SQL string changes the query itself. Placeholders prevent it.

SQLite. A database in a single file, with no server to run; Node 26 includes it as `node:sqlite`.

Stack trace. The chain of calls that led to an error, newest first; start with the top frame in your own file.

Statement. One instruction that does something, usually ended with a semicolon.

Static method. A method called on the class itself, often a factory such as `Product.from(record)`.

Status code. The three-digit result of a request: 2xx success, 3xx elsewhere, 4xx your fault, 5xx the server's.

stdout and stderr. A program's two outputs: normal output (`console.log`) and errors (`console.error`).

Sticky session. Keeping the same proxy IP for a whole session, needed for logins, carts and multi-step flows.

Stream. Data that arrives in chunks, so a program can handle a huge file without loading it all.

`structuredClone`. A built-in function that makes a deep copy, nested arrays and objects included.

Template literal. A backtick string that can span lines and insert values with `${}`.

Temporal dead zone. The lines before a `let` or `const`, where using the name throws a `ReferenceError`.

Terminal. The text window where you type commands such as `node hello.js`; VS Code has one built in.

Test. Code that runs your code and checks the result with assertions; `node --test` runs every `*.test.js` file.

`this`. The object a function was called on, decided by how it's called (arrow functions keep the outer one).

Truthy and falsy. How a value acts in an `if`; only `false`, `0`, `-0`, `0n`, `''`, `null`, `undefined` and `NaN` are falsy.

Type stripping. How Node.js 26 runs `.ts` files: it deletes the types and runs the rest, unchecked.

TypeScript. JavaScript with type annotations, checked by the `tsc` compiler before anything runs.

Upsert. Insert a row, or update it if its key exists, so re-runs never duplicate data.

User-Agent. The request header that names the client, such as `js-course-bot/1.0`. Name your bots honestly.

Value. A piece of data, such as a price, a name or true or false; every value has a type.

Variable. A name for a value, declared with `const` (fixed) or `let` (can point at a new value).

Web scraping. Fetching pages with code and pulling out the data you need.

Webhook. A private URL that turns an HTTP POST into an action, such as a chat message.

Working directory. The folder a command runs from (`process.cwd()`), where relative paths start.

KEEP GOING

Free Tools and Further Reading

Everything below is on ProxyHorizon. The developer tools need no account; the site tools need a free one.

Developer tools, no account

JSON Formatter	Read API replies	cURL Builder	Try a request first
User-Agent Parser	Decode any UA	Random User-Agent	For testing your own site
Proxy Checker	Test a proxy first	Proxy Format Converter	Switch list formats
Proxy Rotator	Rotation scripts	My IP	Your public address
IP Lookup	Where an IP lives	My Fingerprint	What your browser shows
URL Encoder	Escape query values	Base64	Basic auth, data URLs
JWT Decoder	Inside a token	Timestamp Converter	Epoch to dates
UUID Generator	IDs and API keys	Hash Generator	Fingerprint content

Site tools, free account

URL to Markdown	A page as clean text	Robots.txt Tester	Check a path's rules
Schema Extractor	A page's JSON-LD	Webpage Screenshot	Capture any page
Technology Checker	What a site runs on	Meta Tag Checker	Titles, descriptions
Canonical Checker	The preferred URL	Indexability Checker	noindex and friends

Guides worth reading next

- Node.js Scraping With Proxies
- Playwright Scraping Guide
- Rate Limiting in Scraping
- Is Web Scraping Legal?
- Sticky vs Rotating Sessions
- Schedule and Monitor Scrapers

Learn more on ProxyHorizon

APIs explained	Learn hub	Web scraping explained	Learn hub
Proxies explained	Learn hub	The full glossary	Every term
Proxy providers	Tested and rated	More free eBooks	The whole series

JS JAVASCRIPT BEYOND THE BROWSER

From your first line of JavaScript to APIs, scrapers and bots that run while you sleep.

ProxyHorizon

Compare confidently. Implement correctly.

ProxyHorizon is the independent directory and resource hub for proxies, VPNs and antidetect browsers: expert-tested comparisons, free tools and practical guides to choose and use them.

Compare 120+ proxy providers · Free developer and web data tools · In-depth, fact-checked guides



MORE FREE EBOOKS FROM PROXYHORIZON

The Scraping Blueprint	The Antidetect Browser Handbook	Proxies for AI Agents and LLM Data	Lead Generation with Public Web Data	The Ultimate Proxy Handbook
Free PDF	Free PDF	Free PDF	Free PDF	Free PDF

© 2026 PROXYHORIZON · JAVASCRIPT BEYOND THE BROWSER