

EXPERT INSIGHT

Node.js Design Patterns

Level up your Node.js skills and design production-grade applications using proven techniques

Fourth Edition

Forewords by

Colin J. Ihrig

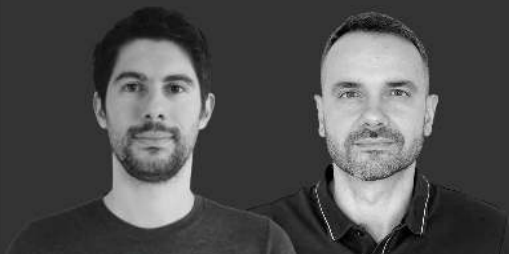
Node.js Contributor

Matteo Collina

Co-Founder & CTO at Platformatic.dev

Chair of the Node.js Technical Steering Committee

Lead Maintainer of Fastify



Luciano Mammino
Mario Casciaro

«packt»

Node.js Design Patterns

Fourth Edition

Level up your Node.js skills and design production-grade applications using proven techniques

Luciano Mammino

Mario Casciaro



Node.js Design Patterns

Fourth Edition

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Ashwin Nair

Relationship Lead: Aaron Lazar

Project Manager: Ruvika Rao

Content Engineer: Nisha Cleetus

Technical Editor: Rohit Singh

Copy Editor: Safis Editing

Indexer: Tejal Soni

Proofreader: Nisha Cleetus

Production Designer: Prashant Ghare

Growth Lead: Priyadarshini Sharma

First published: December 2014

Second edition: July 2016

Third edition: June 2020

Fourth edition: September 2025

Production reference: 1180925

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80323-894-4

www.packtpub.com

Foreword 1

When the first edition of *Node.js Design Patterns* was published, the server-side JavaScript landscape looked very different from today. Node.js itself was still only a few years old, although it was already being deployed at scale by numerous large companies. Developers were more likely to be using callbacks and CommonJS than Promises and ECMAScript modules. In *Node.js Design Patterns*, Mario covered the important topics for building applications with Node.js at the time.

Since that time, a lot has changed in the JavaScript ecosystem. New language features such as `async/await` have fundamentally changed how code is written. JavaScript code is no longer confined to a single thread. Node.js is regularly improving its compatibility with the web platform, as well as adding new functionality of its own. New code is often written in TypeScript, and developers expect to be able to run it without having to transpile it first. Node.js is no longer considered new and shiny, but rather boring and dependable — as any popular, mature software project should be.

Keeping up with the evolution of Node.js can be challenging. As a long-time Node.js contributor and production user, I can attest to that. Fortunately for the community, *Node.js Design Patterns* has continued to prove itself up to the task. Second and third editions of the book have been published along the way, with Luciano joining as a co-author. Now in its fourth edition, I still think this book covers the important topics for building applications with Node.js.

I think this book has something for Node.js developers of all experience levels. For developers who are new to Node.js, this book can be used as a good starting point on their journey. These readers will learn about some of the history, design choices, and architecture of Node.js, before moving on to common patterns for building and scaling applications. Additionally, this edition of the book includes a chapter on testing with the new Node.js test runner. As the original author of Node's test runner, this chapter piqued my interest. Not only is the test runner explored in detail, but the chapter also does a great job of covering generic testing philosophies that can be applied to other technologies.

More experienced Node.js developers can use the material in this book to keep up with the latest developments in the ecosystem, and will find this book to be an overall handy reference. However, this book doesn't just focus on Node.js in isolation. There is also coverage of important tools for successfully running Node.js in production, such as Nginx and Kubernetes, which I think advanced users will find appealing.

No matter where you are on your software journey, I sincerely hope you enjoy this book. As you explore its pages, remember that Node.js has always been about more than just technology. It is also about community and collaboration between countless developers around the world. This book is both a product of, and a tribute to that community.

I am excited to see how you, the reader, will continue to carry the Node.js ecosystem forward!

Colin J. Ihrig

Node.js Contributor

Foreword 2

Performance is not an afterthought in Node.js development; it is a design principle. How can we build applications that are not just functional but blazingly fast? How do we structure code that can handle millions of requests without breaking a sweat? These questions have driven my work for years, and I am thrilled to introduce a book that provides concrete answers: this exceptional guide to Node.js Design Patterns.

The authors have created something remarkable here: a comprehensive resource that not only teaches patterns but also shows you how to think about performance and architecture from day one. When I started writing Fastify, aiming for that astonishing speed of 47k requests per second, I learned that the secret was not in clever tricks but in understanding the fundamental patterns that make Node.js tick. This book captures that essence perfectly.

The journey begins with understanding the Node.js platform itself and its module system. These foundational chapters set the stage for everything that follows. The progression through callbacks, events, and asynchronous control flow patterns reflects the evolution every Node.js developer experiences. The authors wisely dedicate two full chapters to async patterns, acknowledging that mastering asynchronous programming is crucial for Node.js success.

I am particularly excited about *Chapter 6, Coding with Streams*. You are probably using streams already, even if you do not realize it. Streams are the bedrock of Node.js, present everywhere from HTTP requests to file operations. Yet they remain one of the most misunderstood APIs in our ecosystem. Almost every developer thinks streams are complex until they understand the patterns behind them. This chapter demystifies streams in a way that will make you wonder why you ever found them difficult. The authors show you not just how to use streams, but how to think in streams and how to build pipelines that can process gigabytes of data with minimal memory footprint.

The design-pattern chapters that follow (Creational, Structural, and Behavioral) translate classic software engineering concepts to the Node.js context. This is not a mechanical port of Gang of Four patterns; it is a thoughtful adaptation that respects the event-driven, asynchronous nature of Node.js. *Chapter 10* on testing is essential reading because performance without reliability is meaningless.

What truly sets this book apart are the final chapters. *Chapter 12* on scalability and architectural patterns addresses the questions I hear most often: How do we scale? Should we use microservices or a modular monolith? The authors provide practical guidance based on real-world experience. *Chapter 13* on messaging and integration patterns is particularly relevant in today's distributed systems landscape, where Node.js applications rarely exist in isolation.

The patterns presented here are not academic exercises. They are the same patterns we use at Platformatic to build enterprise-grade Node.js applications. They power npm, handling millions of package downloads daily. They will also help you reduce your cloud bill by writing more efficient code.

Node.js is resilient, popular, and more relevant than ever. Despite claims of its decline, Node.js continues to thrive, and books like this are the reason why. They elevate our community's knowledge, helping developers move from "it works" to "it works at scale."

Whether you are building your first Node.js application or architecting microservices for millions of users, this book will transform how you think about Node.js development. The cost of not understanding these patterns is too high, literally in terms of your AWS bill and figuratively in terms of technical debt.

Dive in, embrace the patterns, and remember: we are not just writing code; we are building the future of the web. And with the knowledge in this book, you will be building it at ludicrous speed.

Matteo Collina

Co-Founder & CTO at Platformatic.dev

Chair of the Node.js Technical Steering Committee

Lead Maintainer of Fastify

P.S. If after reading this book your application is not at least 2x faster, you are probably not measuring correctly. Remember: always benchmark!

Contributors

About the authors

Luciano Mammino began his coding journey at the age of 12 on his father's old i386, and he hasn't stopped since. With over 15 years of experience in the software industry, he now serves as a senior architect at fourTheorem (fourtheorem.com), where he empowers global clients to fully leverage AWS and serverless technologies. As an AWS Serverless Hero and Microsoft MVP, Luciano is recognized for his expertise and contributions to the tech community. An active international speaker, he has delivered over 160 talks at conferences and meetups worldwide, sharing his knowledge and passion for cutting-edge technology. You can stay in touch with Luciano and explore his many passions on his blog at loige.co.

The biggest thanks of all goes to Mario Casciaro for inviting me into such an amazing project. Looking back over the past 11 years, it is incredible to think that this book, in its first edition, was what first introduced me to Node.js. Life came full circle when I had the privilege of becoming a co-author for the following three editions. It has been a fantastic journey, and I have learned and grown immensely while working together. I truly hope we will have many more opportunities to collaborate in the future.

Thanks to the entire Packt team and to everyone who has worked hard to bring this book to life across all four editions.

Thanks to our incredible reviewers, Maksim and Nick, along with all the early readers who provided feedback. This new edition would not be the same without your support and guidance.

A heartfelt thanks to Stefan Judis, Ludovico Besana, and Michael Di Prisco for generously sharing your expertise while reviewing the new chapter on testing. Your insightful comments and suggestions have taught me a great deal and have helped shape this new part of the book into something I am confident every reader will truly enjoy.

To our readers: so many of you have shared meaningful stories about how the previous editions of this book have had a positive impact on your career. That is the fuel that keeps us motivated to ensure it remains an up-to-date and valuable resource for every Node.js developer. I am grateful to everyone who has taken the time to leave a review, recommend the book to a friend or colleague, or offer feedback on what could be improved.

Thanks to my colleagues at fourTheorem, and especially to Fiona, Peter, and Eoin, for always trusting and empowering me even beyond the boundaries of my role.

A huge thank you to my family, whose love, encouragement, and belief in me have been my anchor and my driving force in chasing and achieving my goals.

And finally, a very special thanks to my lovely wife, Francesca. None of this would have been possible without you. Here's to all the chapters still waiting to be written in our shared story.

Mario Casciaro is a software architect, technology leader, and entrepreneur with a long career in building and scaling software products for mission-critical industries. Throughout his career, he has held roles ranging from software engineer to team leader to CTO. His passion for software and technology has led him to develop multiple side projects, launch a start-up, and, of course, write the bestselling book *Node.js Design Patterns*. His proudest moment is seeing his software being used for astronaut rescue operations.

I want to thank everyone who made this edition possible, and especially Luciano. Without him, and his relentless dedication and knowledge, this edition simply wouldn't exist. Thank you, Luciano.

About the reviewers

Nicholas Ramsbottom is a software developer based in London with experience in e-commerce, publishing, and marketing. He can be found swimming, cycling, and brewing beer when not coding.

Maksim Sinik is a senior engineering manager at TrustLayer, where he leads engineering teams developing innovative insurance compliance solutions using cloud computing and microservices architecture. As a core team member of Fastify, the high-performance Node.js web framework, he actively contributes to one of the fastest-growing open-source projects in the ecosystem. With deep expertise in Node.js, MongoDB, and DevOps practices, Maksim combines technical leadership with a passion for building scalable, reliable systems and leveraging technology to solve real-world problems and create positive impact worldwide.

Beta readers

Péter Szabó

Changho Lee

Nader

Table of Contents

Preface	xxv
Chapter 1: The Node.js Platform	1
The Node.js philosophy	2
Small core • 3	
Small modules • 3	
Small surface area • 5	
Simplicity and pragmatism • 5	
How Node.js works	6
I/O is often the bottleneck • 7	
Blocking I/O • 7	
Non-blocking I/O • 8	
Event demultiplexing • 9	
The reactor pattern • 11	
libuv, the I/O engine of Node.js • 13	
The complete recipe for Node.js • 14	
JavaScript in Node.js	15
Run the latest JavaScript with confidence • 15	
The module system • 16	
Full access to operating system services • 16	
Running native code • 17	
Node.js and TypeScript • 18	
<i>Using TypeScript with Node.js</i> • 18	
<i>The @types/node package</i> • 19	
Summary	20

Chapter 2: The Module System	21
The need for modules	22
Module systems in JavaScript and Node.js	22
The revealing module pattern	23
ES modules	25
Using ES modules in Node.js • 25	
The ES module syntax • 27	
<i>Named exports and imports • 27</i>	
<i>Default exports and imports • 29</i>	
<i>Mixed exports • 30</i>	
<i>Module identifiers • 32</i>	
<i>Static and dynamic imports • 33</i>	
The module resolution algorithm • 35	
Module loading in depth • 37	
<i>Loading phases • 37</i>	
<i>Read-only live bindings • 39</i>	
<i>Circular dependencies • 40</i>	
Modules that modify other modules • 46	
<i>How monkey patching affects type safety in TypeScript projects • 52</i>	
CommonJS modules	53
ES modules and CommonJS—differences and interoperability	54
Strict mode • 54	
Top-level await • 55	
Behavior of `this` • 55	
Missing references in ES modules • 56	
Import interoperability • 57	
<i>Import CommonJS modules from ES modules • 57</i>	
<i>Import ES modules from CommonJS • 59</i>	
Importing JSON files • 60	
Using modules in TypeScript	62
The role of the TypeScript compiler • 62	
Configuring the module output format • 64	
Input module syntax and output emission • 64	
Module resolution • 64	
Summary	65

Chapter 3: Callbacks and Events	67
The Callback pattern	68
The continuation-passing style • 68	
<i>Synchronous CPS</i> • 69	
<i>Asynchronous CPS</i> • 70	
<i>Non-CPS callbacks</i> • 73	
Synchronous or asynchronous? • 73	
<i>Writing an inconsistent function</i> • 73	
<i>Unleashing Zalgo</i> • 74	
<i>Using synchronous APIs</i> • 76	
<i>Guaranteeing asynchronicity with deferred execution</i> • 77	
Node.js callback conventions • 80	
<i>The callback is the last argument</i> • 80	
<i>Any error always comes first</i> • 80	
<i>Propagating errors</i> • 81	
<i>Avoiding uncaught exceptions</i> • 82	
The Observer pattern	84
The EventEmitter • 84	
Creating and using the EventEmitter • 85	
Propagating errors • 86	
Making any object observable • 87	
The risk of memory leaks • 88	
Synchronous and asynchronous events • 90	
EventEmitter versus callbacks	91
Combining callbacks and events	92
Summary	96
Exercises	97
 Chapter 4: Asynchronous Control Flow Patterns with Callbacks	 99
The challenges of asynchronous programming	100
Creating a simple web spider • 100	
Callback hell • 103	
Callback best practices	105
Callback discipline • 106	
Applying the callback discipline • 106	

Control flow patterns	109
Sequential execution • 109	
<i>Executing a known set of tasks in sequence • 109</i>	
<i>Sequential iteration • 110</i>	
Concurrent execution • 115	
<i>Web spider version 3 • 117</i>	
<i>The pattern • 118</i>	
<i>Fixing race conditions with concurrent tasks • 119</i>	
Limited concurrent execution • 121	
<i>Limiting concurrency • 122</i>	
<i>Globally limiting concurrency • 124</i>	
Summary	130
Exercises	130

Chapter 5: Asynchronous Control Flow Patterns with Promises and Async/Await 133

Promises	134
What is a promise? • 134	
Promises/A+ and thenables • 137	
The promise API • 138	
Creating a promise • 140	
Promisification • 141	
Sequential execution and iteration • 143	
Concurrent execution • 147	
Limited concurrent execution • 148	
<i>Implementing the TaskQueue class with promises • 148</i>	
<i>Updating the web spider • 150</i>	
Lazy promises • 153	
Async/await	157
Async functions and the await expression • 157	
Top-level await • 158	
Error handling with async/await • 159	
<i>A unified try...catch experience • 160</i>	
<i>The “return” versus “return await” trap • 161</i>	
Sequential execution and iteration • 162	
<i>Antipattern – using async/await with Array.forEach for serial execution • 164</i>	
Concurrent execution • 164	
Limited concurrent execution • 166	

The problem with infinite recursive promise resolution chains	167
Summary	170
Exercises	171

Chapter 6: Coding with Streams **173**

Discovering the importance of streams	175
Buffering versus streaming • 175	
Spatial efficiency • 176	
<i>Gzipping using a buffered API • 177</i>	
<i>Gzipping using streams • 178</i>	
Time efficiency • 178	
Composability • 183	
<i>Adding client-side encryption • 184</i>	
<i>Adding server-side decryption • 185</i>	
Getting started with streams	186
Anatomy of streams • 187	
Readable streams • 187	
<i>Reading from a stream • 187</i>	
<i>Implementing Readable streams • 190</i>	
Writable streams • 195	
<i>Writing to a stream • 195</i>	
<i>Backpressure • 197</i>	
<i>Implementing Writable streams • 199</i>	
Duplex streams • 201	
Transform streams • 202	
<i>Implementing Transform streams • 203</i>	
<i>Filtering and aggregating data with Transform streams • 206</i>	
PassThrough streams • 210	
<i>Observability • 210</i>	
<i>Late piping • 211</i>	
Lazy streams • 214	
Connecting streams using pipes • 215	
<i>Pipes and error handling • 216</i>	
<i>Better error handling with pipeline() • 217</i>	

Asynchronous control flow patterns with streams	218
Sequential execution • 219	
Unordered concurrent execution • 221	
<i>Implementing an unordered concurrent stream • 221</i>	
<i>Implementing a URL status monitoring application • 223</i>	
Unordered limited concurrent execution • 225	
Ordered concurrent execution • 227	
Piping patterns	229
Combining streams • 229	
<i>Implementing a combined stream • 231</i>	
Forking streams • 233	
<i>Implementing a multiple checksum generator • 234</i>	
Merging streams • 235	
<i>Merging text files • 235</i>	
Multiplexing and demultiplexing • 237	
<i>Building a remote logger • 238</i>	
<i>Multiplexing and demultiplexing object streams • 242</i>	
Readable stream utilities	243
Mapping and transformation • 243	
Filtering and iteration • 244	
Searching and evaluation • 244	
Limiting and reducing • 244	
Web Streams	246
Converting Node.js streams to Web Streams • 247	
Converting Web Streams to Node.js streams • 247	
Stream consumer utilities	249
Summary	251
Exercises	251

Chapter 7: Creational Design Patterns 253

Factory	254
Decoupling object creation and implementation • 255	
A mechanism to enforce encapsulation • 256	
Building a simple code profiler • 257	
In the wild • 260	

Builder	260
Implementing a URL object builder • 265	
In the wild • 268	
Revealing Constructor	271
Building an immutable buffer • 272	
In the wild • 274	
Singleton	275
Wiring modules	278
Singleton dependencies • 279	
Dependency Injection • 282	
Summary	286
Exercises	287
 Chapter 8: Structural Design Patterns	 289
Proxy	289
Techniques for implementing proxies • 290	
<i>Object composition</i> • 292	
<i>Object augmentation</i> • 295	
<i>The built-in Proxy object</i> • 296	
<i>A comparison of the different proxying techniques</i> • 299	
Creating a logging Writable stream • 300	
Change Observer with Proxy • 301	
In the wild • 304	
Decorator	304
Techniques for implementing decorators • 304	
<i>Composition</i> • 305	
<i>Object decoration</i> • 306	
<i>Decorating with the Proxy object</i> • 307	
Decorating a Level database • 309	
<i>Introducing Level and LevelDB</i> • 309	
<i>Implementing a Level plugin</i> • 309	
In the wild • 311	
The decorator proposal for ECMAScript • 312	

The line between Proxy and Decorator	313
Adapter	314
Using Level through the filesystem API • 314	
In the wild • 317	
Summary	318
Exercises	318
 Chapter 9: Behavioral Design Patterns	 321
Strategy	322
Multi-format configuration objects • 325	
In the wild • 330	
State	331
Implementing a basic failsafe socket • 332	
In the wild • 340	
Template	341
A configuration manager template • 342	
In the wild • 344	
Iterator	344
The iterator protocol • 345	
The iterable protocol • 348	
Iterators and iterables as a native JavaScript interface • 350	
Implementing the iterable protocol on iterators • 352	
Iterator utilities • 353	
Generators • 359	
<i>Generators in theory • 359</i>	
<i>A simple generator function • 360</i>	
<i>Controlling a generator iterator • 361</i>	
<i>How to use generators in place of iterators • 362</i>	
Async iterators • 364	
Async generators • 368	
Async iterators and Node.js streams • 369	
Async iterator utilities • 370	
In the wild • 372	

Middleware	373
Middleware in Express • 373	
Middleware as a pattern • 374	
Creating a middleware framework for ZeroMQ • 375	
<i>The middleware manager • 375</i>	
<i>Implementing the middleware to process messages • 378</i>	
<i>Using the ZeroMQ middleware framework • 379</i>	
In the wild • 381	
Command	382
The Task pattern • 383	
A more complex command • 383	
In the wild • 387	
Summary	388
Exercises	389
 Chapter 10: Testing: Patterns and Best Practices	 391
An introduction to software testing	392
Definitions • 393	
<i>System under test • 393</i>	
<i>Arrange, Act, Assert • 393</i>	
<i>Code coverage • 394</i>	
<i>Test doubles: Stubs, spies, and mocks • 394</i>	
<i>Test-driven development • 395</i>	
<i>Behavior-driven development • 396</i>	
<i>Continuous integration • 397</i>	
<i>Continuous delivery and continuous deployment • 397</i>	
Types of tests • 398	
<i>Unit tests • 399</i>	
<i>Integration tests • 400</i>	
<i>End-to-end tests • 400</i>	
<i>Other types of tests • 402</i>	
The testing pyramid • 403	
Writing tests with Node.js	404
Our first unit test • 404	

The Node.js test runner	407
Our first test with the Node.js test runner • 408	
Organizing tests • 409	
<i>Subtests • 411</i>	
<i>Subtest concurrency • 412</i>	
Parametrized test cases • 413	
<i>Test suites • 415</i>	
Test runner tips and tricks • 416	
<i>Watch mode • 416</i>	
<i>Targeted test execution with custom glob patterns • 417</i>	
Test reporters • 419	
<i>Collecting code coverage • 420</i>	
<i>Using the test runner with TypeScript • 423</i>	
Writing unit tests	423
Testing asynchronous code • 424	
Mocking • 430	
<i>Creating spies with mock.fn() • 430</i>	
<i>Mocking HTTP requests with the built-in test mock • 432</i>	
<i>Mocking HTTP requests with Undici • 435</i>	
<i>Mocking Node.js core modules • 438</i>	
<i>Mocking other dependencies • 441</i>	
<i>Problems with mocking imports • 445</i>	
<i>Mocking imports versus dependency injection • 446</i>	
Writing integration tests	449
Testing with a local database • 449	
Testing a web application • 457	
<i>Setting up the project • 458</i>	
Writing E2E tests	467
The application structure • 468	
<i>Home page • 468</i>	
<i>Sign-in and sign-up forms • 469</i>	
<i>Event Page • 469</i>	
<i>My reservations • 470</i>	
The user flow • 470	
Browser automation • 472	

Writing an E2E test with Playwright • 473	
<i>Setting up a new Playwright project</i> • 473	
<i>Understanding the Playwright API</i> • 475	
<i>Understanding timeouts</i> • 479	
<i>Testing our user flow</i> • 479	
Summary	484
Exercises	484
 Chapter 11: Advanced Recipes	 487
Dealing with asynchronously initialized components	488
The issue with asynchronously initialized components • 488	
Local initialization check • 490	
Delayed startup • 491	
Pre-initialization queues • 492	
<i>Using the State pattern</i> • 495	
In the wild • 498	
Asynchronous request batching and caching	498
What's asynchronous request batching? • 498	
Optimal asynchronous request caching • 500	
An API server without caching or batching • 502	
Batching and caching with promises • 505	
<i>Batching requests</i> • 505	
<i>Batching and caching requests</i> • 507	
Canceling asynchronous operations	509
A basic recipe for creating cancelable functions • 509	
Wrapping asynchronous invocations • 511	
Cancelable async functions with AbortController • 512	
Running CPU-bound tasks	516
Solving the subset sum problem • 516	
Interleaving with setImmediate • 520	
<i>Interleaving the steps of the subset sum algorithm</i> • 520	
<i>Considerations on the interleaving approach</i> • 522	
Using external processes • 522	
<i>Delegating the subset sum task to an external process</i> • 523	
<i>Considerations for the multi-process approach</i> • 529	

Using worker threads • 530	
<i>Running the subset sum task in a worker thread • 531</i>	
Running CPU-bound tasks in production • 534	
Summary	534
Exercises	534
 Chapter 12: Scalability and Architectural Patterns	 537
An introduction to application scaling	538
Scaling Node.js applications • 538	
The three dimensions of scalability • 538	
<i>X-axis: cloning • 539</i>	
<i>Y-axis: decomposing by service or functionality • 540</i>	
<i>Z-axis: splitting by data partition • 540</i>	
<i>Combining strategies across the scale cube • 540</i>	
Cloning and load balancing	541
The cluster module • 542	
<i>Notes on the behavior of the cluster module • 543</i>	
<i>Building a simple HTTP server • 544</i>	
<i>Scaling with the cluster module • 545</i>	
<i>Resiliency and availability with the cluster module • 547</i>	
<i>Zero-downtime restart • 549</i>	
Dealing with stateful communications • 552	
<i>Sharing the state across multiple instances • 553</i>	
<i>Sticky load balancing • 554</i>	
Scaling with a reverse proxy • 555	
<i>Load balancing with Nginx • 557</i>	
Dynamic horizontal scaling • 561	
<i>Using a service registry • 562</i>	
<i>Implementing a dynamic load balancer • 563</i>	
Peer-to-peer load balancing • 570	
<i>Implementing an HTTP client that can balance requests across multiple servers • 572</i>	
Scaling applications using containers • 574	
<i>What is a container? • 574</i>	
<i>Creating and running a container with Docker • 575</i>	
<i>What is Kubernetes? • 578</i>	
<i>Deploying and scaling an application on Kubernetes • 580</i>	

Decomposing complex applications	583
Monolithic architecture • 583	
The microservice architecture • 585	
<i>An example of a microservice architecture • 586</i>	
<i>Microservices: advantages and disadvantages • 587</i>	
Integration patterns in a microservice architecture • 588	
<i>The API proxy • 589</i>	
<i>API orchestration • 590</i>	
<i>Integration with a message broker • 593</i>	
Summary	595
Exercises	595
 Chapter 13: Messaging and Integration Patterns	 597
Fundamentals of a messaging system	598
One way versus request/reply patterns • 598	
Message types • 599	
<i>Command messages • 599</i>	
<i>Event messages • 600</i>	
<i>Document messages • 600</i>	
Push versus pull delivery semantics • 601	
<i>Pull delivery (consumer-initiated) • 601</i>	
<i>Push delivery (producer-initiated) • 601</i>	
<i>Choosing between pull and push delivery • 601</i>	
Asynchronous messaging, queues, and streams • 602	
Peer-to-peer or broker-based messaging • 604	
Publish/Subscribe pattern	605
Building a minimalist real-time chat application • 606	
<i>Implementing the server side • 606</i>	
<i>Implementing the client side • 607</i>	
<i>Running and scaling the chat application • 610</i>	
Using Redis as a simple message broker • 611	
Peer-to-peer Publish/Subscribe with ZeroMQ • 615	
<i>Introducing ZeroMQ • 615</i>	
<i>Designing a peer-to-peer architecture for the chat server • 615</i>	
<i>Using the ZeroMQ PUB/SUB sockets • 616</i>	

Reliable message delivery with queues • 619	
<i>Introducing AMQP • 621</i>	
<i>Durable subscribers with AMQP and RabbitMQ • 623</i>	
Reliable messaging with streams • 629	
<i>Characteristics of a streaming platform • 630</i>	
<i>Streams versus message queues • 631</i>	
<i>Implementing the chat application using Redis streams • 631</i>	
Task distribution patterns	635
The ZeroMQ fan-out/fan-in pattern • 637	
<i>PUSH/PULL sockets • 638</i>	
<i>Building a distributed hashsum cracker with ZeroMQ • 638</i>	
Pipelines and competing consumers in AMQP • 646	
<i>Point-to-point communications and competing consumers • 646</i>	
<i>Implementing the hashsum cracker using AMQP • 647</i>	
Distributing tasks with Redis streams • 650	
<i>Redis consumer groups • 650</i>	
<i>Implementing the hashsum cracker using Redis streams • 651</i>	
Request/Reply patterns	656
Correlation Identifier • 656	
<i>Implementing a request/reply abstraction using correlation identifiers • 657</i>	
Return address • 662	
<i>Implementing the Return Address pattern in AMQP • 662</i>	
Summary	668
Exercises	668
 Other Books You May Enjoy	 673
 Index	 677

Preface

Over the last decade, Node.js has grown from an experimental runtime into one of the most important technologies in modern web development. It is now a staple in companies of every size, powering everything from quick prototypes to some of the most ambitious, large-scale systems in production today. This rise is no accident. Node.js combines the flexibility and reach of JavaScript, a language understood by millions and supported in every browser, with an event-driven asynchronous architecture that excels at handling I/O-bound workloads. On top of this, a thriving ecosystem of modules, frameworks, and tools has emerged, providing solutions for almost any problem a developer might face.

The ability to use the same language on both the client and the server has broken down barriers between frontend and backend work. Models, validation rules, and utility functions can be shared across the stack, creating tighter collaboration between teams and reducing duplication. This has opened the door to single-language application stacks that are fast to build, easier to maintain, and more adaptable to change.

Yet the very qualities that make Node.js appealing can also make it challenging. The asynchronous nature of JavaScript, while powerful, takes time and practice to master. The Node.js core library evolves with every release, adding features that are often underused or poorly understood. Choosing and integrating third-party modules can be overwhelming, especially when quality, maintenance, and compatibility vary. And building production-grade applications almost always requires integration with databases, caches, message brokers, and other services, where architectural decisions can have lasting consequences on performance, scalability, and maintainability.

These are not abstract concerns. They appear in real projects every day. The following are examples:

- Code that works during development but slows to a crawl under load
- Asynchronous flows that are hard to follow, hard to test, and even harder to debug
- Modules assembled without a clear structure, making future changes risky and expensive
- Services that scale in one dimension but become bottlenecks in another
- Integrations that fail when the volume of data or traffic increases unexpectedly

This book was written to address these challenges directly. *Node.js Design Patterns* distills years of experience building large, distributed, high-performance systems into a structured, practical, and approachable guide. It covers the essential concepts that every Node.js developer should understand, such as the event loop, the module system, streams, and asynchronous programming techniques.

From this foundation, it introduces a broad set of patterns to solve recurring design problems. You will find most of the traditional patterns from the original *Gang of Four* book adapted to the realities of JavaScript and Node.js, alongside modern patterns that have emerged from the unique nature of the platform. These are patterns you can reuse in your own projects to tackle common challenges with clarity and consistency.

Patterns are more than recipes. They are a shared vocabulary and a way of thinking about problems. Once you know them, you can approach design decisions with a library of proven solutions in mind, instead of reinventing the wheel each time. This makes your code not only more robust and maintainable but also easier to discuss and share with other developers.

This book has been designed to be a loyal guide. You can read it from beginning to end, following the progression from fundamentals to advanced architecture, or you can keep it at your side as a long-term reference. Each chapter blends theory with practical examples and exercises, so you can put what you have learned into practice immediately and confirm that the concepts are clear before moving on. The examples are realistic and grounded in the kinds of problems you will face in production. The exercises give you the opportunity to test your understanding and refine your skills.

Over the years, *Node.js Design Patterns* has become one of the most trusted resources for mastering Node.js, helping tens of thousands of developers worldwide and being translated into multiple languages. This fourth edition builds on that foundation with fully updated content for Node.js 24; modern JavaScript features such as ECMAScript modules and `async/await`; a brand-new chapter dedicated to testing with practical strategies for unit, integration, and end-to-end tests; and expanded coverage of scalability, security, and architecture for today's production environments.

The main themes you will encounter throughout the book mirror the most common challenges in real-world Node.js development:

- Mastering asynchronous programming so that callbacks, promises, and `async/await` become tools you use with confidence rather than sources of bugs and confusion
- Designing modular, maintainable architectures using proven patterns adapted to JavaScript's capabilities and constraints
- Building scalable systems by understanding how to make the most of Node.js's strengths and knowing when to use clustering, containers, and distributed architectures
- Integrating effectively with other technologies, from databases to message brokers, so that your applications can grow without losing performance or reliability

This book is also about the developer's journey. While it is rooted in Node.js, it goes beyond the runtime to cover topics that are essential for any senior engineer: security, scalability, microservices, systems architecture, messaging patterns, and testing strategies. These concepts are just as relevant in other programming environments, and mastering them will prepare you to contribute confidently to complex and ambitious projects. Our goal is to help you grow from a junior or mid-level developer into a professional who can design and deliver production-grade systems with skill and clarity. In many ways, this is the book we wished we had when we began our own web development careers: a single resource that combines the theory, the practice, and the patterns we have learned over our years of experience.

If you are returning from a previous edition, thank you for trusting this book once again as your reference. If you are reading for the first time, welcome. We hope you find in these pages the same value that has helped tens of thousands of developers around the world build better systems, sharpen their skills, and take the next steps in their careers.

Also, if you are just starting to work with Node.js, you are very welcome in this vibrant community. It is a friendly one, so do not hesitate to reach out, share your progress, and even contribute by publishing your own modules or helping others with their projects. After all, that is the beauty of open-source communities: we are all here to get better together, and every small contribution makes a real difference.

Let's keep building great things together.

Who this book is for

This book is for developers who have already had initial contact with Node.js and now want to get the most out of it in terms of productivity, design quality, and scalability. You are only required to have had some prior exposure to the technology through some basic examples and some degree of familiarity with the JavaScript language, since this book will cover some basic concepts as well. Developers with intermediate experience in Node.js will also find the techniques presented in this book beneficial.

Some background in software design theory is also an advantage to understand some of the concepts presented.

This book assumes that you have a working knowledge of web application development, web services, databases, and data structures.

What this book covers

Chapter 1, The Node.js Platform, introduces the “Node way” of building applications and why its asynchronous mindset is both powerful and challenging. It covers the Node.js architecture, the **event loop**, the **reactor** pattern, and the implications of running JavaScript on the server. These foundational concepts shape everything from how you handle I/O to how you structure your applications.

Chapter 2, The Module System, explores the Node.js module system in depth, with a focus on **ECMAScript modules (ESM)** as the modern standard for writing modular JavaScript. It explains how ESM differs from **CommonJS** (Node.js's original module system), how to manage interoperability between the two, and how TypeScript fits into the picture. This chapter will help you avoid common pitfalls and work confidently with modules.

Chapter 3, Callbacks and Events, introduces Node.js's core asynchronous patterns: callbacks and the event-driven model. You will learn how callbacks work, how to avoid common mistakes such as **Zalgo text** and uncaught exceptions, and how to use the **Observer** pattern and **EventEmitter** to respond to asynchronous events predictably.

Chapter 4, Asynchronous Control Flow Patterns with Callbacks, presents proven techniques for managing asynchronous operations using callbacks. You will learn about *sequential*, *concurrent*, and *limited concurrent* execution patterns, along with best practices for avoiding **callback hell** and writing maintainable asynchronous code.

Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await, shows how **Promises** and `async/await` simplify asynchronous programming. It explains how Promises work under the hood, common pitfalls such as the `return await` trap, and how to apply these modern techniques to write clean, robust asynchronous flows.

Chapter 6, Coding with Streams, dives into one of Node.js's most powerful features: **streams**. You will learn how to use, compose, and create streams to process data efficiently, handle *backpressure*, and build stream pipelines. The chapter covers both fundamental and advanced stream patterns that are essential for building performant and memory-efficient applications.

Chapter 7, Creational Design Patterns, introduces classic object creation patterns adapted for JavaScript and Node.js, including **Factory**, **Builder**, **Singleton**, and **Dependency Injection**, as well as novel patterns that are JavaScript-specific, such as the **Revealing Constructor** pattern. These patterns will help you simplify object creation and enforce clean, extensible boundaries in your code.

Chapter 8, Structural Design Patterns, explores how to organize and compose your code using patterns such as **Proxy**, **Decorator**, and **Adapter**. You will see how to dynamically extend objects, intercept behaviors, and adapt interfaces, with real-world examples from the Node.js ecosystem.

Chapter 9, Behavioral Design Patterns, focuses on how objects interact and behave over time. Patterns covered include **Strategy**, **State**, **Command**, **Middleware**, and **Iterator** (including **Generators** and **Async Iterators**). The chapter shows how Node.js often gives these patterns a more expressive twist compared to their traditional forms.

Chapter 10, Testing: Patterns and Best Practices, emphasizes that testing is a mindset as well as a skill. It covers **unit testing**, **integration testing**, and **end-to-end testing**, showing how to structure tests, isolate dependencies, and make full use of the Node.js built-in test runner and tools such as **Playwright**.

Chapter 11, Advanced Recipes, takes a problem-solution approach to common but tricky Node.js challenges. Topics include **asynchronous initialization**, **batching**, **caching**, **cancellation**, and handling CPU-bound tasks. Each recipe is based on real-world needs and is ready to adapt to your own projects.

Chapter 12, Scalability and Architectural Patterns, teaches techniques for scaling Node.js applications using **clustering**, **containers**, **load balancing**, and **microservices**. You will learn how to design for scale with the *scale cube* model, and how to deploy resilient systems using *Docker* and *Kubernetes*.

Chapter 13, Messaging and Integration Patterns, covers the patterns that make distributed systems reliable and maintainable. It explains one-way and request/reply messaging, event and command messages, brokered versus peer-to-peer topologies, and hands-on integration with tools such as **ZeroMQ**, **RabbitMQ**, and **Redis streams**.

To get the most out of this book

To experiment with the code, you will need a working installation of Node.js version 24 (or greater) and npm version 11 (or greater) or another package manager such as yarn or pnpm. If some examples will require you to use some extra tooling, these will be described accordingly in place. You will also need to be familiar with the command prompt, know how to install a package, and know how to run Node.js applications. Finally, you will need a text editor to work with the code and a modern web browser.

To get the most out of this book, you can download the example code files and the color images as per the following instructions.

Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Node.js-Design-Patterns-Fourth-Edition>. If there's an update to the code, it will be updated in the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781803238944>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “ES2015 introduces the `let` keyword to declare variables that respect the block scope.”

A block of code is set as follows:

```
import zmq from 'zeromq'

async function main() {
  const sink = new zmq.Pull()
  await sink.bind('tcp://*:5017')

  for await (const rawMessage of sink) {
    console.log('Message from worker: ', rawMessage.toString())
  }
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
const wss = new ws.Server({ server })
wss.on('connection', client => {
  console.log('Client connected')
  client.on('message', msg => {
    console.log(`Message: ${msg}`)
    redisPub.publish('chat_messages', msg)
  })
})
```

Any command-line input or output is written as follows:

```
node replier.js
node requestor.js
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “To explain the problem, we will create a little **web spider**, a command-line application that takes in a web URL as the input and downloads its contents locally into a file.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Most URLs are linked through our own short URL system to make it easier for readers using the print edition to access them. These links are in the form `nodejsdp.link/some-descriptive-id`.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Share your thoughts

Once you've read *Node.js Design Patterns — Fourth Edition*, we'd love to hear your thoughts! Please click [here](#) to go straight to the Amazon review page for this book and share your feedback.

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

Your Book Comes with Exclusive Perks - Here's How to Unlock Them

Unlock this book's exclusive benefits now

Scan this QR code or go to <https://packtpub.com/unlock>, then search this book by name. Ensure it's the correct edition.

UNLOCK NOW



Note: Keep your purchase invoice ready before you start.



Enhanced reading experience with our Next-gen Reader:

- 🔄 **Multi-device progress sync:** Learn from any device with seamless progress sync.
- 📖 **Highlighting and notetaking:** Turn your reading into lasting knowledge.
- 🔖 **Bookmarking:** Revisit your most important learnings anytime.
- 🌙 **Dark mode:** Focus with minimal eye strain by switching to dark or sepia mode.

Learn smarter using our AI assistant (Beta):

- 💡 **Summarize it:** Summarize key sections or an entire chapter.
- 💡 **AI code explainers:** In the next-gen Packt Reader, click the **Explain** button above each code block for AI-powered code explanations.



Note: The AI assistant is part of next-gen Packt Reader and is still in beta.

Learn anytime, anywhere:

 Access your content offline with DRM-free PDF and ePub versions—compatible with your favorite e-readers.

Unlock Your Book's Exclusive Benefits

Your copy of this book comes with the following exclusive benefits:

-  Next-gen Packt Reader
-  AI assistant (beta)
-  DRM-free PDF/ePub downloads

Use the following guide to unlock them if you haven't already. The process takes just a few minutes and needs to be done only once.

How to unlock these benefits in three easy steps

Step 1

Keep your purchase invoice for this book ready, as you'll need it in *Step 3*. If you received a physical invoice, scan it on your phone and have it ready as either a PDF, JPG, or PNG.

For more help on finding your invoice, visit <https://www.packtpub.com/unlock-benefits/help>.



Note: Did you buy this book directly from Packt? You don't need an invoice. After completing Step 2, you can jump straight to your exclusive content.

Step 2

Scan this QR code or go to <https://packtpub.com/unlock>.



On the page that opens (which will look similar to Figure 0.1 if you're on desktop), search for this book by name. Make sure you select the correct edition.

The screenshot shows the Packt unlock landing page. At the top is the Packt logo and a search bar. Below the logo are navigation links: Explore Products, Best Sellers, New Releases, Books, Videos, Audiobooks, Learning Hub, Newsletter Hub, and Free Learning. The main heading is "Discover and unlock your book's exclusive benefits". Below this is a subheading: "Bought a Packt book? Your purchase may come with free bonus benefits designed to maximise your learning. Discover and unlock them here". A progress bar shows three steps: "Discover Benefits" (active), "Sign Up/In", and "Upload Invoice". On the right, there are links for "Subscription", a shopping cart icon, and a user profile icon. A "Need Help?" link is also present. The main content area has a section titled "1. Discover your book's exclusive benefits" with a search bar labeled "Search by title or ISBN" and a "CONTINUE TO STEP 2" button. Below this are two more sections: "2. Login or sign up for free" and "3. Upload your invoice and unlock", each with a dropdown arrow.

Figure 0.1: Packt unlock landing page on desktop

Step 3

Sign in to your Packt account or create a new one for free. Once you're logged in, upload your invoice. It can be in PDF, PNG, or JPG format and must be no larger than 10 MB. Follow the rest of the instructions on the screen to complete the process.

Need help?

If you get stuck and need help, visit <https://www.packtpub.com/unlock-benefits/help> for a detailed FAQ on how to find your invoices and more. The following QR code will take you to the help page directly:



Note: If you are still facing issues, reach out to customer@packt.com.

1

The Node.js Platform

Over the past decade, Node.js has become a cornerstone of modern web development, revolutionizing how companies of all sizes approach building scalable and high-performance applications. Its ability to handle asynchronous operations allows for real-time, responsive solutions that can manage great amounts of concurrent connections efficiently. Node.js offers a unified environment where JavaScript can run seamlessly on both the client and the server, streamlining the development process.

Mastering Node.js goes beyond learning its syntax and libraries; in fact, some key ideas and patterns shape how developers use Node.js and its ecosystem. Its asynchronous nature means that we need to embrace asynchronous primitives such as callbacks, promises, and `async/await`, and those come with their own unique challenges and patterns. In this first chapter, we'll look into why Node.js works this way. This isn't just theoretical: knowing how Node.js works at its core will give you a strong foundation for understanding the reasoning behind the more complex topics and patterns that we will cover later in the book.

Another defining aspect of Node.js is its opinionated philosophy. Embracing Node.js means joining a culture and community that deeply influence how we design applications and interact with the broader ecosystem.

In this chapter, you'll discover:

- The Node.js philosophy or the “Node way”
- The reactor pattern—the mechanism at the heart of the Node.js asynchronous event-driven architecture
- What it means to run JavaScript on the server compared to the browser



To highlight the importance of this philosophy, let me share a bit of my (Luciano's) journey with Node.js. As a web developer, I had a solid background in frontend JavaScript (yes, lots of jQuery!), so when I discovered Node.js—version 0.12 at the time—I was excited to use JavaScript on the backend and replace the other languages I was working with, like PHP, Java, and .NET.

I started learning Node.js with the first edition of this very book, which Mario had authored solo. After finishing the book, I was eager to build something, so I took on a personal project: downloading an entire photo gallery from Flickr (a popular photo hosting site at the time). Flickr didn't offer a way to download all the photos in one go, so I decided to use its API and my new Node.js skills to create a CLI tool that could do just that.

This project was the perfect opportunity to leverage Node.js's asynchronous nature. Downloading hundreds of files from URLs is ideal for concurrency—there's no reason to fetch them one by one when you can download several at once. However, doing this effectively required limiting concurrency to avoid overwhelming system resources like memory and network bandwidth. After building the tool, I shared the project on GitHub ([nodejsdp.link/flickr-set-get](#)) and sought feedback from various Node.js communities. And I got a lot of feedback! Many people pointed out that, while my code worked, it didn't fully embrace the “Node way.” My approach still had traces of a PHP-like style, which made it harder to integrate smoothly with the broader Node.js ecosystem.

That feedback was invaluable—it helped me improve my Node.js skills and understand the importance of adopting its design principles. And here's a fun twist: Mario was one of the people who gave me feedback on that project! That interaction sparked a connection between us, which eventually led to me joining him as a co-author for the second edition of the book a few years later.

So, maybe there's a lesson in not being afraid to share your work and ask for candid feedback. The Node.js community is quite supportive and helpful, providing countless opportunities to learn and grow. Who knows where those opportunities might lead!

Enough with the motivational story—let's dive into some learning.

The Node.js philosophy

Every programming platform has its own philosophy, a set of principles and guidelines that are generally accepted by the community, or an ideology for doing things that influence both the evolution of the platform and how applications are developed and designed. Some of these principles arise from the technology itself, some of them are enabled by its ecosystem, some are just trends in the community, and others are evolutions of ideologies borrowed from other platforms. In Node.js, some of these principles come directly from its creator—Ryan Dahl—while others come from the people who contribute to the core or from charismatic figures in the community, and, finally, some are inherited from the larger JavaScript movement.

These guidelines aren't strict rules and should be used with pragmatism. However, they can be very helpful when you're looking for inspiration in designing your software.



If you are curious to see other examples of software development philosophies, you can find an extensive list on Wikipedia at [nodejsdp.link/dev-philosophies](https://en.wikipedia.org/wiki/Node.js/development_philosophies).

Small core

Historically, the Node.js core, which includes the runtime and built-in modules, has been kept minimal, with most features left to the “**userland**” (or **userspace**)—the ecosystem of modules outside the core. This approach has allowed the community to experiment and develop new solutions quickly, rather than relying on a single, slowly evolving core solution. Keeping the core minimal makes it easier to maintain and positively impacts the community by encouraging innovation in the userland modules. In recent years, the principle of keeping the Node.js core minimal has become less strict. The community has shown interest in having more built-in features, so several important capabilities have been added directly to the core. These include command-line argument parsing, WebSockets, a unit testing framework, file watch capability, file globbing, the web `fetch` API, and more. This shift doesn't change the core principles of the Node.js community but reflects its evolution. Many common interfaces have become mature and stable, making it sensible to include them in the core for easy access without needing third-party libraries.

Small modules

Node.js uses the concept of a **module** as the fundamental means for structuring the code of a program. It is the building block for creating applications and reusable libraries. In Node.js, one of the most evangelized principles is designing small modules (and packages), not only in terms of raw code size but also, most importantly, in terms of scope.

This principle has its roots in the Unix philosophy, and particularly in two of its precepts, which are as follows:

- “Small is beautiful.”
- “Make each program do one thing well.”

Node.js has brought these concepts to a whole new level. Along with the help of its module managers—with **npm**, **pnpm**, and **yarn** being the most popular—Node.js helps to solve the *dependency hell* problem by making sure that two (or more) packages depending on different versions of the same package will use their own installations of such a package, thus avoiding conflicts. This aspect allows packages to depend on a high number of small, well-focused dependencies without the risk of creating conflicts.

For example, suppose we have a project that uses two dependencies: `depA` and `depB`. Both `depA` and `depB` rely on a third library, `depC`, but they need different versions—`depA` requires `depC@1.0.0`, while `depB` requires `depC@2.0.0`. Node.js handles this without conflict by organizing the dependencies like this:

```

.
├── node_modules
│   ├── depA@1.0.0
│   │   └── node_modules
│   │       └── depC@1.0.0
│   └── depB@1.0.0
│       └── node_modules
│           └── depC@2.0.0

```

 **Quick tip:** Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button

(1) to quickly copy code into your coding environment, or click the **Explain** button

(2) to get the AI assistant to explain a block of code to you.



```
function calculate(a, b) {
  return {sum: a + b};
};
```

Copy

1

Explain

2

 **The next-gen Packt Reader** is included for free with the purchase of this book. Go to <https://packtpub.com/unlock>, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.

Here, two versions of `depC` are installed: one under `depA` for version `1.0.0` and another under `depB` for version `2.0.0`. This way, Node.js ensures that when `depA` needs `depC`, it loads version `1.0.0`, and when `depB` needs `depC`, it loads version `2.0.0`, avoiding version conflicts entirely.

While this can be considered unpractical or even totally unfeasible in other platforms, in Node.js, this practice is the norm. This enables extreme levels of reusability; they are so extreme, in fact, that sometimes we can find packages comprising of a single module containing just a few lines of code—for example, `is-sorted`, a library to check if an array is sorted (`nodejsdp.link/is-sorted`).

Besides the clear advantage in terms of reusability, a small module is also:

- Easier to understand and use
- Simpler to test and maintain
- Lightweight in terms of kilobytes, ideal in the browser (since the npm registry is used by both Node.js and frontend applications), and in serverless environments that require quick start times such as AWS Lambda

Having smaller and more focused modules empowers everyone to share or reuse even the smallest piece of code; it's the **Don't Repeat Yourself (DRY)** principle applied at a whole new level.



While the principle of small modules applies, the recent rise in supply chain vulnerabilities (nodejsdp.link/supply-chain) has made the software industry more cautious about adding third-party dependencies. This is true for Node.js projects as well. It's important to carefully evaluate if a third-party module is well-maintained and if adding a new dependency is necessary. The more third-party dependencies, the higher the risk of one of them getting compromised and affecting the project's security.

Small surface area

In addition to being small in size and scope, a desirable characteristic of Node.js modules is exposing a minimal set of functionalities to the outside world. This has the effect of producing an API that is clearer to use and less susceptible to erroneous usage. In fact, most of the time, the user of a component is only interested in a very limited and focused set of features, without needing to extend its functionality or tap into more advanced aspects.

In Node.js, a very common pattern for defining modules is to expose only one functionality, such as a function or a class, for the simple fact that it provides a single, unmistakably clear entry point.

Another characteristic of many Node.js modules is that they are designed to be used, not extended. Locking down the internals of a module by preventing extensions might seem inflexible, but it simplifies implementation, makes maintenance easier, and improves usability. In practice, this means preferring to expose functions instead of classes and ensuring no internals are exposed to the outside world.

Simplicity and pragmatism

Have you ever heard of the **Keep It Simple, Stupid (KISS)** principle? Richard P. Gabriel, a prominent computer scientist, coined the term “worse is better” to describe the model whereby less and simpler functionality is a good design choice for software. In his essay *The Rise of “Worse is Better”*, he says:



The design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.”

Designing simple, as opposed to perfect, fully featured software is a good practice for several reasons:

- It takes less effort to implement.
- It allows shipping faster with fewer resources.
- It's easier to adapt.
- It's easier to maintain and understand.

The positive effects of these factors encourage community contributions and allow the software itself to grow and improve.

In Node.js, this principle is supported by JavaScript's pragmatic nature. Instead of using complex class hierarchies, it's common to see simple classes, functions, and closures. Pure object-oriented designs often attempt to model the real world using mathematical concepts, which can overlook the real world's imperfections and complexities. In reality, software is always an approximation of reality. We are likely to be more successful if we focus on getting something functional quickly with manageable complexity, rather than striving for nearly perfect software abstractions that require extensive effort and tons of code to maintain.

Throughout this book, you'll see this principle applied often. For example, traditional design patterns like Singleton or Decorator can be implemented simply, which might not be perfect but is usually practical.

In Node.js, we prefer straightforward and practical solutions over complex, flawless designs. This doesn't mean we're lowering our standards; rather, we carefully weigh the trade-offs between extensive coverage and complexity versus simplicity and clear boundaries.

Next, we will take a look inside the Node.js core to reveal its internal patterns and event-driven architecture.

How Node.js works

In this section, you'll learn how Node.js operates internally and be introduced to the reactor pattern, which is central to Node.js' asynchronous nature. We'll cover key concepts like the single-threaded architecture and non-blocking I/O, and show how these elements form the basis of the Node.js platform. Understanding how Node.js works will be crucial for mastering the runtime and writing clean, efficient code.



While we commonly describe Node.js as “single-threaded” due to its ability to handle asynchronous tasks concurrently on a single thread, this doesn't mean it can't leverage background threads for certain operations. Node.js uses a single thread for the event loop, but when needed, it can execute CPU-intensive tasks on separate threads, allowing for more efficient handling of complex operations. In *Chapter 11, Advanced Recipes*, we'll explore how to use worker threads to perform such CPU-heavy tasks in parallel, demonstrating how Node.js can go beyond its single-threaded nature when necessary.

I/O is often the bottleneck

Regardless of your choice of programming language, I/O (short for input/output) is definitely the slowest among the fundamental operations of a computer. Accessing the RAM is in the order of nanoseconds ($10E-9$ seconds), while accessing data on the disk or the network is in the order of milliseconds ($10E-3$ seconds). The same applies to the bandwidth. RAM has a transfer rate consistently in the order of GB/s, while the disk or network varies from MB/s to, optimistically, GB/s. I/O is usually not expensive in terms of CPU, but it adds a delay between the moment the request is sent to the device and the moment the operation completes.

On top of that, we have to consider the human factor. In fact, in many circumstances, the input of an application comes from a real person—a mouse click, for example—so the speed and frequency of I/O don't only depend on technical aspects, and it can be many orders of magnitude slower than the disk or network.

Despite the inherent slowness of I/O, Node.js is designed to handle it with remarkable efficiency. Its non-blocking, event-driven architecture allows the system to remain responsive even while waiting for I/O operations to complete, making it an excellent choice for applications that need to perform large amounts of I/O without sacrificing performance. But to understand how the non-blocking model of Node.js works, we need to briefly explore the concept of blocking I/O.

Blocking I/O

In traditional blocking I/O programming, the function call corresponding to an I/O request will block the execution of the thread until the operation completes. This can range from a few milliseconds, in the case of disk access, to minutes or even more, in the case of data being generated from user actions, such as pressing a key. The following pseudocode shows a typical blocking thread performed against a socket:

```
// blocks the thread until the data is available  
data = socket.read()  
// data is available  
print(data)
```

It is really important to understand that a web server that is implemented using blocking I/O will not be able to handle multiple connections in the same thread. This is because each I/O operation on a socket will block the processing of any other connection. The traditional approach to solving this problem is to use a separate thread (or process) to handle each concurrent connection. This way, a thread blocked on an I/O operation will not impact the availability of the other connections because they are handled in separate threads.

The following diagram illustrates this scenario:

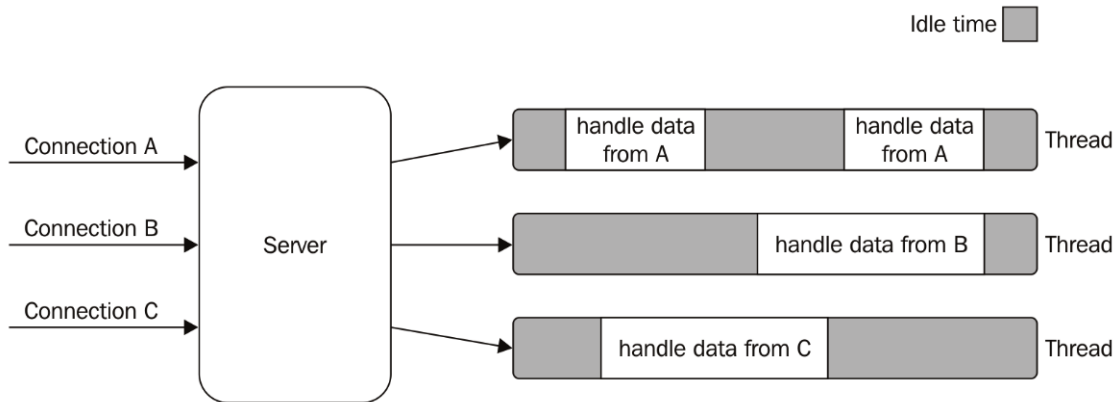


Figure 1.1: Using multiple threads to process multiple connections

Figure 1.1 lays emphasis on the amount of time each thread is idle and waiting for new data to be received from the associated connection. Now, if we also consider that any type of I/O can possibly block a request—for example, while interacting with databases or with the filesystem—we will soon realize how many times a thread must block in order to wait for the result of an I/O operation. Unfortunately, a thread is not cheap in terms of system resources—it consumes memory and causes context switches—so having a long-running thread for each connection and not using it for most of the time means wasting precious memory and CPU cycles.

The key takeaway here is that achieving concurrency with a traditional blocking approach typically requires multiple threads, which are costly in terms of system resources. In contrast, Node.js uses a non-blocking, single-threaded approach that can be far more efficient when handling I/O operations.



For a deeper dive into the concepts discussed, we recommend checking out this Wikipedia page on computing threads: [nodejsdp.link/thread](https://en.wikipedia.org/wiki/Thread_(computer_science)).

Non-blocking I/O

In addition to blocking I/O, most modern operating systems support another mechanism to access resources, called non-blocking I/O. In this operating mode, the system call always returns immediately without waiting for the data to be read or written. If no results are available at the moment of the call, the function will simply return a predefined constant, indicating that there is no data available to return at that moment.

For example, in Unix operating systems, the `fcntl()` function is used to modify an existing file descriptor (which is a reference to a local file or network socket) to change its operating mode to non-blocking using the `O_NONBLOCK` flag. When a resource is in non-blocking mode, any read operation will return the error code `EAGAIN` if there is no data ready to be read.

The simplest way to handle non-blocking I/O is to actively check the resource in a loop until data is available, a method known as **busy-waiting**. The following pseudocode demonstrates how to read from multiple resources using non-blocking I/O and an active polling loop:

```
resources = [socketA, socketB, fileA]
while (!resources.isEmpty()) {
    for (resource of resources) {
        // try to read
        data = resource.read()
        if (data === NO_DATA_AVAILABLE) {
            // there is no data to read at the moment
            continue
        }
        if (data === RESOURCE_CLOSED) {
            // the resource was closed, remove it from the list
            resources.remove(i)
        } else {
            //some data was received, process it
            consumeData(data)
        }
    }
}
```

As you can see, with this simple technique, it is possible to handle different resources in the same thread, but it's still not efficient. In fact, in the preceding example, the loop will consume precious CPU cycles for iterating over resources that are unavailable most of the time. Polling algorithms usually result in a huge amount of wasted CPU time. Let's see how we can implement non-blocking I/O in a more efficient way.

Event demultiplexing

Busy-waiting is definitely not an ideal technique for processing non-blocking resources, but luckily, most modern operating systems provide a native mechanism to handle concurrent non-blocking resources in an efficient way. We are talking about the **synchronous event demultiplexer** (also known as the **event notification interface**).



If you are unfamiliar with the term, in telecommunications, **multiplexing** refers to the method by which multiple signals are combined into one so that they can be easily transmitted over a medium with limited capacity.

Demultiplexing refers to the opposite operation, whereby the signal is split again into its original components. Both terms are used in other areas (for example, video processing) to describe the general operation of combining different things into one, and vice versa.

This type of event demultiplexer monitors multiple resources and generates an event (or a set of events) when a read or write operation on one of those resources completes. The advantage is that the synchronous event demultiplexer blocks until there are new events to process. The following pseudocode demonstrates an algorithm using a generic synchronous event demultiplexer to read from two different resources:

```
watchList.add(socketA, FOR_READ) // 1
watchList.add(fileB, FOR_READ)
while (events = demultiplexer.watch(watchList)) { // 2
  // event loop
  for (event of events) { // 3
    // This read will never block and will always return data
    data = event.resource.read()
    if (data === RESOURCE_CLOSED) {
      // the resource was closed, remove it from the watched list
      demultiplexer.unwatch(event.resource)
    } else {
      // some actual data was received, process it
      consumeData(data)
    }
  }
}
```

Let's see what happens in the preceding pseudocode:

1. The resources are added to a data structure, associating each one of them with a specific operation (in our example, a read).
2. The demultiplexer is set up with the group of resources to be watched. The call to `demultiplexer.watch()` is synchronous and blocks until any of the watched resources are ready for read. When this occurs, the event demultiplexer returns from the call and a new set of events is available to be processed.
3. Each event returned by the event demultiplexer is processed. At this point, the resource associated with each event is guaranteed to be ready to read and not block during the operation. When all the events are processed, the flow will block again on the event demultiplexer until new events are again available to be processed. This is called the **event loop**.

It's interesting to see that, with this pattern, we can now handle several I/O operations inside a single thread, without using the busy-waiting technique. It should now be clear why we are talking about demultiplexing; using just a single thread, we can deal with multiple resources. *Figure 1.2* will help you visualize what's happening in a web server that uses a synchronous event demultiplexer and a single thread to handle multiple concurrent connections:

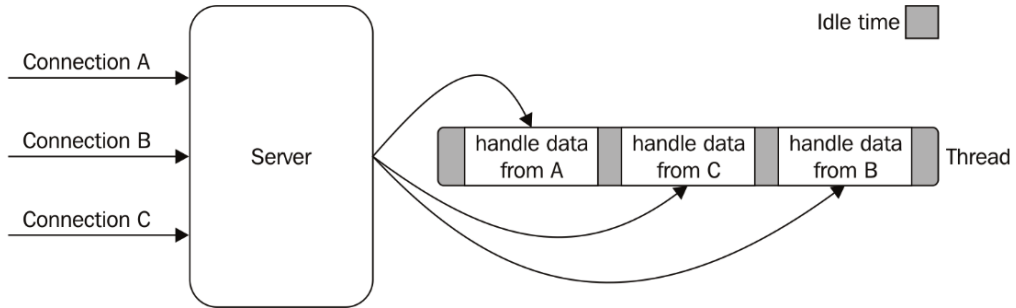


Figure 1.2: Using a single thread to process multiple connections

As this shows, using only one thread does not impair our ability to run multiple I/O-bound tasks concurrently. The tasks are spread over time, instead of being spread across multiple threads. This has the clear advantage of minimizing the total idle time of the thread, as is clearly shown in *Figure 1.2*.

But this is not the only reason for choosing this I/O model. In fact, having a single thread also has a beneficial impact on the way programmers approach concurrency in general. Throughout the book, you will see how the absence of in-process race conditions and multiple threads to synchronize allows us to use much simpler concurrency strategies.

The reactor pattern

We can now introduce the reactor pattern, which is a variation of the algorithms presented in the previous sections and what Node.js utilizes under the hood. The main idea behind the reactor pattern is to have a handler associated with each I/O operation. A handler in Node.js is represented by a callback (or *cb* for short) function.

The handler will be invoked as soon as an event is produced and processed by the event loop. The structure of the reactor pattern is shown in *Figure 1.3*:

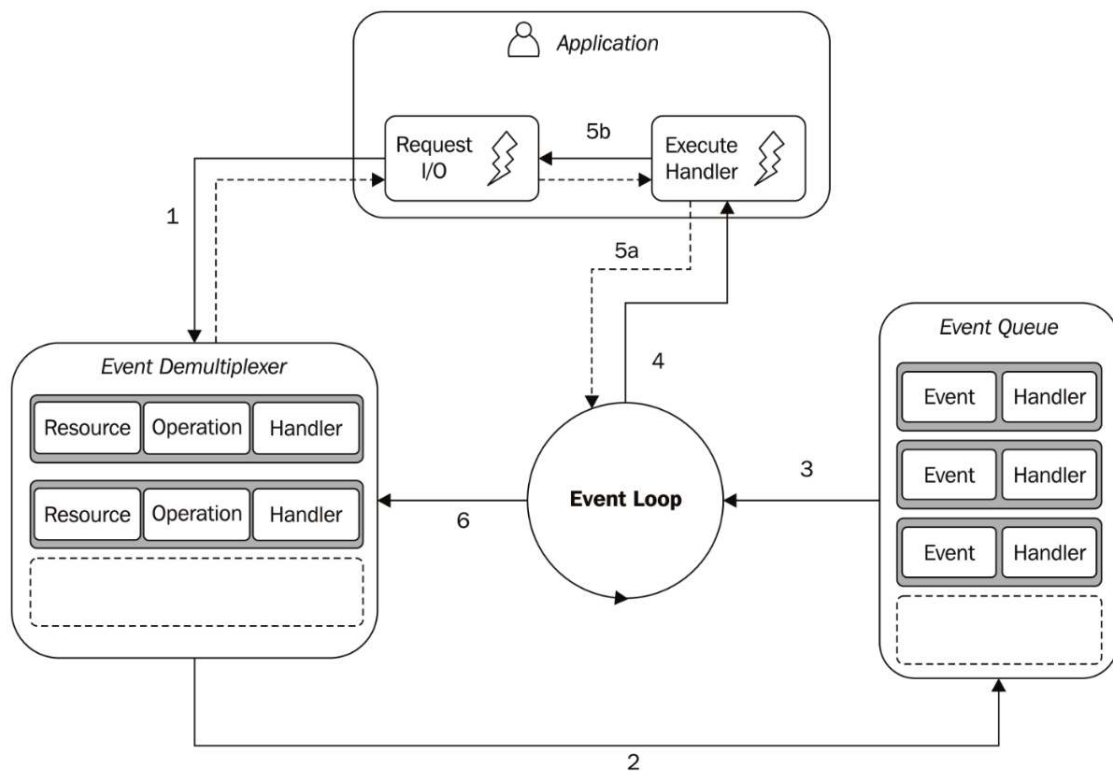


Figure 1.3: The reactor pattern

 **Quick tip:** Need to see a high-resolution version of this image? Open this book in the next-gen Packt Reader or view it in the PDF/ePub copy.

 **The next-gen Packt Reader** and a **free PDF/ePub copy** of this book are included with your purchase. Visit <https://packtpub.com/unlock>, then use the search bar to find this book by name. Double-check the edition shown to make sure you get the right one.

This is what happens in an application using the reactor pattern:

1. The application generates a new I/O operation by submitting a request to the **Event Demultiplexer**. The application also specifies a handler, which will be invoked when the operation completes. Submitting a new request to the **Event Demultiplexer** is a non-blocking call and it immediately returns control to the application.
2. When a set of I/O operations completes, the **Event Demultiplexer** pushes a set of corresponding events into the **Event Queue**.
3. At this point, the **Event Loop** iterates over the items of the **Event Queue**.

4. For each event, the associated handler is invoked.
5. The handler, which is part of the application code, gives back control to the **Event Loop** when its execution completes (5a). While the handler executes, it can request new asynchronous operations (5b), causing new items to be added to the **Event Demultiplexer** (1).
6. When all the items in the **Event Queue** are processed, the **Event Loop** blocks again on the **Event Demultiplexer**, which then triggers another cycle when a new event is available.

The asynchronous behavior has now become clear. The application expresses interest in accessing a resource at one point in time (without blocking) and provides a handler, which will then be invoked at another point in time when the operation completes.



A Node.js application exits when the event loop determines that there are no pending operations or events left to handle. Specifically, when there are no more active handles or requests (such as timers, open sockets, or filesystem operations) within the event demultiplexer, the event loop stops. However, certain resources like an active HTTP server, open network sockets, or pending I/O operations keep the event loop alive by continuously registering events. For example, in the case of an HTTP server, the `http.createServer()` call creates a server that listens on a specified port. This listening socket is considered an active handle by the event loop. As long as the server is listening, Node.js maintains this socket to accept incoming connections, which keeps the process running. The event loop will only exit when this socket is explicitly closed, and no other active handles remain in the system.

We can finally define the reactor pattern, the pattern at the heart of Node.js: it handles I/O by blocking until new events are available from a set of observed resources and then reacts by dispatching each event to an associated handler.



The reactor pattern should not be confused with the **proactor pattern**, even though both aim to manage multiple I/O operations concurrently. While they share a similar goal, they differ significantly in how they handle event processing. In the reactor pattern, the application has more control over when and how I/O operations are performed, as it responds to signals indicating that I/O is ready. In contrast, the proactor pattern abstracts the entire I/O process, notifying the application only once the operation is complete. Since Node.js uses the reactor pattern, we won't be diving into the proactor approach here. However, if you're interested, you can learn more by visiting nodejsdp.link/proactor.

libuv, the I/O engine of Node.js

Each operating system has its own interface for the event demultiplexer: `epoll` on Linux, `kqueue` on macOS, and the I/O completion port (IOCP) API on Windows. On top of that, each I/O operation can behave quite differently depending on the type of resource, even within the same operating system. In Unix operating systems, for example, regular filesystem files do not support non-blocking operations, so in order to simulate non-blocking behavior, it is necessary to use a separate thread outside the event loop.

All these inconsistencies across and within the different operating systems required a higher-level abstraction to be built for the event demultiplexer. This is exactly why the Node.js core team created a native library called **libuv**, with the objective of making Node.js compatible with all the major operating systems and normalizing the non-blocking behavior of the different types of resources. libuv represents the low-level I/O engine of Node.js and is probably the most important component that Node.js is built on.

Other than abstracting the underlying system calls, libuv also implements the reactor pattern, thus providing an API for creating event loops, managing the event queue, running asynchronous I/O operations, and queuing other types of tasks.



A great resource to learn more about libuv is the free online book created by Nikhil Marathe, which is available at nodejsdp.link/uvbook.

The complete recipe for Node.js

With our understanding of the reactor pattern and libuv, we've uncovered some of the key components that make Node.js work. However, to complete the full recipe for building the Node.js platform, we need three more crucial ingredients:

- A set of bindings responsible for wrapping and exposing libuv and other low-level functionalities to JavaScript.
- **V8**, the JavaScript engine originally developed by Google for the Chrome browser. This is one of the reasons why Node.js is so fast and efficient. V8 is acclaimed for its revolutionary design, its speed, and its efficient memory management.
- A core JavaScript library that implements the high-level Node.js API.

This is the recipe for creating Node.js, and the following image represents its final architecture:

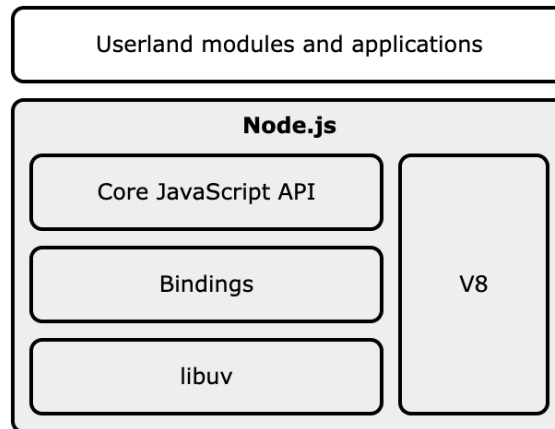


Figure 1.4: The Node.js internal components

This concludes our journey through the internal mechanisms of Node.js. Next, we'll take a look at some important aspects to take into consideration when working with JavaScript in Node.js.

JavaScript in Node.js

One important consequence of the architecture we have just analyzed is that the JavaScript we use in Node.js is somewhat different from the JavaScript we use in the browser.

The major difference between running JavaScript in the browser (client-side) and in Node.js (server-side) lies in their execution environments. In the browser, JavaScript runs when a user visits a web page, requiring a secure environment to prevent unrestricted access to the user's system and potential vulnerabilities. On the server side, however, we have more control and typically need access to databases, the filesystem, the network, and other system resources to build functional applications. As a result, Node.js has access to all the services provided by the operating system.

Although both Node.js and the browser can execute JavaScript, their different use cases and security requirements lead to significantly different APIs. The most important difference is that, in Node.js, we don't have a DOM, and there is no window or document object.

In this overview, we'll take a look at some key facts to keep in mind when using JavaScript in Node.js.

Run the latest JavaScript with confidence

One of the main challenges of using JavaScript in the browser is that our code needs to run on various devices and browsers. Different browsers can have small differences and may not support the latest features of the language or the web platform. This used to be a big problem in the past. If you've ever had to support Internet Explorer 6, you know what we mean. Luckily, today's browsers are much better at following standards and they get updated frequently, which greatly reduces unexpected issues. If you want to use the newest features, you can often use *transpilers* and *polyfills* to make sure they work. However, these tools add complexity to your project, and not everything can be polyfilled.



A **transpiler** is a tool that converts code from one programming language or version to another, usually to make it compatible with older environments. For example, a transpiler can convert modern JavaScript (ES2025) syntax into the equivalent ES5 syntax so it runs in older browsers or older versions of Node.js.

A **polyfill** is a piece of code (typically a library or function) that adds functionality to older environments that don't natively support newer features. It mimics the behavior of new features, allowing developers to use them without breaking compatibility.

All these challenges don't apply when developing applications with Node.js. Our Node.js applications usually run on a system and a Node.js runtime that are known in advance. This makes a big difference, as it allows us to write code for specific JavaScript and Node.js versions, ensuring there won't be any surprises when we run it on production machines.

This, combined with the fact that newer versions of Node.js come with recent versions of V8, means we can confidently use most of the latest **ECMAScript (ES)** features without needing any extra transpilation steps. (ES is the standard on which the JavaScript language is based.)

Bear in mind, however, that if we are developing a library meant for third-party use, we still need to consider that our code may run on different versions of Node.js. The common practice, in this case, is to target the oldest active **long-term support (LTS)** release and specify the engines section in our `package.json`. This way, the package manager will warn users if they try to install a package that isn't compatible with their version of Node.js.



You can find out more about the Node.js release cycles at nodejsdp.link/node-releases. Also, you can find the reference for the engines section of `package.json` at nodejsdp.link/package-engines. Finally, you can get an idea of what ES feature is supported by each Node.js version at nodejsdp.link/node-green.

The module system

From its inception, Node.js included a module system, even when JavaScript had no official support for one. The original Node.js module system is called **CommonJS**, and it uses the `require` keyword to import functions, variables, and classes from built-in modules or other modules on the device's filesystem.

CommonJS was revolutionary for the JavaScript world, gaining popularity even on the client side, where it is used with module bundlers like Webpack or Rollup to create code bundles easily executable by the browser. CommonJS was essential for Node.js, allowing developers to create large, well-organized applications comparable to those on other server-side platforms.



A **module bundler** is a tool that combines multiple JavaScript files and their dependencies into a single or smaller number of files, often called bundles. This process helps optimize loading times and reduces the complexity of managing dependencies. Module bundlers analyze the relationships between various modules, resolve dependencies, and output them into a format that can be efficiently executed in a browser or other runtime environments.

Today, JavaScript has the ES modules syntax (using the `import` keyword), which Node.js adopts in syntax only, as its underlying implementation differs from the browser's. While browsers mainly handle remote modules, Node.js, at least for now, deals only with modules on the local filesystem.

We'll discuss modules in the context of Node.js in more detail in the next chapter.

Full access to operating system services

As we already mentioned, even if Node.js uses JavaScript, it doesn't run inside the boundaries of a browser. This allows Node.js to have bindings for all the major services offered by the underlying operating system.

For example, we can access any file on the filesystem (subject to any operating system-level permission) thanks to the `fs` module, or we can write applications that use low-level TCP or UDP sockets thanks to the `net` and `dgram` modules. We can create HTTP(S) servers (with the `http` and `https` modules) or use the standard encryption and hashing algorithms of OpenSSL (with the `crypto` module). We can also access some of the V8 internals (the `v8` module) or run code in a different V8 context (with the `vm` module).

We can also run other processes (with the `child_process` module) or retrieve our own application's process information using the `process` global variable. In particular, from the `process` global variable, we can get a list of the environment variables assigned to the process (with `process.env`) or the command-line arguments passed to the application at the moment of its launch (with `process.argv`).

Throughout the book, you'll have the opportunity to use many of the modules described here, but for a complete reference, you can check the official Node.js documentation at nodejsdp.link/node-docs.

Running native code

One of the most powerful capabilities offered by Node.js is certainly the possibility to create userland modules that can bind to native compiled code (e.g., written in compiled languages such as C, C++, or Rust). This gives the platform a tremendous advantage as it allows us to reuse existing or use new components written in performant compiled languages such as C/C++. Node.js officially provides great support for implementing native modules thanks to the Node-API interface.

But what's the advantage? First of all, it allows us to reuse, with little effort, a vast amount of existing open-source libraries, and most importantly, it allows a company to reuse its own C/C++ legacy code without the need to migrate it.

Another important consideration is that native code is still necessary to access low-level features such as communicating with hardware drivers or with hardware ports (for example, USB or serial). In fact, thanks to its ability to link to native code, Node.js has become popular in the world of the **Internet of Things (IoT)** and homemade robotics.

Finally, even though V8 is very (very) fast at executing JavaScript, it still has a performance penalty to pay compared to executing native code. In everyday computing, this is rarely an issue, but for CPU-intensive applications, such as those with a lot of data processing and manipulation, delegating the work to native code can make a lot of sense.

We should also mention that, nowadays, most JavaScript **virtual machines (VMs)** (and also Node.js) support **WebAssembly (Wasm)**, a low-level instruction format that allows us to compile languages other than JavaScript (such as C++ or Rust) into a format that is “executable” by JavaScript VMs. This brings many of the advantages we have mentioned, without the need to directly interface with native code.



You can learn more about Wasm on the official website of the project at nodejsdp.link/webassembly.

Node.js and TypeScript

TypeScript is an open-source language developed by Microsoft to add a strong static type system to JavaScript. Think of TypeScript as an enhanced version of JavaScript with extra features. For instance, with TypeScript, developers can specify the types of arguments a function accepts, the type it returns, or the structure of an object. These types are checked before your code runs, helping to catch issues like accessing non-existent properties or passing incorrect parameters. This makes your code more secure and robust, preventing many bugs before the code even goes live.

It's important to know that TypeScript code can't run directly on JavaScript platforms like Node.js. Instead, TypeScript is used for static analysis and must be compiled (or “transpiled”) into plain JavaScript to run. This process converts TypeScript into JavaScript files that can work in an environment supporting JavaScript. While this extra step might seem like more work, the advantages of catching errors early and enhancing code quality make it highly beneficial, especially for larger projects.

Using TypeScript with Node.js

If you want to use TypeScript with Node.js, you have a few options available.

The first option is to use the official TypeScript compiler to convert your TypeScript files into equivalent JavaScript files that can be executed by Node.js. To install TypeScript in your project, run:

```
npm install --save-dev typescript
```

If you have a TypeScript file called `example.ts`, for example, you can transpile it to JavaScript with the following command:

```
npx tsc example.ts
```

This command will first check `example.ts` for any type error and, if everything is fine, it will convert it to plain JavaScript, creating a file called `example.js`. You can then execute `example.js` with Node.js:

```
node example.js
```

When developing an application, it can be tedious to manually transpile your code every time you want to run it. Fortunately, there are tools that handle this for you automatically, such as `ts-node` and `tsx`.

To install `ts-node` or `tsx`, run:

```
npm install --save-dev ts-node  
# or  
npm install --save-dev tsx
```

Then you can directly execute `example.ts` with:

```
npx ts-node example.ts  
# or  
npx tsx example.ts
```

Additionally, `tsx` can also be used as a *Node.js loader*, which can be convenient in case you want to invoke the node CLI directly (e.g., if you need to pass additional CLI options arguments while executing your code):

```
node --import=tsx example.ts
```



Loaders are a mechanism that allows developers to customize how modules are loaded and processed in Node.js. By default, Node.js uses its built-in module loading system (CommonJS or ES modules), but with custom loaders, you can intercept and modify the original behavior. By using a custom TypeScript loader like `tsx`, you can intercept TypeScript modules during the loading process, transpile them into JavaScript, and then pass the resulting code to Node.js for execution. This eliminates the need for a separate build step, making development workflows smoother and more flexible.



Keep in mind that when using tools like `ts-node` and `tsx` to run TypeScript code directly, the code is being transpiled “on the fly.” This means you incur the time cost of transpilation each time you run your code. While this approach is convenient for development, it’s more efficient to pre-transpile your code before deploying it to production. This avoids the overhead of transpilation during runtime and ensures better performance.

The Node.js core team is putting a lot of effort into making TypeScript a first-class citizen of the platform, so we can expect that it is going to become easier to execute TypeScript code in the future without having to install third-party tools.

In fact, as of Node.js 24 you can execute TypeScript files directly with the node CLI via built in type stripping. Only erasable TypeScript syntax is supported and Node ignores `tsconfig.json`, so your mileage might vary and a runner is still best for full features; for the latest guidance see nodejsdp.link/node-ts.

The `@types/node` package

When developing TypeScript applications in a Node.js environment, you should use the `@types/node` package for a smooth development experience. This package provides TypeScript with the necessary type definitions for Node.js, enabling strong typing and enhanced autocompletion within your IDE. Node.js itself is written in JavaScript, which does not inherently include type definitions. Therefore, without `@types/node`, TypeScript would lack the knowledge of Node.js-specific globals, modules, and APIs, making it difficult to write type-safe code.

The `@types/node` package includes type definitions for the entire Node.js API, covering everything from core modules like `fs`, `http`, and `path` to global objects like `process` and `Buffer`. By incorporating this package into your project, you gain access to TypeScript’s powerful static type-checking features, which can help catch potential bugs early in the development process. Moreover, it provides comprehensive autocompletion in your code editor, making development faster and reducing the likelihood of errors caused by incorrect method usage or misconfigured parameters.

To install the `@types/node` package, you can use npm:

```
npm install --save-dev @types/node
```

These commands add the `@types/node` package as a development dependency (i.e., it will be installed only during development but not in production environments).



TypeScript has a lot to offer, but it's not the main focus of this book. We'll provide TypeScript examples and tips when they help clarify specific topics, but for a comprehensive introduction to TypeScript, we recommend visiting the official TypeScript website at nodejsdp.link/ts. If you're already familiar with TypeScript, you'll still find this book valuable. The patterns and techniques we discuss can be easily applied to any TypeScript project.

Summary

In this chapter, you have seen how the Node.js platform is built upon a few important principles that shape both its internal architecture and the code we write. You have learned that Node.js has a minimal core and that embracing the “Node way” means writing modules that are smaller, simpler, and that expose only the minimum functionality necessary.

Next, you discovered the reactor pattern, which is the pulsating heart of Node.js, and dissected the internal architecture of the platform runtime to reveal its other pillars: V8, libuv, bindings, and the core JavaScript library.

Finally, we analyzed some of the main characteristics of using JavaScript in Node.js compared to the browser and learned how TypeScript can be leveraged when working with Node.js.

Besides the obvious technical advantages enabled by its internal architecture, Node.js draws significant interest due to the principles it embodies and the vibrant community surrounding it. Its focus on simplicity and efficiency resonates with developers, offering a more human-centered approach to programming that balances ease of use with scalability. This is why so many developers find themselves falling in love with Node.js.

In the next chapter, we will go deep into one of the most fundamental and important topics of Node.js, its module system.

2

The Module System

In *Chapter 1, The Node.js Platform*, we briefly introduced the importance of modules in Node.js. We discussed how modules play a fundamental role in defining some of the pillars of the Node.js philosophy and its programming experience. However, what do we mean when we talk about modules and why are they so important?

In generic terms, modules are the bricks for structuring non-trivial applications. Modules allow you to divide the code base into small units that can be developed and tested independently. Modules are also the main mechanism to enforce information hiding by keeping all the functions and variables that are not explicitly marked to be exported private. Additionally, modules make it easier to share and reuse code across projects.

Due to historical reasons, Node.js has two different module systems: **ECMAScript modules (ES modules)** and **CommonJS**. Since ES modules are now the main module system in Node.js and are widely used on the web, we will focus on them in this chapter. We will briefly discuss CommonJS, as it is still found in older code bases and libraries, and it's useful to understand how to work with both module systems if needed. By the end of this chapter, you should be able to make informed decisions on how to use and write modules effectively.

Getting a good grasp of Node.js' module systems and module patterns is very important as we will rely on this knowledge in all the other chapters of this book.

In short, these are the main topics we will be discussing throughout this chapter:

- Why modules are necessary
- The different module systems available in Node.js
- The revealing module pattern
- ES modules in Node.js
- CommonJS modules
- CommonJS and interoperability with ES modules
- How to use modules in TypeScript

Let's begin with why we need modules.

The need for modules

Let's start by clarifying the difference between a **module** and a **module system**. A module is the actual piece of software, while a module system is the syntax and tools that let us define and use modules in our projects.

No matter the programming language or the platform, a good module system should help with some fundamental needs of software engineering:

- **Organizing code:** It should allow the code base to be split into multiple files. This keeps the code more organized and easier to understand, and enables independent development and testing of different parts.
- **Code reuse and dependency management:** A good module system should make it easy to share and reuse functionality across projects while handling dependencies in a straightforward way. Developers should be able to build on existing modules, including third-party ones, and users should be able to import a module together with everything it requires without friction.
- **Encapsulation and information hiding:** It should help hide implementation details and expose only a clear, simple interface. Most module systems allow certain parts of the code to remain *private* while providing *public* functions, classes, or objects for users of the module.

Module systems in JavaScript and Node.js

Not all programming languages come with a built-in module system, and JavaScript lacked this feature for a long time after its inception.

When writing JavaScript code for the browser, it is possible to split the code base into multiple files and then import them by using different `<script>` tags. For many years, this approach was good enough to build simple interactive websites, and JavaScript developers managed to get things done without having a fully-fledged module system.

Only when JavaScript browser applications became more advanced and frameworks like **jQuery**, **Backbone**, and **AngularJS** took over the ecosystem did the JavaScript community come up with several initiatives aimed at defining a module system that could be effectively adopted within JavaScript projects. The most successful ones were **asynchronous module definition (AMD)**, popularized by **RequireJS** (nodejsdp.link/requirejs), and later **universal module definition (UMD)**—nodejsdp.link/umd).

When Node.js was created, it was conceived as a server runtime for JavaScript with direct access to the underlying filesystem so there was a unique opportunity to introduce a different way to manage modules. The idea was not to rely on HTML `<script>` tags and resources accessible through URLs. Instead, the choice was to rely purely on JavaScript files available on the local filesystem. For its module system, Node.js came up with an implementation of the CommonJS specification (nodejsdp.link/commonjs), which was designed to provide a module system for JavaScript in *browserless* environments. It's worth noting that CommonJS was not part of the official ECMAScript standard, but an independent initiative to standardize JavaScript outside the browser.

CommonJS has been the dominant module system in Node.js for many years. It became so popular that people started using it to write modules that could be used even in browser applications thanks to **module bundlers** like **Browserify** (nodejsdp.link/browserify) and **Webpack** (nodejsdp.link/webpack).

In 2015, with the release of **ECMAScript 2015 (ES2015)**, there was an official proposal to define a standard module system for JavaScript: **ES modules**. ES modules bring a lot of innovation in the JavaScript ecosystem, and, among other things, they try to bridge the gap between how modules are managed on browsers and servers.

ES2015 defined only the formal specification for ES modules in terms of syntax and semantics, but it didn't provide any implementation details. It took several years for browser companies and the Node.js community to develop strong implementations, with Node.js offering stable support for ES modules starting from version 13.2 (released in 2019). As a result, the transition from CommonJS to ES modules has been somewhat slow and, during the transition years, developers have been using various techniques to publish *dual-mode* libraries that can work with both ES modules and CommonJS.

Today, ES modules are the widely accepted standard module system for JavaScript and Node.js. While CommonJS is still common in legacy code bases and older libraries, most new projects are now written using ESM, and CommonJS usage is expected to decline over time. This book uses ES modules for all code examples, but we will also examine some CommonJS code to understand how the two module systems can work together.

The revealing module pattern

Before diving straight into ES modules, it's worth taking a brief detour to explore a foundational JavaScript pattern, the **revealing module pattern**, a pattern that facilitates information hiding and will be useful in building a primitive module system. This background will not only deepen our appreciation of fully-fledged module systems like ES modules and CommonJS but also provide a great opportunity to see how these module systems can be implemented behind the scenes. As such, this pattern will become a foundational piece of knowledge that will help us to deeply understand ES modules.

As we said, modules are the bricks for structuring non-trivial applications and the main mechanism to enforce information hiding by keeping all the functions and variables that are not explicitly marked to be exported private.

One major issue with JavaScript in the browser is the lack of namespacing. Since every script runs in the global scope, both internal application code and third-party dependencies can clutter the scope by exposing their own functions and variables. This can be very problematic. For example, if a third-party library creates a global variable called `utils` and another library or the application code itself accidentally overwrites or changes `utils`, the code that depends on it might crash in unpredictable ways. Similar issues can arise if other libraries or the application code accidentally call a function that was intended only for internal use by another library.

Relying on the global scope is risky, especially as your application grows and you depend more on code written by others.

A popular technique to solve this class of problems is called the *revealing module pattern*, and it looks like this:

```
const myModule = (() => {
  const privateFoo = () => {}
  const privateBar = []

  console.log('Inside:', privateFoo, privateBar)

  const exported = {
    publicFoo: () => {},
    publicBar: () => {},
  }

  return exported
})(); // once the parenthesis here are parsed
      // the function will be invoked
      // and the returned value assigned to myModule

console.log('Outside:', myModule.privateFoo, myModule.privateBar)
console.log('Module:', myModule)
```

This pattern leverages a self-invoking function. This type of function is sometimes also referred to as an **immediately invoked function expression (IIFE)** and it is used to create a private scope, exporting only the parts that are meant to be public.

In JavaScript, variables created inside a function are not accessible from the outer scope (outside the function). Functions can use the return statement to selectively propagate information to the outer scope.

This pattern essentially exploits these properties to keep the private information hidden and export only a public-facing API.

In the preceding code, the `myModule` variable contains only the exported API, while the rest of the module content is practically inaccessible from the outside.

The log statement is going to print something like this:

```
Inside: [Function: privateFoo] []
Outside: undefined undefined
Module: { publicFoo: [Function: publicFoo], publicBar: [Function: publicBar] }
```

This demonstrates a few important details:

- The private `privateFoo` and `privateBar` variables are accessible from within the immediately invoked function.
- Printing `myModule` doesn't show `privateFoo` and `privateBar` among the members of the object and we have no way to access these values from outside the immediately invoked function. If we try to access `myModule.privateFoo` and `myModule.privateBar`, these values are undefined.
- Only the exported properties `publicFoo` and `publicBar` are directly accessible from `myModule`.

The revealing module pattern can be used to create a basic module system, and this concept is actually the foundation of the CommonJS module system. In this chapter, the revealing module pattern helps demonstrate how certain intrinsic features of JavaScript can be used to structure code modularly and implement information hiding. However, in practice, you're not expected to build your own module system from scratch; instead, you'll be using ES modules.

ES modules

ES modules were introduced as part of the ES2015 specification with the goal of giving JavaScript an official module system suitable for different execution environments. The ES modules specification tries to retain some good ideas from previous existing module systems like CommonJS and AMD. The syntax is very simple and compact. There is support for cyclic dependencies and the possibility to load modules asynchronously.



A cyclic dependency in a module system occurs when two or more modules depend on each other in a circular manner. For example, Module A imports from Module B, and Module B, in turn, imports from Module A. This creates a loop in dependencies that can lead to issues like incomplete module loading, runtime errors, or unexpected behavior, as the modules are unable to fully resolve due to their mutual reliance.

Using ES modules in Node.js

As previously discussed, when ES modules were introduced in Node.js, CommonJS had already long been the default module system. As a result, support for ES modules in Node.js had to be added carefully to maintain backward compatibility. This means that ES modules are not the default and must be explicitly enabled. To adopt ES modules in a project, there are specific steps that signal to Node.js that a file or module should use the ES module syntax, making it an opt-in feature for developers.

Node.js will consider every `.js` file to be written using the CommonJS syntax by default. For example, let's say that we have a file called `index.js` with the following content:

```
import { someFeature } from './someModule.js'
console.log(someFeature)
```

We could then try to execute it with the following:

```
node index.js
```

You might see an error that looks like the following:

```
(node:69441) Warning: To load an ES module, set "type": "module" in the
package.json or use the .mjs extension.
(Use `node --trace-warnings ...` to show where the warning was created)
index.js:1
import { someFeature } from './someModule.js'
^^^^^^

SyntaxError: Cannot use import statement outside a module
```

This message means that our `index.js` file isn't recognized as an ES module, so we can't use ES module syntax. Node.js also provides helpful suggestions on how to fix this: we need to tell Node.js to load this file as an ES module. There are a few ways we can do that:

- Give the module file the extension `.mjs` (note that, alternatively, you can use `.cjs` to force the module to be interpreted as a CommonJS module).
- Add a field called `"type"` with a value of `"module"` to the nearest parent `package.json` as in the following example:

```
{
  "name": "sample-esm-project",
  "version": "1.0.0",
  "main": "index.js",
  "type": "module"
}
```

- Use the flag `--experimental-default-type="module"`. This flag sets the default module system to ES modules, making it equivalent to adding `"type": "module"` in your `package.json`. However, you'll need to specify this flag explicitly each time you run `node` to execute an ES module.
- Use the flag `--experimental-detect-module`. This flag instructs Node.js to analyze ambiguous files (such as those with a `.js` extension) when the module system is not explicitly specified. Node.js will attempt to infer the module type by scanning the file for keywords like `import` and `export`, which typically indicate an ES module rather than CommonJS.



Our current recommendation is to use `"type": "module"` in the `package.json`, so you can keep using the `.js` extension which is the most commonly supported across text editors. Chances are that in the future Node.js will default to ES modules or that it will automatically detect the correct module system to use to load your files (at the time of writing, Node.js 23 has adopted module detection by default, so this is likely to become the default behavior in the future). Throughout this book, we will be using our recommended approach for most of our examples, so if you are copying and pasting examples straight from the book, make sure that you also create a `package.json` file with the `"type": "module"` entry.

Let's now have a look at the ES module syntax.

The ES module syntax

In this section, we'll focus specifically on the syntax of ES modules, covering its core elements like named and default exports, mixed exports, and module identifiers. We'll also look at how ES module syntax supports both static imports (loaded at the start of execution) and dynamic imports, which can be loaded conditionally or asynchronously.

Named exports and imports

ES modules allow us to export constants, functions, and classes from a module through the `export` keyword.

In an ES module, everything is private by default and only exported entities are publicly accessible from other modules.

The `export` keyword can be used in front of the entities that we want to make available to the module users. Let's see an example:

```
// Logger.js

// exports a function as `log`
export function log(message) {
  console.log(message)
}

// exports a constant as `DEFAULT_LEVEL`
export const DEFAULT_LEVEL = 'info'

// exports an object as `LEVELS`
export const LEVELS = {
  error: 0,
  debug: 1,
  warn: 2,
  data: 3,
  info: 4,
  verbose: 5
}

// exports a class as `Logger`
export class Logger {
  constructor(name) {
    this.name = name
  }
}
```



```
log(message) {  
  console.log(`[${this.name}] ${message}`)  
}  
}
```

If we want to import entities from a module, we can use the `import` keyword. The syntax is quite flexible, and it allows us to import one or more entities and even rename imports. Let's see some examples:

```
import * as loggerModule from './logger.js'  
console.log(loggerModule)
```

In this example, we are using the `*` syntax (also called **namespace import**) to import all the members of the module and assign them to the local `loggerModule` variable. This example will output something like this:

```
[Module] {  
  DEFAULT_LEVEL: 'info',  
  LEVELS: { error: 0, debug: 1, warn: 2, data: 3, info: 4,  
    verbose: 5 },  
  Logger: [Function: Logger],  
  log: [Function: log]  
}
```

As we can see, all the entities exported in our module are now accessible in the `loggerModule` namespace. For instance, we could refer to the `log()` function through `loggerModule.log`.

It's generally better to be explicit and to avoid importing an entire module and instead import only the specific entities that are needed in the current context:

```
import { log } from './logger.js'  
log('Hello World')
```

If we want to import more than one entity, this is how we would do that:

```
import { Logger, log } from './logger.js'  
log('Hello World')  
const logger = new Logger('DEFAULT')  
logger.log('Hello world')
```

When we use this type of `import` statement, the entities are imported into the current scope, so there is a risk of a name clash. The following code, for example, would not work:

```
import { log } from './logger.js'  
const log = console.log
```

If we try to execute the preceding snippet, the interpreter fails with the following error:

```
SyntaxError: Identifier 'log' has already been declared
```

In situations like this one, we can resolve the clash by renaming the imported entity with the `as` keyword:

```
import { log as log2 } from './logger.js'
const log = console.log

log('message from log')
log2('message from log2')
```

This approach can be particularly useful when the clash is generated by importing two entities with the same name from different modules, and therefore, changing the original names is outside the consumer's control. It can also be useful when you want to use a shorter or cleaner name for the entities you are importing.



Note that if you try to import a module member that is not exported, this will result in a syntax error at runtime. For example, we might try to execute the following code:

```
import { something } from './logger.js'
```

We will then get the following error:

```
SyntaxError: The requested module './logger.js' does not provide
an export named 'something'
```

Default exports and imports

Sometimes, you might want to export a single unnamed entity. With ES modules, we can do that with a **default export**. A default export makes use of the `export default` keywords and it looks like this:

```
// Logger.js
export default class Logger {
  constructor(name) {
    this.name = name
  }

  log(message) {
    console.log(`[${this.name}] ${message}`)
  }
}
```

In this case, the name `Logger` is ignored, and the entity exported is registered under the name `default`. This exported name can be imported as follows:

```
// main.js
import MyLogger from './logger.js'
const logger = new MyLogger('info')
logger.log('Hello World')
```

The difference with named ES module imports is that here, since the default export is considered unnamed, we can import it and at the same time assign it a local name of our choice. In this example, we can replace `MyLogger` with anything else that makes sense in our context. Note also that we don't have to wrap the import name around brackets or use the `as` keyword when renaming.

Internally, a default export is equivalent to a named export with `default` as the name. We can easily verify this statement by running the following snippet of code:

```
// showDefault.js
import * as loggerModule from './logger.js'
console.log(loggerModule.default)
```

When executed, the previous code will print something like this:

```
[class Logger]
```

One thing that we cannot do, though, is import the default entity explicitly. In fact, something like the following will fail:

```
import { default } from './logger.js'
```

The execution will fail with a `SyntaxError: Unexpected reserved word` error. This happens because the `default` keyword cannot be used as a variable name. It is valid as an object attribute, so in the previous example, it is okay to use `loggerModule.default`, but we can't have a variable named `default` directly in the scope.

Mixed exports

It is possible to mix named exports and a default export within an ES module. Let's have a look at an example:

```
// logger.js
export default function log(message) {
  console.log(message)
}

export function info(message) {
  log(`info: ${message}`)
}
```

The preceding code is exporting the `log()` function as a default export and a named export for a function called `info()`.



Note that `info()` can reference `log()` internally. It would not be possible to replace the call to `log()` with `default()` to do that, as it would be a syntax error (unexpected token `default`).

If we want to import both the default export and one or more named exports, we can do it using the following format:

```
import mylog, { info } from './logger.js'
```

In the previous example, we are importing the default export from `logger.js` (as `mylog`) and the named export `info`.



Note that, with this syntax, the default member should always come first. Writing the following code would result in a syntax error:

```
import { info }, mylog from './logger.js'
```

Let's now discuss some key details and differences between the default export and named exports:

- The default export is a convenient mechanism to communicate what is the single most important functionality for a module. Also, from the perspective of the user, it can be easier to import the obvious piece of functionality without having to know the exact name of the binding.
- Named exports are explicit. Having predetermined names allows IDEs to support the developer with automatic imports, autocomplete, and refactoring tools. For instance, if we type `writeFileSync`, the editor might automatically add `import { writeFileSync } from 'node:fs'` at the beginning of the current file. Default exports, on the contrary, make all these things more complicated as a given functionality could have different names in different files, so it's harder to make inferences on which module might provide a given functionality based only on a given name.
- In some circumstances, default exports might make it harder to apply dead code elimination (tree shaking). For example, a module could provide only a default export, which is an object where all the functionality is exposed as properties of such an object. When we import this default object, most module bundlers will consider the entire object being used and they won't be able to eliminate any unused code from the exported functionality.

The drawbacks of using default exports in JavaScript are greater than the benefits, leading the community to generally favor avoiding them. Some code linters now include rules to detect and warn against the use of default exports.



Biome (nodejsdp.link/biome), a JavaScript linter and formatter, enables a linting rule to disable default exports by default. The documentation page for this rule provides some additional details on why this is to be considered good practice: nodejsdp.link/no-default-export

This is not a hard rule and there are notable exceptions to this suggestion. For instance, all Node.js core modules have both a default export and a number of named exports. Also, React (nodejsdp.link/react) uses mixed exports.

Consider carefully what the best approach for your specific module is and what you want the developer experience to be for the users of your module.

Module identifiers

Module identifiers (also called *module specifiers*) are the different types of values that we can use in our `import` statements to specify the location of the module we want to load.

So far, we have only seen relative paths, but there are several other possibilities and some nuances to keep in mind. Let's list all the possibilities:

- *Relative specifiers* like `./logger.js` or `../logger.js` are used to refer to a path relative to the location of the importing file.
- *Absolute specifiers* like `file:///opt/nodejs/config.js` or `/opt/nodejs/config.js` refer directly and explicitly to a full path.
- *Bare specifiers* are identifiers like `fastify` or `http`. They represent modules available in the `node_modules` folder and generally installed through a package manager (such as `npm`) or available as core Node.js modules.



To avoid ambiguity between core Node.js modules and third-party modules, Node.js supports an optional `node:` prefix (e.g., `node:http`). This is the recommended way to import Node.js core modules.

- *Deep import specifiers* like `fastify/lib/logger.js` refer to a path within a package in `node_modules` (`fastify`, in this case).

In browser environments, it is possible to import modules directly by specifying the module URL, for instance, `https://unpkg.com/lodash`. This feature is not supported by Node.js.

Static and dynamic imports

ES modules are *static*, which means that imports are described at the top level of every module and outside any control flow statement. Also, the name of the imported modules cannot be dynamically generated at runtime using expressions; only constant strings are allowed.

For instance, the following code wouldn't be valid when using ES modules:

```
if (condition) {  
  import module1 from 'module1'  
} else {  
  import module2 from 'module2'  
}
```

At first glance, these characteristics of ES modules might seem like unnecessary limitations, but having static imports opens up a number of interesting scenarios. For instance, static imports allow the static analysis of the dependency tree, which allows optimizations such as dead code elimination (tree shaking).

Does this mean that with ES modules, we can't create module identifiers at runtime or import modules conditionally? For example, what if we want to load a specific module—perhaps a heavy one—only when the user accesses the feature that needs it? Or what if we need to import a particular translation module based on the user's language, or a version of a module that depends on the user's operating system?

Fortunately, ES modules include a feature called *dynamic imports* (or *async imports*) that addresses these scenarios.

Async imports can be performed at runtime using the special `import()` operator.

The `import()` operator is syntactically equivalent to a function that takes a module identifier as an argument. It returns a promise that resolves to a module object.



We will learn more about promises in *Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await*, so don't worry too much about understanding all the nuances of the specific promise syntax for now.

The module identifier can be any module identifier supported by static imports as discussed in the previous section. Now, let's see how to use dynamic imports with a simple example.

We want to build a command line application that can print “Hello World” in different languages. In the future, we will probably want to support many more phrases and languages, so it makes sense to have one file with the translations of all the user-facing strings for each supported language.

Let's create some example modules for some of the languages we want to support, for example:

```
// strings-el.js
export const HELLO = 'Γεια σου κόσμο'
```

```
// strings-en.js
export const HELLO = 'Hello World'
```

```
// strings-es.js
export const HELLO = 'Hola mundo'
```

Now let's create the main script that takes a language code from the command line and prints “Hello World” in the selected language:

```
// main.js
const SUPPORTED_LANGUAGES = ['el', 'en', 'es', 'it', 'pl'] // 1
const selectedLanguage = process.argv[2] // 2

if (!selectedLanguage) { // 3
  console.error(
    `Please specify a language

    Usage: node ${process.argv[1]} <language_code>
    Supported languages: ${SUPPORTED_LANGUAGES.join(', ')}`
  )
  process.exit(1)
}

if (!SUPPORTED_LANGUAGES.includes(selectedLanguage)) { // 4
  console.error('The specified language is not supported')
  process.exit(1)
}

const translationModule = `./strings-${selectedLanguage}.js` // 5
const strings = await import(translationModule) // 6
console.log(strings.HELLO) // 7
```

The first part of the script is quite simple. What we do there is as follows:

1. Define a list of supported languages.
2. Read the selected language from the first argument passed in the command line.

3. Validate that the user has passed the argument and if not, provide a helpful error message.
4. Finally, we handle the case where the selected language is not supported.

The second part of the code is where we actually use dynamic imports:

5. First of all, we dynamically build the name of the module we want to import based on the selected language. Note that the module name needs to be a relative path to the current module file; that's why we are prepending `./` to the filename.
6. We use the `import()` operator to trigger the dynamic import of the module. This import happens asynchronously, returning a promise. We use `await` to wait for the promise to resolve and store the resolved value in the `strings` variable.
7. Now, we can access `strings.HELLO` and print its value to the console.

We can execute this script like this:

```
node main.js it
```

We should see `Ciao mondo` being printed to our console.

The module resolution algorithm

The term *dependency hell* describes a scenario where two or more dependencies in a program rely on a shared library but require different, incompatible versions of it. This is a common problem in many programming languages and is often difficult to resolve.

Node.js addresses this problem elegantly by loading different versions of a module based on where the module is loaded from. This capability is largely due to how Node.js package managers (such as npm or pnpm) organize an application's dependencies and the resolving algorithm that maps a module specifier to a file in the file system. For example, this algorithm translates an identifier such as `'fastify/lib/logger.js'` to a URL that represents a file that can be loaded from the filesystem such as `file:///Users/luciano/projects/example_web_server/node_modules/fastify/lib/logger.js`, allowing the file to be loaded correctly.

Node.js implements the ES module function `import.meta.resolve()`, which allows us to see how a given module specifier is resolved in the current context. Let's try to have a high-level overview of how the module resolution algorithm works. We can split the algorithm into the following three major branches:

- **File modules:** If the module specifier starts with `/`, it is considered an absolute path to the module file in the filesystem. The path is normalized and converted into a URL with a `file://` prefix. If it starts with `./` or `../`, then the module specifier is considered a relative path, which is calculated starting from the directory of the requiring module.
- **Node.js core modules:** If the module specifier is not prefixed with `/`, `./`, or `../`, the algorithm will first try to search within the core Node.js modules. If the module specifier matches one of the Node.js core modules, then the given specifier is prefixed with `node:` and returned. If the provided module specifier is already prefixed with `node:` then it is returned as is.

- **Package modules:** If no core module is found matching the given module specifier, then the search continues by looking for a matching module in the first `node_modules` directory that is found navigating up in the directory structure starting from the importing module. The algorithm continues to search for a match by looking into the next `node_modules` directory up in the directory tree until it reaches the root of the filesystem.



There are some additional types of specifiers supported such as the `data:` prefix. The complete, formal documentation of the resolving algorithm can be found at nodejsdp.link/resolve_esm.

We can see this algorithm in action with the following code:

```
// relative import
console.log(import.meta.resolve('./utils/example.js'))
// -> file:///<project_path>/utils/example.js

// Node.js core module import
console.log(import.meta.resolve('assert'))
// -> node:assert
console.log(import.meta.resolve('node:assert'))
// -> node:assert

// Third-party library import (with fastify@5.1.0 installed)
console.log(import.meta.resolve('fastify/lib/logger.js'))
// -> file:///<project_path>/node_modules/fastify/lib/logger.js
```

The `node_modules` directory is actually where the package managers install the dependencies of each package. This means that, based on the algorithm we just described, each package can have its own private dependencies. For example, consider the following directory structure:

```
myApp
├─ foo.js
└─ node_modules
   ├─ depA
   │   └─ index.js
   ├─ depB
   │   ├─ bar.js
   │   └─ node_modules
   │       └─ depA
   │           └─ index.js
   └─ depC
       └─ foobar.js
```

```

└─ node_modules
    └─ depA
        └─ index.js

```

In the previous example, `myApp`, `depB`, and `depC` all depend on `depA`. However, they all have their own private version of the dependency! Following the rules of the resolving algorithm, importing `depA` will load a different file depending on the module that requires it. Let's look at some examples:

- Importing `depA` from `/myApp/foo.js` will load `/myApp/node_modules/depA/index.js`.
- Importing `depA` from `/myApp/node_modules/depB/bar.js` will load `/myApp/node_modules/depB/node_modules/depA/index.js`.
- Importing `depA` from `/myApp/node_modules/depC/foobar.js` will load `/myApp/node_modules/depC/node_modules/depA/index.js`.

The resolving algorithm is the core part behind the robustness of the Node.js dependency management, and it makes it possible to have hundreds or even thousands of packages in an application without having collisions or problems of version compatibility between the installed packages.

Module loading in depth

To understand how ES modules work and how they can deal effectively with dependencies and even circular dependencies, we must deep dive a little bit deeper into how JavaScript code is parsed and evaluated when using ES modules.

In this section, we will learn how ES modules are loaded, present the idea of **read-only live bindings**, and finally, discuss an example with circular dependencies.

Loading phases

Node.js uses dependency graphs to load modules in the correct order. The interpreter builds this graph comprising all the necessary modules. By respecting dependencies, this graph prevents errors and guarantees that each module is available when needed.

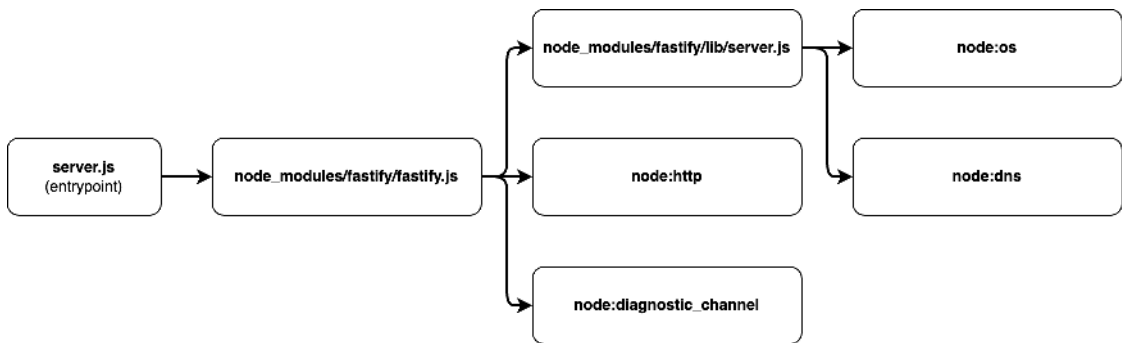


Figure 2.1: An example of Dependency Graph for a web server application using Fastify

In *Figure 2.1*, we see an example of a dependency graph. Each node in the graph represents a module, and arrows indicate when one module depends on another. The main module, `server.js` (also called the **entry point**), depends on `fastify`. `Fastify` itself relies on several modules: `node:http`, `node:diagnostic_channel`, and an internal module (`fastify/lib/server.js`). This internal module, in turn, depends on `node:os` and `node:dns`. Note that this representation is only a partial picture for illustrative purposes; in a real `Fastify`-based project, there will likely be many more dependencies.



In generic terms, a **dependency graph** can be defined as a **directed graph** (nodejsdp.link/directed-graph) representing the dependencies of a group of objects. In the context of this section, when we refer to a dependency graph, we want to indicate the dependency relationship between ES modules. As we will see, using a dependency graph allows us to determine the order in which all the necessary modules should be loaded in a given project.

Essentially, the dependency graph is needed by the interpreter to figure out how modules depend on each other and in what order the code needs to be executed. When the `node` interpreter is launched, it gets passed some code to execute, generally in the form of a JavaScript file. This file is the starting point for the dependency resolution, and it is called the **entry point**. From the entry point, the interpreter will find and follow all the `import` statements recursively in a depth-first fashion, until all the necessary code is explored and then evaluated.

This process happens in three separate phases:

- **Phase 1—construction (or parsing):** The interpreter identifies all imports and recursively loads the content of each module from their respective files.
- **Phase 2—instantiation:** For each exported entity in every module, the interpreter creates a named reference in memory, but it does not assign it a value yet. References are created for all the `import` and `export` statements to track the dependency relationships between them (**linking**). No JavaScript code is executed during this phase.
- **Phase 3—evaluation:** The `Node.js` interpreter executes the code so that all the previously instantiated entities can get an actual value. Now, running the code starting from the entry point is possible because all the blanks have been filled.

We could say that Phase 1 is about finding all the dots, Phase 2 connects those dots creating paths, and finally, Phase 3 walks through the paths in the right order.

Since these three phases are separate, no code can be executed until the entire dependency graph is fully constructed. As a result, module imports and exports must be static. We'll dive deeper into this process when we discuss how ES modules manage circular dependencies, so don't worry if the details aren't fully clear yet.



This process differs significantly from how CommonJS manages dependencies. CommonJS is dynamic; when a module is required, its content is immediately loaded and executed. This flexibility allows `require` to be used within `if` statements or loops, and module identifiers can be created from variables. However, this dynamic approach makes it challenging for CommonJS to handle cyclic dependencies effectively — a challenge that, as we'll see, ES modules manage more efficiently.

Read-only live bindings

Another fundamental characteristic of ES modules, which helps with cyclic dependencies, is the idea that imported modules are effectively *read-only live bindings* to their exported values.

Let's clarify what this means with a simple example:

```
// counter.js
export let count = 0
export function increment() {
  count++
}
```

This module exports two values: a simple integer counter called `count` and an `increment` function that increases the counter by one.

Let's now write some code that uses this module:

```
// main.js
import { count, increment } from './counter.js'
console.log(count) // prints 0
increment()
console.log(count) // prints 1
count++ // TypeError: Assignment to constant variable!
```

What we can see in this code is that we can read the value of `count` at any time and change it using the `increment()` function, but as soon as we try to mutate the `count` variable directly, we get an error as if we were trying to mutate a `const` binding.

This proves that when an entity is imported in the scope, the binding to its original value cannot be changed (*read-only binding*) unless the bound value changes within the scope of the original module itself (*live binding*), which is outside the direct control of the consumer code.



This approach is fundamentally different from CommonJS. In fact, in CommonJS, the entire exports object is copied (shallow copy) when required from a module. This means that, if the value of primitive variables like numbers or strings are changed later, the requiring module won't be able to see those changes.

Circular dependencies

Many consider circular dependencies an intrinsic design issue, but it is something that might actually happen in a real project, so it's useful for us to know how this works.

Let's walk through an example together to see how ES modules behave when dealing with circular dependencies. Let's suppose we have the scenario represented in *Figure 2.2*:

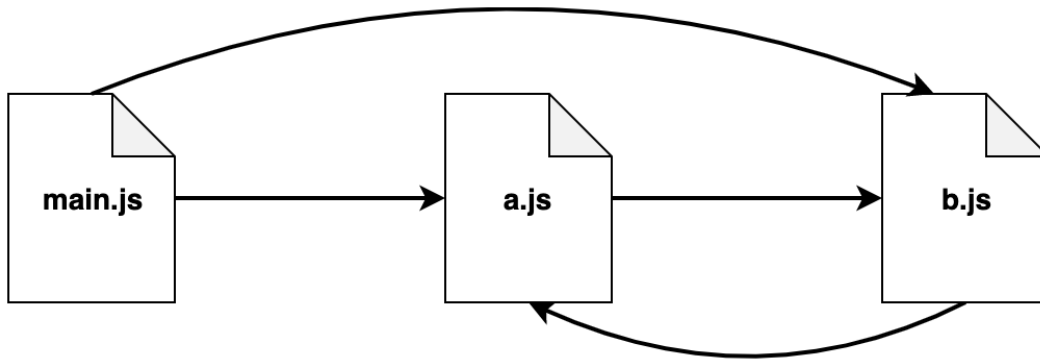


Figure 2.2: An example of circular dependency

A module called `main.js` imports `a.js` and `b.js`. In turn, `a.js` imports `b.js`. However, `b.js` relies on `a.js` as well! It's obvious that we have a circular dependency here as module `a.js` requires module `b.js` and module `b.js` requires module `a.js`.

Let's implement modules `a.js` and `b.js` first:

```
// a.js
import * as bModule from './b.js'
export let loaded = false
export const b = bModule
loaded = true
```

```
// b.js
import * as aModule from './a.js'
export let loaded = false
export const a = aModule
loaded = true
```

Note that both `a.js` and `b.js` export a binding called `loaded`, which we can use to observe what happens during the loading of each module. The value is initially set to `false`, and when the module is executed, it is immediately updated to `true`. It's also important to note that `a.js` imports `b.js`, and `b.js` imports `a.js` in return, creating the circular dependency.

Now let's see how to import those two modules in our `main.js` file (the entry point):

```
// main.js
import * as a from './a.js'
import * as b from './b.js'
console.log('a ->', a)
console.log('b ->', b)
```

When we run `main.js`, we will see the following output:

```
a -> <ref *1> [Module: null prototype] {
  b: [Module: null prototype] { a: [Circular *1], loaded: true },
  loaded: true
}
b -> <ref *1> [Module: null prototype] {
  a: [Module: null prototype] { b: [Circular *1], loaded: true },
  loaded: true
}
```

The interesting part here is that `a.js` and `b.js` have a complete view of each other. We can observe that `loaded` is set to `true` in all references to both `a` and `b`. This makes even more sense when we realize that `b` within `a`, and `a` within `b`, are actual references to the respective modules, not copies. There's no data duplication during the import and execution of the modules. If we were to swap the order of the imports in `main.js`, the output would remain the same.



If we implemented this same example using CommonJS, the result would be significantly different: the inner `loaded` values would be `false`. Additionally, if we swapped the order of imports in CommonJS, we would see a different output. These differences arise from the dynamic nature of CommonJS and the shallow copying that occurs when entities are imported.

To clearly understand how ES modules make it possible to support circular dependencies, it's worth spending some more time observing what happens in the three phases of the module resolution (parsing, instantiation, and evaluation) for this specific example.

Phase 1—parsing

During the parsing phase, the code is explored starting from the entry point (`main.js`). The interpreter looks only for `import` statements to find all the necessary modules and to load the source code from the module files. The dependency graph is explored in a depth-first fashion, and every module is visited only once. This way, the interpreter builds a view of the dependencies in a way that looks like a tree structure, as shown in *Figure 2.3*:

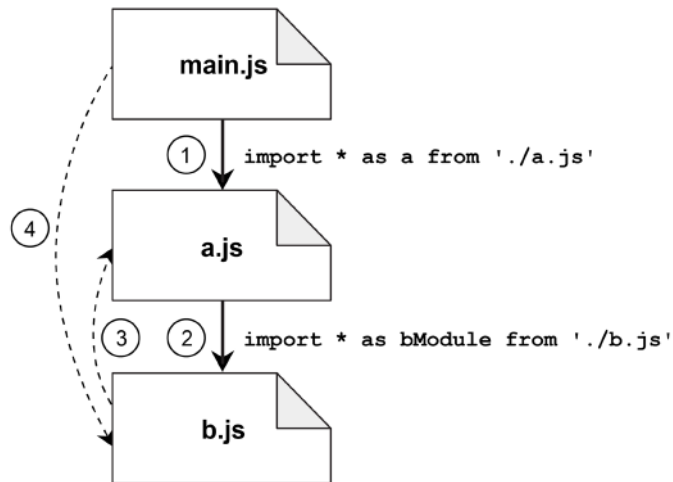


Figure 2.3: Parsing of cyclic dependencies with ES modules

Given the example in *Figure 2.3*, let's discuss the various steps of the parsing phase:

1. From `main.js`, the first import found leads us straight into `a.js`.
2. In `a.js`, we find an import pointing to `b.js`.
3. In `b.js`, we also have an import back to `a.js` (our cycle), but since `a.js` has already been visited, this path is not explored again.
4. At this point, the exploration starts to wind back: `b.js` doesn't have other imports, so we go back to `a.js`; `a.js` doesn't have other import statements, so we go back to `main.js`. Here, we find another import pointing to `b.js`, but again, this module has been explored already, so this path is ignored.



To clarify the depth-first approach, imagine `main.js` also imports a module called `c.js`, listed after `a.js`. With this setup, `c.js` would only be parsed once both `a.js` and `b.js` are fully processed, and the parsing process returns to `main.js` to handle the remaining imports. The general idea behind a depth-first traversal is to explore each branch as deeply as possible before backtracking to the parent node to explore other paths. If this concept is still unclear, consider reviewing this link on **depth-first search (DFS)** for more examples and illustrations: nodejsdp.link/dfs

At this point, our depth-first visit of the dependency graph has been completed and we have a linear view of the modules, as shown in *Figure 2.4*:

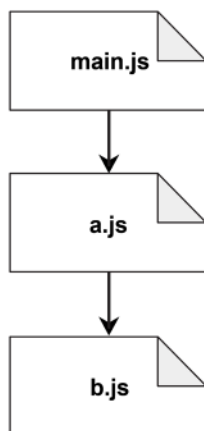


Figure 2.4: A linear view of the module graph where cycles have been removed

This particular view is quite simple. In more realistic scenarios with a lot more modules, the view will look more like a tree structure.

Phase 2—instantiation

In the instantiation phase, the interpreter walks the tree view obtained from the previous phase from the bottom to the top. For every module, the interpreter will look for all the exported properties first and builds out a map of the exported names in memory:

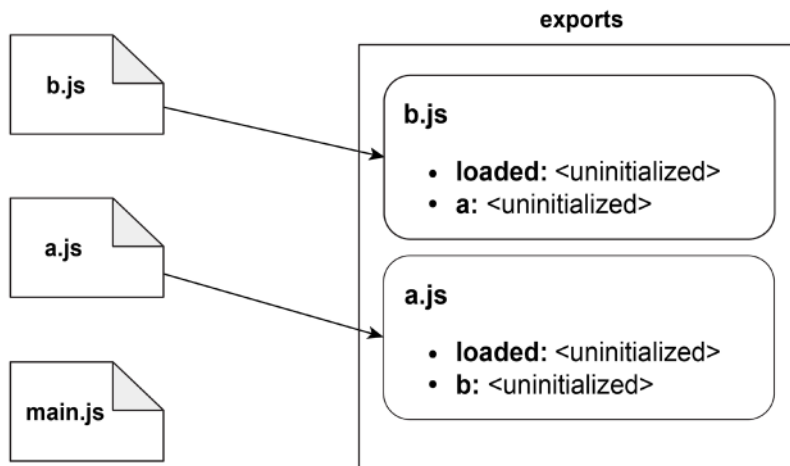


Figure 2.5: A visual representation of the instantiation phase

Figure 2.5 describes the order in which every module is instantiated:

1. The interpreter starts from `b.js` and discovers that the module exports `loaded` and `a`.
2. Then, the interpreter moves to `a.js`, which exports `loaded` and `b`.
3. Finally, it moves to `main.js`, which does not export any functionality.



Note that, in this phase, the exports map only keeps track of the exported names; their associated values are considered uninitialized for now.

After this sequence of steps, the interpreter will do another pass to link the exported names to the modules importing them, as shown in Figure 2.6:

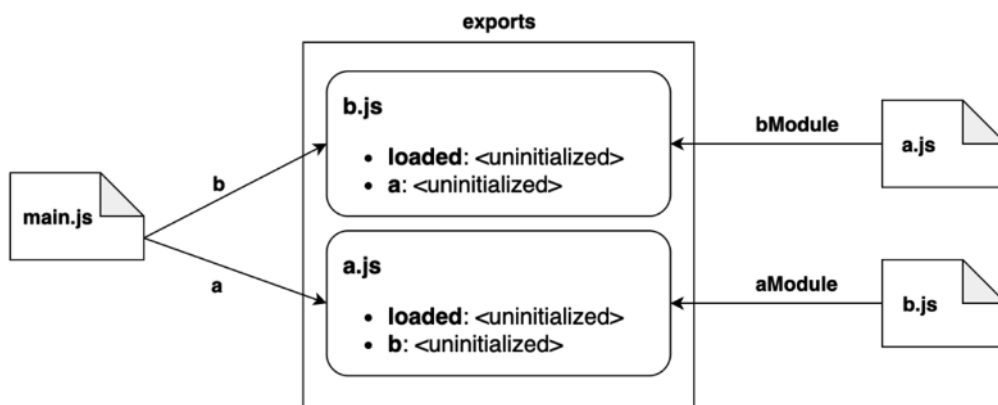


Figure 2.6: Linking exports with imports across modules

We can describe what we see in Figure 2.6 through the following steps:

1. Module `b.js` will link the exports from `a.js`, referring to them as `aModule`.
2. In turn, `a.js` will link to all the exports from `b.js`, referring to them as `bModule`.
3. Finally, `main.js` will import all the exports in `b.js`, referring to them as `b`; similarly, it will import everything from `a.js`, referring to them as `a`.



Again, it's important to note that all the values are still uninitialized. In this phase, we are only linking references to values that will be available at the end of the next phase.

Phase 3—evaluation

The last step is the evaluation phase. In this phase, all the code in every file is finally executed. The execution order is again bottom-up, respecting the post-order depth-first visit of our original dependency graph. With this approach, `b.js` is executed first, then `a.js`, and `main.js` is the last file to be executed. This way, we can be sure that all the exported values have been initialized before we start executing our main business logic:

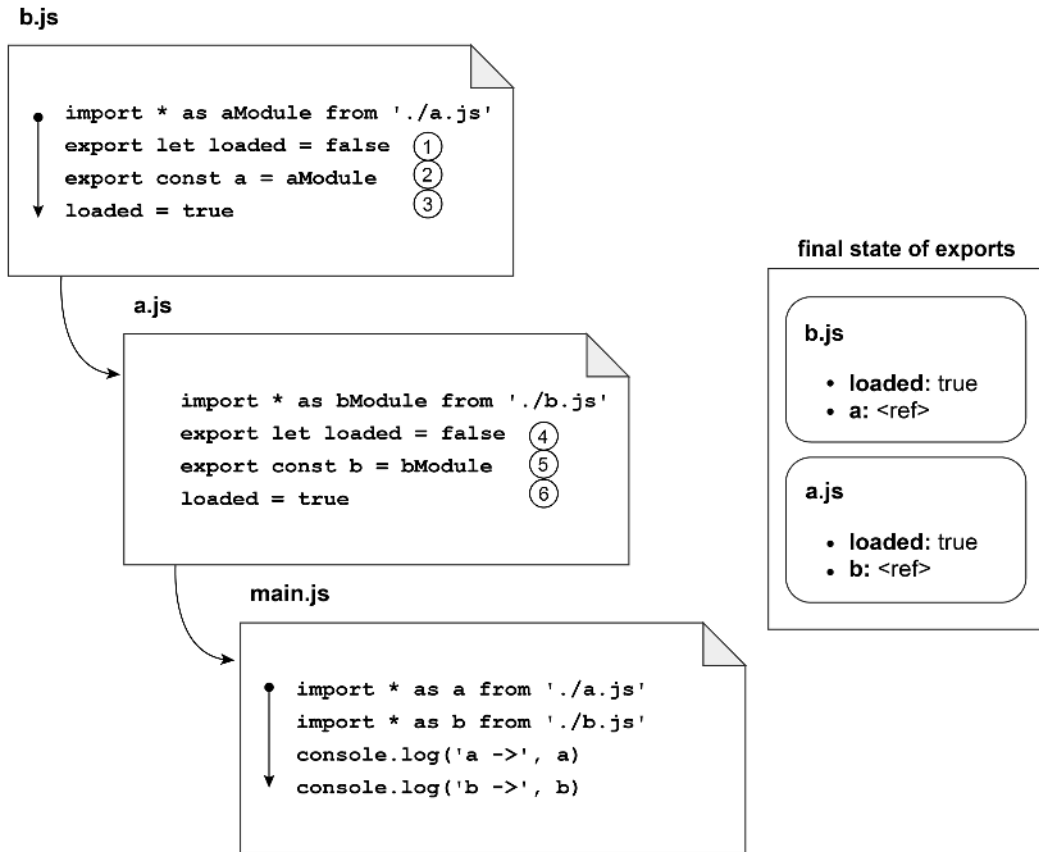


Figure 2.7: A visual representation of the evaluation phase

Following along from the diagram in Figure 2.7, this is what happens:

1. The execution starts from `b.js` and the first line to be evaluated initializes the `loaded` export to `false` for the module.
2. Similarly, the exported property `a` gets evaluated here. This time, it will be evaluated as a reference to the module object representing module `a.js`.
3. The value of the `loaded` property gets changed to `true`. At this point, we have fully evaluated the state of the exports for module `b.js`.
4. Now the execution moves to `a.js`. Again, we start by setting `loaded` to `false`.

5. At this point, the `b` export is evaluated to a reference to module `b.js`.
6. Finally, the `loaded` property is changed to `true`. Now we have finally evaluated all the exports for `a.js` as well.

After all these steps, the code in `main.js` can be executed, and at this point, all the exported properties are fully evaluated. Since imported modules are tracked as references, we can be sure every module has an up-to-date picture of the other modules, even in the presence of circular dependencies.

Modules that modify other modules

Sometimes, we come across modules that don't provide any exports. While this might seem unusual, it's important to recognize that a module can still offer functionality by modifying the global scope or objects within it, including other modules. Although this practice is generally discouraged, it can be useful and safe in specific situations, such as for testing, and is occasionally employed in real-world projects.

This technique, where a module modifies other modules or objects in the global scope, is known as **monkey patching**. Monkey patching refers to the practice of altering existing objects at runtime to change or extend their behavior, or to apply temporary fixes.

Let's consider an example of monkey patching. Suppose we have a module that provides a basic console logger implementation. We could then use another module, perhaps maintained by a third party, to enhance the original logger by adding ASCII color codes. This modification would colorize the logger's output in the terminal based on the log level: info messages in green, warnings in yellow, and errors in red. This colorizer module is optional, allowing users to enable colored output only if they choose to import it:

```
// Logger.js
export const logger = {
  info(message) {
    console.log(` [INFO] \t${message}` )
  },
  error(message) {
    console.log(` [ERROR] \t${message}` )
  },
  warn(message) {
    console.log(` [WARN] \t${message}` )
  },
  debug(message) {
    console.log(` [DEBUG] \t${message}` )
  },
}
```

This is a basic logger implementation that exports a global logger object with methods such as `info()`, `error()`, `warn()`, and others. Each method prints a message with a prefix tag indicating the specific log level. This design helps users easily search and filter the logs produced by the application.

Now, let's implement the `colorizeLogger.js` module that enhances this logger by adding support for ASCII colors:

```
// colorizeLogger.js
import { logger } from './logger.js'

const RED = '\x1b[31m'
const YELLOW = '\x1b[33m'
const GREEN = '\x1b[32m'
const WHITE = '\x1b[37m'
const RESET = '\x1b[0m'

const originalInfo = logger.info
const originalWarn = logger.warn
const originalError = logger.error
const originalDebug = logger.debug

logger.info = message => originalInfo(`${GREEN}${message}${RESET}`)
logger.warn = message => originalWarn(`${YELLOW}${message}${RESET}`)
logger.error = message => originalError(`${RED}${message}${RESET}`)
logger.debug = message => originalDebug(`${WHITE}${message}${RESET}`)
```

`colorizeLogger.js` imports the original `logger.js` module and overrides the implementations of the `info`, `warn`, `error`, and `debug` methods. However, these new implementations still call the original methods to handle the actual logging. The `colorizeLogger.js` module only modifies the input message by applying ASCII color codes, while the original methods remain responsible for printing the log entries. This approach maintains a clear separation of concerns: the logger module is responsible for how logs are printed to the terminal, while the colorizer module is solely responsible for adding color to the messages before they are sent to the logger.

Finally, let's see how we can use these two modules together in a hypothetical application:

```
// main.js
import { logger } from './logger.js'
import './colorizeLogger.js'

logger.info('Hello, World!')
logger.warn('Free disk space is running low')
logger.error('Failed to connect to database')
logger.debug('main() is starting')
```

In this file, importing `colorizeLogger.js` could be optional. You could comment it out to disable colored output without affecting the application's functionality. Additionally, since `colorizeLogger.js` does not export any entities, we use a simplified import syntax that omits the `from` keyword.

This example illustrates how a module can patch another module using ES modules. However, this technique can be risky. Modifying the global namespace or other modules introduces **side effects**, impacting the state of entities outside their intended scope. This can lead to unpredictable consequences, especially when multiple modules interact with the same entities. For instance, if two different modules attempt to set the same global variable or alter the same property of a module, the results can be uncertain (e.g., which module's changes prevail?). This unpredictability can affect the entire application.

Therefore, it is important to use this technique with caution and ensure you understand all potential side effects before implementing it.



This logger implementation is purely for demonstration purposes and is not intended for production use. If you're looking for a logger for your Node.js applications, consider using `pino` (nodejsdp.link/pino), a highly efficient, comprehensive, and extensible logging library. Pino is designed with extensibility in mind, and there are extensions like `pino-colada` (nodejsdp.link/pino-colada) that provide enhanced output formatting and coloring.

We learned that entities imported through ES modules are *read-only live bindings*, meaning we cannot reassign them from an external module. However, there is a caveat. While we can't change the bindings of the default export or named exports of an existing module from another module, if one of these bindings is an object, we can still mutate the object itself by reassigning some of its properties. This is precisely what we did in our logger example. Since `logger.js` exports a binding called `logger` that is an object, we are able to modify the properties within that object, though we wouldn't be able to reassign the `logger` binding entirely, as in the following example:

```
// replaceLogger.js
import { logger } from './logger.js'

const GREEN = '\x1b[32m'
// ...
const RESET = '\x1b[0m'

// replacing the logger binding with a new object
// this will fail with a TypeError: Assignment to constant variable.
logger = {
  info: message => {
    console.log(`INFO: ${GREEN}${message}${RESET}`)
  },
  // ...
}
```

This partially implemented example demonstrates a potential, but flawed, alternative approach to implementing our log output colorizer. Since this implementation attempts to override the logger binding entirely, running this module will lead to an error: `TypeError: Assignment to constant variable`. It's important to note, though, that while we cannot reassign the logger object itself, we could still patch the individual methods of the logger instance.

An alternative (though still flawed) approach to monkey patching the entire logger binding could be as follows:

```
// replaceLogger2.js
import * as loggerModule from './logger.js'

const GREEN = '\x1b[32m'
// ...
const RESET = '\x1b[0m'

// replacing the logger binding inside the loggerModule with a new object
// this will fail with a TypeError: Cannot assign to read only property
loggerModule.logger = {
  info: message => {
    console.log(`INFO: ${GREEN}${message}${RESET}`)
  },
  // ...
}
```

The key difference in this approach is that we are now importing the entire `logger.js` module using a namespace import and then attempting to reassign the `logger` member within it. However, this approach also fails because module members are read-only bindings. Running this code would result in an error message confirming the issue: `TypeError: Cannot assign to read only property 'logger' of object '[object Module]'`. Note that even in this case, while we cannot reassign the logger object itself, we could still patch the individual methods of the logger instance.

There's another approach we could attempt, but it requires making some modifications to the `logger.js` module first:

```
// logger.js
export const logger = {
  // ...
}

export default {
  logger,
}
```

We haven't changed the module's implementation, but we've added a default export that wraps all the module's members—specifically, the `logger` binding—into an object. By doing this, the default export becomes an object, and while we cannot reassign the entire default binding, we can still modify the individual members of the object. This approach provides the flexibility needed to create a functional alternative for our logger replacement module:

```
// replaceLogger3.js
import loggerModule from './logger.js'

const GREEN = '\x1b[32m'
// ...
const RESET = '\x1b[0m'

loggerModule.logger = {
  info: message => {
    console.log(`INFO: ${GREEN}${message}${RESET}`)
  },
  // ...
}
```

This approach closely resembles what we did with our implementation of the `replaceLogger2.js` module, but with one crucial difference: instead of performing a namespace import, we are now importing the default export from the `logger.js` module. With a namespace import, we receive a module instance with read-only members, preventing us from reassigning the `logger` member. By using a default import, however, we obtain a plain object. This allows us to modify its members, including the `logger` member, as intended. As a result, executing this code will proceed smoothly without errors.

Let's use this new monkey-patching module. However, there's a caveat. Let's say that we want to use the following code:

```
// main2Broken.js
import { logger } from './logger.js'
import './replaceLogger3.js'

logger.info('Hello, World!')
```

This will mean that the output will not be colorized as expected. This is because we're importing the named export `logger` from `logger.js`, but `replaceLogger3.js` only patches the object exported as default export by `logger.js`. We can illustrate the effect of monkey patching applied by `replaceLogger3.js` in *Figure 2.8*:

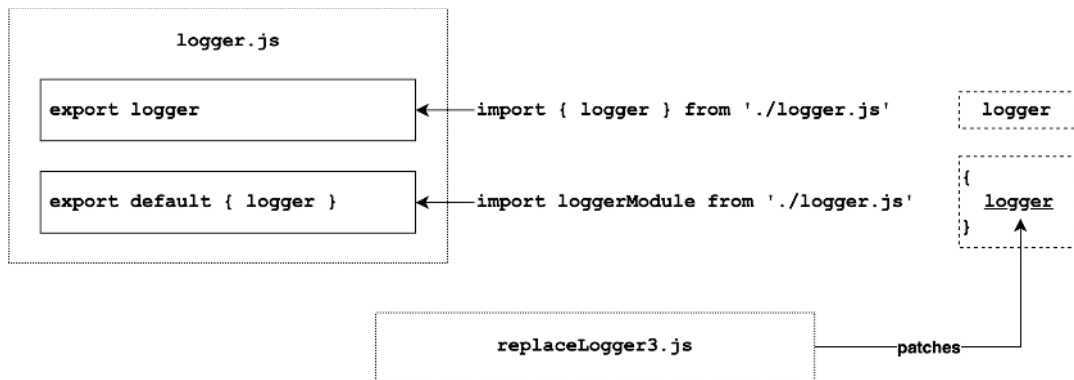


Figure 2.8: How `replaceLogger3.js` monkey-patches a member of the default export

In this example, `replaceLogger3.js` imports the object that `logger.js` exports as the default export. It then patches the specific `logger` binding within that object. This means the patching only affects the default export and not the named export. Therefore, to use the patched version of the `logger`, we must ensure that our main module imports the default export.

Let's correct our code to do exactly this:

```
// main2.js
import loggerModule from './logger.js'
import './replaceLogger3.js'

loggerModule.logger.info('Hello, World!')
loggerModule.logger.warn('Free disk space is running low')
loggerModule.logger.error('Failed to connect to database')
loggerModule.logger.debug('main() is starting')
```

In this updated version, we import the default export from `logger.js`. By doing so, `replaceLogger3.js` patches the same reference that is used throughout the rest of the code, resulting in colorized output as intended.

This approach is somewhat fragile because it requires several conditions to be met:

- The subject module (`logger.js` in our example) must export an object containing all its members as the default export.
- The patching module (`replaceLogger3.js` in our example) must import the default export and modify its members as needed.
- The using module (`main2.js` in our example) must import and use the default export of the subject module rather than its named exports directly.

If the subject or patching module is from a third party, we might not have control over these conditions, which could prevent us from applying this technique. However, as we will see in *Chapter 10, Testing: Patterns and Best Practices*, this technique can be used with Node.js core modules and can be useful for writing unit tests. We will explore examples of using this patching technique to mock the filesystem module in unit tests.



Monkey patching can sometimes be used as an attack vector. For example, if an attacker manages to include malicious code in an application, this code might apply monkey patching to specific modules to be able to exfiltrate sensitive data, perform privilege escalation, or change important behaviors of the application. For this reason, the Node.js team has associated monkey patching with the CWE-349. **Common Weakness Enumeration (CWE)** is a community-developed list of software and hardware weaknesses that can become vulnerabilities. The Node.js teams also provide some suggestions on how you might prevent monkey patching entirely if you want to harden your application: nodejsdp.link/cwe-349

How monkey patching affects type safety in TypeScript projects

If you use TypeScript, you should be aware that applying monkey patching with TypeScript comes with additional challenges that need to be carefully considered. Before we explore some of these, keep in mind that TypeScript executes type checks and compiles your code to plain JavaScript before it can be executed. Monkey patching is something that happens at runtime, meaning it takes place only after TypeScript has done its job.

Let's see what kind of challenges this creates:

- **Type safety issues:** TypeScript is built to provide type safety by ensuring that variables, functions, and objects are used according to their defined types. With monkey patching, you are generally altering the structure or behavior of existing objects or modules at runtime. You might be changing some types in a way that conflicts with the original type definitions. Imagine, for example, inadvertently changing the type of a member from a number to a string. This can lead to unexpected type errors at runtime. If, at some point in the code, the `toFixed()` method is called, this would result in a runtime error because the new type (`string`) does not have this particular method. TypeScript might be able to warn you about this potential issue if you are performing monkey patching in the same TypeScript project by providing some errors in the place where the patching is performed (typically with an error such as `"Type 'string' is not assignable to type 'number'"`). However, if the patching happens in a third-party library or a plain JavaScript file, in most circumstances, TypeScript won't be able to spot the type-safety issue and you'll be at risk of type-related runtime errors.
- **Challenges with type declaration files:** Working with TypeScript declaration files (`.d.ts`) becomes more complex when monkey patching is involved. You may need to extend existing types or modules, which can be tricky and error-prone, particularly with complex type structures.

- **Loss of IntelliSense and autocompletion:** One of TypeScript's major benefits is providing intelligent code completion and suggestions based on types. When you perform monkey patching, you might end up changing the signature of functions or the members of an object. The IDE might not be able to correctly capture these changes, so you might not see them being correctly represented when an autocompletion suggestion pops up. If you are adding extra functionality, for example, by adding a new method to a class, this method might not appear at all in the autocompletion box. All of these small issues can create development friction, slow down development, and increase the likelihood of mistakes.

Given these challenges, it's important to think carefully before using monkey patching in a TypeScript project. If you care about type safety, you should also be aware if you are importing third-party modules that might apply monkey patching as this might come with the undesired side effects that we just discussed. While monkey patching can be useful for testing, in most other cases, it should be viewed as a last resort for extending modules.

CommonJS modules

As we've already mentioned throughout this chapter, CommonJS is gradually being replaced by ES modules, and it's now recommended to use ES modules for new projects and libraries. However, CommonJS is not officially deprecated, and given its long-standing role in the JavaScript ecosystem, it's still important to have a basic understanding of it. This knowledge will enable you to work confidently with older code bases and libraries.

Let's have a look at the basic syntax of CommonJS.

Two of the main concepts of the CommonJS specification are as follows:

- `require`, which is a function that allows you to import a module from the local filesystem.
- `exports` and `module.exports`, which are special variables that can be used to export public functionality from the current module.

A simple CommonJS module might look like this:

```
// math.cjs
'use strict'

function add(a, b) {
  return a + b
}

module.exports = { add } // or `exports.add = add`
```

We can then use this module as follows:

```
'use strict'  
  
const { add } = require('./math.cjs')  
  
console.log(add(2, 3)) // 5
```

These snippets show how straightforward it is to export functionality using the `module.exports` variable (or simply `exports`) and how we can easily import that functionality in another module with the `require()` function.

It's important to note that we explicitly enabled strict mode using `'use strict'`, as CommonJS does not operate in strict mode by default. We'll explore this detail further in the next section.

Another crucial detail to note is that the `require()` function in CommonJS is both synchronous and dynamic. It operates in a straightforward manner—when executed, `require()` resolves the specified module, loads the associated file, and runs its content immediately, without needing a callback. This synchronous nature impacts how we define modules, as it generally limits us to using synchronous code during module initialization. This is also why many core Node.js libraries provide synchronous APIs alongside their asynchronous counterparts (e.g., `readFile` and `readFileSync` in `node:fs`).

If a module requires asynchronous initialization, one approach is to define and export an uninitialized module that is later initialized asynchronously. However, this method doesn't ensure the module is ready for use immediately after being loaded. We will explore this issue and offer some elegant solutions in *Chapter 11, Advanced Recipes*.



Node.js originally included an asynchronous version of `require()`, but it was removed early on because it complicated a process meant for use during initialization, where asynchronous I/O often introduces more challenges than benefits.

ES modules and CommonJS—differences and interoperability

Let's now discuss some important differences between ES modules and CommonJS and how the two module systems can work together when necessary.

Strict mode

As opposed to CommonJS, ES modules run implicitly in strict mode. This means that we don't have to explicitly add the `'use strict'` statements at the beginning of every file. Strict mode cannot be disabled; therefore, we cannot use undeclared variables or the `with` statement or have other features that are only available in non-strict mode. However, this is definitely a good thing, as strict mode is a safer execution mode.



If you are curious to find out more about the differences between the two modes, you can check out a very detailed article on MDN Web Docs (nodejsdp.link/strict-mode).

Top-level await

Top-level await allows developers to use `await` at the top level of a module, simplifying asynchronous code without needing to wrap it in an `async` function. In ES modules, `await` can be used directly at the top level, as shown here:

```
// main.mjs
import { loadData } from 'someModule'

console.log(await loadData())
```

However, in CommonJS modules, `await` cannot be used at the top level directly. Instead, you must use it within an `async` function:

```
// main.cjs
'use strict'

const { loadData } = require('someModule')

async function main() {
  console.log(await loadData())
}

main()
```

If you try to use `await loadData()` outside of an `async` function, you'll get a `SyntaxError`. This limitation makes it easier to work with `async/await` in ES modules compared to CommonJS.

Behavior of `this`

An interesting difference between ES modules and CommonJS is the behavior of the `this` keyword.

In the global scope of an ES module, `this` is `undefined`, while in CommonJS, `this` is a reference to `exports`:

```
// main.mjs - ES modules
console.log(this) // undefined

// main.cjs - CommonJS
console.log(this === exports) // true
```



Note that in both module systems, the behavior of the `globalThis` variable is consistent and will reference an object that contains global platform utilities such as `setInterval` or `structuredClone`. If you want to find out more about `globalThis`, check out [nodejsdp.link/global-this](https://nodejs.org/docs/latest/api/global.html#globalthis).

Missing references in ES modules

If you're accustomed to using CommonJS, you might be surprised by the absence of certain familiar references in ES modules, such as `require`, `exports`, `module.exports`, `__filename`, and `__dirname`. If we try to use any of them within an ES module, since it also runs in strict mode, we will get a `ReferenceError`:

```
console.log(exports) // ReferenceError: exports is not defined
console.log(module) // ReferenceError: module is not defined
console.log(__filename) // ReferenceError: __filename is not defined
console.log(__dirname) // ReferenceError: __dirname is not defined
```

`__filename` and `__dirname` represent the absolute path to the current module file and the absolute path to its parent folder. Those special variables can be very useful when we need to build a path relative to the current file.

In ES modules, one useful object is `import.meta`. While we've already discussed `import.meta.resolve()`, there are other interesting properties within this object. For example, you can obtain equivalents to `__filename` and `__dirname` by using `import.meta.filename` and `import.meta.dirname`, respectively:

```
// main.js
const __filename = import.meta.filename // /path/to/project/main.js
const __dirname = import.meta.dirname // /path/to/project
```

These properties are still relatively new (they were introduced in Node.js v20.11.0, released in January 2024). If you're working with an earlier version, you can use `import.meta.url`, which is a reference to the current module file in a format similar to `file:///path/to/current_module.js`. Let's see how this value can be used to reconstruct the current file path and its parent directory in the form of absolute paths:

```
// main.js
import { fileURLToPath } from 'node:url'
import { dirname } from 'node:path'

const __filename = fileURLToPath(import.meta.url) // /path/to/project/main.js
const __dirname = dirname(__filename) // /path/to/project
```

This works because `import.meta.url` will give us a URL representing the current file in the `"file:///path/to/project/main.js"` form. The utility `fileURLToPath()` takes a `"file:///..."` URL and converts it to the equivalent path `"/path/to/project/main.js"`. Finally, `dirname()` can take a path for a file and return the directory path for that file `"/path/to/project"`.

Import interoperability

Sometimes, you may need to import an ES module from a CommonJS module or vice versa. There are a few important details and caveats to consider when handling these cross-module imports. Let's take a quick look at what's possible and how to manage these imports under different circumstances.

Import CommonJS modules from ES modules

An import statement can be used in an ES module to load a CommonJS module.

Let's see a quick example:

```
// someModule.cjs
'use strict'

module.exports = {
  someFeature: 'someFeature',
}
```

This CommonJS module exports an object with a property `someFeature` using `module.exports`. Now, let's see how we can import this module from an ES module:

```
// main.js
import someModule from './someModule.cjs'

console.log(someModule)
```

This works as expected and the output of this file will be as follows:

```
{ someFeature: 'someFeature' }
```

However, what if we only want to import `someFeature`? Let's try:

```
// main2.mjs
import { someFeature } from './someModule.cjs'

console.log(someFeature)
```

If we run this code, we will see a rather verbose error message:

```
import { someFeature } from './someModule.cjs'
      ^^^^^^^^^^^^^^

SyntaxError: Named export 'someFeature' not found. The requested module './
someModule.cjs' is a CommonJS module, which may not support all module.exports
as named exports.

CommonJS modules can always be imported via the default export, for example
using:
```

```
import pkg from './someModule.cjs';  
const { someFeature } = pkg;
```

The issue arises because `someFeature` wasn't exported directly using `exports.someFeature`, so it isn't available as a named export when imported from an ES module. If you can modify the CommonJS module, you could solve this by assigning `someFeature` to `exports.someFeature`. However, if you can't change the source module (for example, if it's a third-party module installed from npm), there are a couple of workarounds you can rely on.

The first solution is the one recommended by the preceding error message. We can import the entire module (using the default import syntax) and then use **destructuring assignment** syntax to extract `someFeature` from the imported object:

```
// main3.mjs  
import someModule from './someModule.cjs'  
  
const { someFeature } = someModule // destructuring assignment  
console.log(someFeature)
```



In the previous snippet, the destructuring assignment is a compact way to extract the property `someFeature` from `someModule` and assign it to a local variable called `someFeature`. It is practically equivalent to writing the following:

```
const someFeature = someModule.someFeature
```

The other approach is to recreate the `require()` function in ES modules by using the `createRequire()` function provided by the core Node.js module `node:module`:

```
// main4.mjs  
import { createRequire } from 'node:module'  
  
const require = createRequire(import.meta.url)  
const { someFeature } = require('./someModule.cjs')  
  
console.log(someFeature)
```

The `require()` function that we created in this example behaves exactly like `require()` behaves in a CommonJS module. Therefore, we can use it without being too worried about incompatibilities. We will see how this can be used to import JSON files in the context of ES modules as well later in this chapter.

Import ES modules from CommonJS

At the time of writing this book, when trying to import an ES module into a CommonJS module, you might encounter issues. Let's see an example:

```
// someModule.mjs
export const someFeature = 'someFeature'

// main.cjs
'use strict'
const { someFeature } = require('./someModule.mjs')
```

Depending on your Node.js version, when you run `main.cjs`, you might see the following error:

```
Error [ERR_REQUIRE_ESM]: require() of ES Module [...]someModule.mjs not
supported.
Instead change the require of [...]someModule.mjs to a dynamic import() which is
available in all CommonJS modules.
```

The error message is clear: you can't use `require()` to import an ES module in a CommonJS module. However, you can use a dynamic `import()` instead, which is supported in CommonJS modules. Here's how to update the example:

```
// main2.cjs
'use strict'

async function main() {
  const { someFeature } = await import('./someModule.mjs')
  console.log(someFeature)
}

main()
```

This code works as expected. When executed, `someModule.mjs` is dynamically imported, and "someFeature" is printed to the console. However, keep in mind that because dynamic imports are asynchronous, and you're working within a CommonJS module, you can't use top-level `await`. You must wrap your logic in an async function and then invoke that function, which adds some boilerplate.

Fortunately, Node.js offers an experimental flag, `--experimental-require-module`, that allows direct imports of ES modules from CommonJS modules. This means our original implementation can work if you run the file like this:

```
node --experimental-require-module main.cjs
```


This feature simplifies interoperability between CommonJS and ES modules, making it easier for developers and library maintainers to transition to ES modules without worrying about CommonJS compatibility. By the time you read this, it's possible this flag has become stable, eliminating the need for workarounds entirely.



Note that, at the time of writing, the `--experimental-require-module` flag does not support ES modules using top-level `await`.

Importing JSON files

JSON is a widely used data format in web development, and it's common to need to load data from a JSON file into your programs. In CommonJS, this is straightforward:

```
// main.cjs
'use strict'

const data = require('./sample.json')
console.log(data)
```

Using `require()`, you can load and parse a JSON file with a single line of code. This function automatically reads the file's content, parses it as JSON, and returns the result. It's equivalent to manually reading the file with `readFileSync()` and parsing it with `JSON.parse()`, but with much less boilerplate.

Now, let's see how to do the same in an ES module:

```
// main.mjs
import data from './sample.json'
console.log(data)
```

This seems logical, but it won't work. Running this code will result in an error:

```
TypeError [ERR_IMPORT_ATTRIBUTE_MISSING]: Module "[...]/sample.json" needs an
import attribute of "type: json"
```

This error is actually quite informative. It tells us that we need to specify that we're importing a JSON file by adding an **import attribute**. Here's how to do it:

```
// main2.mjs
import data from './sample.json' with { type: 'json' }
console.log(data)
```

This works, but at the time of writing, you'll see a warning that JSON module support is still experimental in Node.js.



The reason for this special syntax is security. By explicitly declaring the type, the ES module loader understands that it's dealing with JSON and won't try to execute any code from the module. This prevents a scenario where, even if the file has a `.json` extension, it could still contain JavaScript code that could then be executed, creating a potential attack vector. If you want to know more about this feature, you should check out the `import attributes` proposal repository at nodejsdp.link/proposal-import-attributes.

This syntax, with some small syntactic differences, can also be used with dynamic ES module imports:

```
// main3.mjs
const { default: data } = await import('./sample.json', {
  with: { type: 'json' },
})
console.log(data)
```

Here, the only difference is that the result of the import is wrapped in a default property, so you need to access the JSON data through it. Like the static import, this code will also trigger the experimental feature warning.

If you're not comfortable using experimental features, there are alternative approaches. The most obvious is to manually read and parse the JSON file:

```
// main4.js
import { readFile } from 'node:fs/promises'
import { join } from 'node:path'

const jsonPath = join(import.meta.dirname, 'sample.json')
try {
  const dataRaw = await readFile(jsonPath, 'utf-8')
  const data = JSON.parse(dataRaw)
  console.log(data)
} catch (error) {
  console.error(error)
}
```

While this method works, it requires more code. A more concise option is to use the `require()` function within an ES module by utilizing the `createRequire()` utility that we discussed earlier:

```
// main5.mjs
import { createRequire } from 'node:module'

const require = createRequire(import.meta.url)
const data = require('./sample.json')

console.log(data)
```

Importing JSON files in Node.js can be straightforward or a bit tricky, depending on whether you're using CommonJS or ES modules. CommonJS makes it easy with the `require()` function, while ES modules add a bit more complexity with a special syntax to ensure security. Although the new syntax for importing JSON in ES modules is still experimental, it's designed to protect against potential risks. Whether you prefer the simplicity of CommonJS or want to embrace the security features of ES modules, you have options to choose from based on what works best for your project.

Using modules in TypeScript

When using different module systems in TypeScript, it's important to understand that TypeScript is a superset of JavaScript, designed to integrate smoothly with various ecosystems and platforms, including browsers, Node.js, and other JavaScript environments. As a compiler (or transpiler), TypeScript handles modules in two main contexts: how you structure your code with modules during development (input or *detected modules*) and the format of modules when your TypeScript code is compiled into JavaScript (output or *emitted modules*). TypeScript can even convert between different module systems, allowing you to write code using ES modules and compile it to CommonJS, for example.

The way TypeScript operates can vary from project to project and is determined by the `tsconfig.json` configuration file. This file offers a wide range of options with considerable flexibility, which can sometimes be overwhelming and make it difficult to achieve the desired results.

In this section, we will understand how TypeScript works when it comes to handling modules and cover some of the most important configuration options related to them. These options should help you create a configuration that meets your needs or, if you are working on an existing TypeScript code base, understand how modules are intended to be used in that particular project.

The role of the TypeScript compiler

The primary goal of the TypeScript compiler is to catch potential runtime errors during compile time. Whether or not modules are involved, the compiler must understand the specific characteristics of the expected runtime environment (the host system), including, for example, what global variables will be available. When modules are introduced, the compiler has to deal with additional challenges.

Let's discuss them with an example:

```
// hello.ts
import sayHello from 'greetings'

sayHello('world')
```

To accurately compile `hello.ts`, the TypeScript compiler needs to determine several key factors about the structure of the input code and the characteristics of the target environment that will be executing the compiled code:

- **Module loading:** Assuming that the `greetings` module was originally written in TypeScript, will the module system load a TypeScript file (e.g., `greetings.ts`), or will it load a pre-compiled JavaScript file (e.g., `greetings.js`) that might be available in the same package?

- **Module type and module resolution:** What kind of module format does the target system expect, based on the file name and location? This is convention-based. In our example, the module specifier is only "greetings", so this probably refers to a third-party library installed in the `node_modules` folder. However, this information alone doesn't explicitly dictate which exact file needs to be loaded. TypeScript also supports **path aliases** (or **path remapping**), which allows developers to define their own custom path prefixes to import files from the current project (nodejsdp.link/ts-path-aliases). This can be a convenient feature, but it's also something else that can affect module resolution. Therefore, it needs to be properly configured and accounted for during the compilation phase. Once a module file has been identified and loaded, the next question is: what module type is the loaded file using? We are using the ES module syntax when importing the `greetings` module, but this doesn't necessarily imply that the loaded file is an ES module. In fact, it might be a CommonJS module.
- **Output transformation:** How will the module syntax be transformed during the output process? For example, should all imports and exports be converted from ES modules to CommonJS?
- **Compatibility:** Can the detected module types interact correctly based on the syntax transformation?
- **Binding:** What specific export from the `greetings` module is bound to `sayHello`?

These questions are heavily dependent on the characteristics of the host system that consumes the output JavaScript or raw TypeScript—typically a runtime like Node.js or a bundler like Webpack. While the ECMAScript specification outlines how ES module imports and exports should link, it doesn't define how module resolution occurs or address other module systems like CommonJS. This means that runtimes and bundlers have significant leeway in designing their own rules. As a result, TypeScript's approach to these questions can vary depending on where the code will run. There isn't a single correct answer, so the compiler needs to be configured appropriately.

Thus, TypeScript's responsibility when it comes to modules can be summarized as follows:

- **Adapting to the host:** Understanding the rules of the host system (e.g. Node.js) well enough to compile files into a valid output module format.
- **Ensuring compatibility:** Ensuring that imports in the output files will resolve correctly.
- **Type assignment:** Assigning accurate types to imported entities.



For more details and examples, you can consult the official **Modules (theory)** section from the TypeScript website at nodejsdp.link/ts-modules-theory.

Configuring the module output format

The module compiler option informs the compiler about the desired module format for emitted JavaScript. While its main role is to control the output format, it also guides the compiler on how to detect module types, manage imports between different module kinds, and handle features like `import.meta` and top-level `await`. Even if your TypeScript project doesn't emit JavaScript files (`noEmit`), selecting the right module setting is crucial for proper type-checking and IntelliSense.



For exhaustive guidance on choosing the right module setting for your project, see the **Modules** section in the official TypeScript handbook: nodejsdp.link/ts-modules

When working with Node.js, we recommend setting the `module` option to `NodeNext`, reflecting the latest Node.js module system.

Input module syntax and output emission

It's important to understand that the input module syntax in your TypeScript files doesn't always correspond directly to the output module syntax in JavaScript files. For example, a file with ES module imports might be emitted as ES modules or transformed into CommonJS, depending on the `module` compiler option and any relevant detection rules. This flexibility means that merely inspecting an input file isn't enough to determine whether it's an ES module or a CommonJS module—TypeScript can convert between module systems, a feature that, while powerful, can sometimes lead to unexpected results and interoperability issues.

In TypeScript 5.0, the `verbatimModuleSyntax` option was introduced to help developers understand exactly how their import and export statements will be emitted. When enabled, this flag requires that imports and exports in input files be written in the form that will undergo the least transformation before emission. For example, if a file is emitted as ES modules, its imports and exports must be written in ES module syntax. When targeting Node.js, we recommend enabling `verbatimModuleSyntax` to keep results consistent and predictable.

Module resolution

While the ECMAScript specification defines the parsing and interpretation of import and export statements, it leaves the details of module resolution to the host system. To ensure that TypeScript interprets these statements in a way that is compatible with Node.js, you should set the `moduleResolution` option in your `tsconfig.json` file. For Node.js, we recommend setting this option to `NodeNext`, which complements the `module` option and defaults to the same value if not specified. `NodeNext` is designed to support upcoming Node.js module resolution features.



Writing the perfect TypeScript configuration file for your project requires a decent understanding of your specific context and how the TypeScript compiler works. Hopefully, we have provided you with the basics here. If you are looking for ready-made configurations that you can use in different contexts, this repo can be a great resource to check out: nodejsdp.link/total-typescript-tsconfig

Summary

In this chapter, we explored the need for modules in JavaScript and the evolution of module systems in Node.js, focusing on CommonJS and ES modules. We covered key concepts like named and default exports, static and dynamic imports, and the module resolution algorithm. We also discussed the implications of circular dependencies, module loading phases, and how modules can modify others.

We then looked at the differences between CommonJS and ES modules, including interoperability challenges, strict mode, and missing references. Additionally, we addressed the impact of monkey patching on type safety in TypeScript projects.

We examined how to import JSON files in both module systems and workarounds for their limitations. Finally, we discussed how to leverage ES modules when using TypeScript.

With this knowledge, you're now equipped to use both ES modules and CommonJS effectively, laying the groundwork for the next chapter on asynchronous programming in JavaScript. Get ready to explore callbacks, events, and their patterns in depth!

3

Callbacks and Events

In *synchronous programming*, we think of code as a series of consecutive steps that work together to solve a specific problem. Each operation is blocking, meaning that one task must be completed before the next one can begin. This approach makes the code straightforward to read, understand, and debug.

On the other hand, *asynchronous programming* operates differently. Certain operations, like reading from a file or making a network request, run “in the background.” When we initiate an *asynchronous operation*, the next instruction executes immediately, even if the previous task hasn’t finished yet. In this context, we need a way to be notified when the asynchronous operation completes so that we can continue our execution flow with the operation’s result. The most fundamental way to handle this in Node.js is through **callbacks**—functions that are invoked by the runtime once an asynchronous operation is complete.

Callbacks are the foundational building blocks upon which all other asynchronous mechanisms are built. Without callbacks, we wouldn’t have **promises**, and, in turn, we wouldn’t have **async/await**. Additionally, **streams** and **events** also rely on callbacks. For this reason, understanding how callbacks work is crucial.

It’s easy to assume that callbacks and event emitters are somewhat outdated or deprecated in modern JavaScript and Node.js, but that’s far from the truth. Truly understanding callbacks and event emitters is essential to grasping how promises and async/await work under the hood. So, don’t skip this chapter — it’s your first important step for mastering asynchronous programming in JavaScript and Node.js!

In this chapter, you’ll dive deep into the Node.js Callback pattern and explore what it means to write asynchronous code in practice. We’ll cover conventions, patterns, and common pitfalls. By the end of this chapter, you’ll have a solid grasp of the basics of the Callback pattern.

You’ll also be introduced to the **Observer** pattern, which is closely related to the Callback pattern. The Observer pattern, represented by the `EventEmitter` in Node.js, uses callbacks to handle multiple, heterogeneous events and is one of the most widely used components in Node.js programming.

To summarize, this is what you will learn in this chapter:

- The Callback pattern, how it works, what conventions are used in Node.js, and how to deal with its most common pitfalls.
- The Observer pattern and how to implement it in Node.js using the `EventEmitter` class.
- Comparing the `EventEmitter` with callbacks and how to combine the two to get the best of both worlds.

The Callback pattern

Callbacks are the embodiment of the Reactor pattern's handlers (which we covered in *Chapter 1, The Node.js Platform*), a core component of the Node.js architecture.

Callbacks can be defined as functions triggered to handle the result of an operation — exactly what we need when working with asynchronous tasks. In an asynchronous context, callbacks take the place of the typical `return` statement, which only works synchronously. JavaScript is particularly well-suited for callbacks, as functions are first-class objects. This allows them to be easily assigned to variables, passed as arguments, returned from other functions, or stored in data structures.

Callbacks played a key role in shaping Node.js's distinctive programming style for many years. Despite their decline in popularity in modern user-facing code, they remain a fundamental and indispensable component of both the Node.js and JavaScript ecosystems. In fact, callbacks are the building block for constructing Promises and Async/Await, and are essential to understanding these higher-level abstractions and to using them correctly. Additionally, many advanced techniques that will be explored in this chapter and the following ones require a solid understanding of callbacks. This is why we have chosen to start our exploration of mastering asynchronous patterns from the perspective of callbacks in this book.

Closures also make an ideal partner for callbacks. They allow a function to retain access to the environment in which it was created, ensuring that the context of the asynchronous operation is preserved — no matter when or where the callback is eventually invoked.



If you need to refresh your knowledge about closures, you can refer to the article on MDN Web Docs at nodejsdp.link/mdn-closures.

In this section, we'll take a closer look at this programming style that relies on callbacks rather than traditional `return` statements.

The continuation-passing style

In JavaScript (and therefore in Node.js), a callback is a function that is passed as an argument to a function `f`. This callback function will be eventually called with the result of the operation when `f` completes. In functional programming, this way of propagating the result is called **continuation-passing style (CPS)**.

It is a general concept, and it is not always associated with asynchronous operations. In fact, it simply indicates that a result is propagated by passing it to another function (the callback), instead of directly returning it to the caller.

Synchronous CPS

To clarify the concept, let's start with a simple synchronous function:

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

Nothing fancy here: just a simple addition function that takes two numbers and returns their sum. The result is returned to the caller using a `return` statement. This is known as **direct style** and is the most common way to return results in synchronous programming in most programming languages (including JavaScript).

Now, here's the equivalent function using **Continuation-Passing Style (CPS)**:

```
function addCps(a, b, callback) {  
  callback(a + b)  
}
```

If we compare the `addCps()` function with the `add()` function described before we can see 2 main differences:

- `addCps()` expects an additional argument: a callback function
- rather than using a `return` statement, `addCps()` propagates the result of the operation by invoking the callback function and by passing the result of the operation (`a + b`) as an argument.

The `addCps()` function is a synchronous CPS function. It's still synchronous because nothing asynchronous is happening during its execution. Once `addCps()` is called, `a + b` is calculated and the `callback` is immediately invoked. Furthermore, `addCps()` completes its execution only when the callback does.

In more general terms, the idea of CPS is that when we call a function, we also pass a *callback* function. Once the computation is finished, the callback is invoked with the result. In a sense, we're telling the function to "pass the ball" to the callback when it's done.

In this particular example, since this operation is synchronous, the callback is called immediately. Let's clarify this further with an example:

```
console.log('before') // 1  
addCps(1, 2, result => console.log(`Result: ${result}`)) // 2  
console.log('after') // 3
```

The previous code will print the following:

```
before
Result: 3
after
```

Let's try to make sense of this output:

1. The first line of code is performing a synchronous call that logs the word `before` in the standard output.
2. In the second line we invoke `addCps()` and we pass `1`, `2` and a callback to it. The callback function is an arrow function defined inline. If we check again the implementation of `addCps()`, we can see that the callback function will be immediately invoked with the result of the addition between our first 2 arguments ($1 + 2 = 3$). The body of the callback function therefore logs the string `Result: 3`.
3. In the last line, we simply log the word `after`.

At this point, you might still feel not 100% convinced that everything that is happening here is synchronous. For example, you might be wondering: how do we know that `console.log()` is a synchronous function?

This is a legitimate question, and we could answer it by checking the documentation for this function, but, in general, as we explore more synchronous and asynchronous functions, we will learn to recognize common patterns and idioms and develop an instinct for when code is executed synchronously or asynchronously.

So, let's see how *asynchronous* CPS works.

Asynchronous CPS

Now, let's look at an alternative implementation of `addCps()` that behaves asynchronously. Let's call it `addAsync()`:

```
function addAsync(a, b, callback) {
  setTimeout(() => callback(a + b), 100)
}
```

In this example, we introduce an artificial delay using `setTimeout()` to simulate asynchronous behavior. The `setTimeout()` function schedules a task in the event queue to run after a specified number of milliseconds, making this an asynchronous operation. While this is a simple example meant to demonstrate the pattern, you can imagine real-world scenarios where the function might need to fetch external data, like a currency conversion function that retrieves the latest exchange rates before performing the conversion.



We have just learned that `setTimeout()` is an asynchronous function, and by using it, we have transformed our `addAsync()` function into an asynchronous one as well. Asynchronous functions are often constructed by using lower-level asynchronous functions like `setTimeout()`, `setInterval()`, `setImmediate()`, `process.nextTick()`, `fetch()`, `http.request()`, and so on. The documentation for these functions can reveal their asynchronous nature, but as a general rule, whenever we are dealing with timers or waiting for input/output operations to complete (such as reading from a file or making an HTTP request), we will encounter some form of asynchronous abstraction.

Let's try using `addAsync()` and observe how the order of operations changes:

```
console.log('before')
addAsync(1, 2, result => console.log(`Result: ${result}`))
console.log('after')
```

The preceding code will print the following:

```
before
after
Result: 3
```

Since `setTimeout()` triggers an asynchronous operation, it doesn't wait for the callback to be executed; instead, it returns immediately, giving the control back to `addAsync()`, and then back again to its caller. This property in Node.js is crucial, as it gives control back to the event loop as soon as an asynchronous request is sent, thus allowing a new event from the queue to be processed.

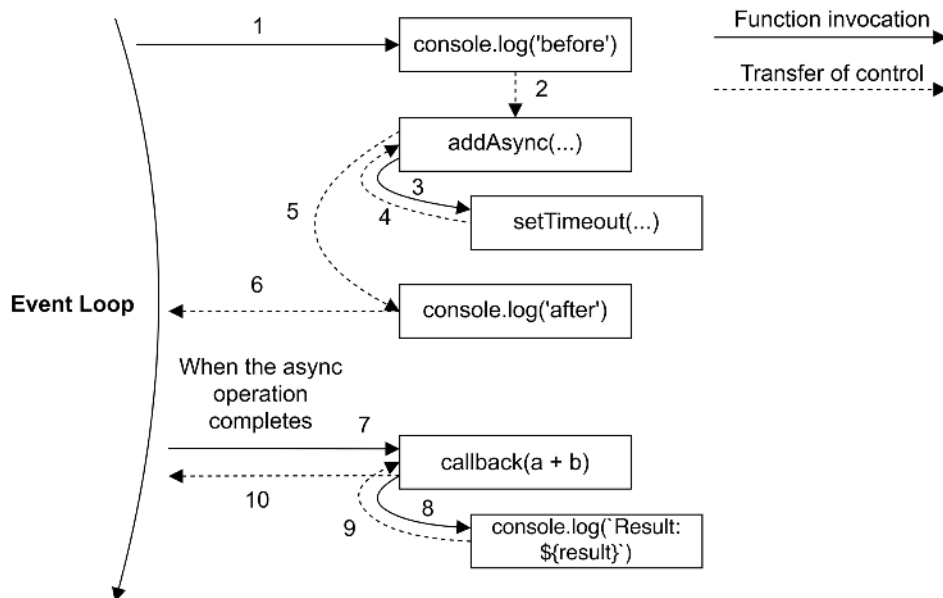


Figure 3.1: Control flow of an asynchronous function's invocation

Figure 3.1 shows step by step how this code is executed:

1. `console.log('before')` is executed: This logs the message before to the console.
2. `addAsync()` is executed: Control is transferred to the `addAsync()` function, where two arguments and a callback are passed.
3. `setTimeout()` is invoked: Within `addAsync()`, the `setTimeout()` function schedules the callback to execute after a specified delay (in this case, 100 milliseconds).
4. Control is returned from `setTimeout()` to `addAsync()`: Because `setTimeout()` is asynchronous, it doesn't wait for the delay to complete before returning control.
5. Control returns to the calling function: Since `setTimeout()` is the last (and only) instruction in `addAsync()`, `addAsync()` returns. Control goes to the next line of code, `console.log('after')` which is immediately executed printing after.
6. Control goes to the event loop: There are no more instructions to execute, so control goes back to the event loop. The event loop is aware that there are pending tasks (the timer associated to the `setTimeout()` call) so it's going to wait for the async operation to complete.
7. The asynchronous task is completed: When the timeout completes after 100 milliseconds, the callback passed to `setTimeout()` is invoked.
8. The callback is executed and it receives as argument the result of the expression `a + b`.
9. `console.log('Result: ${result}')` is invoked: This is the body of the provided callback function. We called the argument `result`, so this will print "Result: 3" (since `a` is 1 and `b` is 2) in the console.
10. Control returns to the event loop: After the callback finishes execution, control returns to the event loop, which is now ready to handle other pending tasks, or, if there are no other tasks, exit.

It's important to emphasize that when a callback is executed, it starts with a fresh call stack, and it starts from the event loop. This is where JavaScript truly excels. Thanks to closures, it's simple to retain the context of the original caller, even if the callback is invoked later or from a different part of the code. As we'll explore in the examples later in this chapter, this makes asynchronous CPS a highly effective programming style.



The call stack is the mechanism used by JavaScript's interpreter to keep track of the execution context. It makes it possible to know where the program currently is and what functions are being executed. When a function is called, it is pushed onto the call stack. If that function calls another function, it too is added to the stack. Once a function completes, it is removed from the stack, and the interpreter resumes execution at the previous level. When a callback function is executed via the event loop, it begins with a fresh call stack. This clean execution context is fundamental to JavaScript's asynchronous behavior. Closures make it possible to retain the original context, ensuring that data and variables remain accessible when the callback eventually runs. This ability to retain context makes closures indispensable for Continuation-Passing Style programming, where functions are passed as arguments and executed later, often asynchronously. Closures bridge the gap between the temporary nature of the call stack and the persistent data needed for callbacks.

Non-CPS callbacks

There are many situations where the presence of a callback might lead us to assume that a function is asynchronous or using CPS, but that's not always the case. Take the `map()` method of an array, for example:

```
const result = [1, 5, 7].map(element => element - 1)
console.log(result) // [0, 4, 6]
```

Here, the callback is simply used to iterate over the array elements, not to handle the result of the operation. In fact, the result is returned synchronously using direct style. There's no syntactic distinction between non-CPS callbacks and CPS ones, which is why the intended use of a callback should be clearly explained in the API documentation.

In the next section, we'll dive into one of the most common pitfalls with callbacks that every Node.js developer needs to be aware of.

Synchronous or asynchronous?

You have seen how the execution order of the instructions changes radically depending on the nature of a function—synchronous or asynchronous. This has strong repercussions on the flow of the entire application, both in terms of correctness and efficiency. The following is an analysis of these two paradigms and their pitfalls. In general, what must be avoided is creating inconsistency and confusion around the nature of an API, as doing so can lead to a set of problems that might be very hard to detect and reproduce. To drive our analysis, we will take, as an example, the case of an inconsistently asynchronous function.

Writing an inconsistent function

One of the most dangerous situations is to have an API that behaves synchronously under certain conditions and asynchronously under others.

Imagine a common scenario where we want to create an abstraction that fetches the content of a file from disk and caches it for faster future reads. Here's a potential (but flawed) implementation:

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

const cache = new Map()

function inconsistentRead(filename, cb) {
  if (cache.has(filename)) {
    // invoked synchronously
    cb(cache.get(filename))
  } else {
    // asynchronous function
    readFile(filename, 'utf8', (_err, data) => {
      cache.set(filename, data)
    })
  }
}
```

```
    cb(data)
  })
}
```

This function uses the `cache` variable to store file contents. It's a very basic example with a few intrinsic issues — it lacks error handling (e.g. what happens if we fail to read the file), and the caching logic could be significantly improved (in *Chapter 11, Advanced Recipes*, you'll learn how to handle asynchronous caching properly).

However, the real issue lies in its inconsistency: the function is asynchronous the first time a file is read but becomes synchronous once the file is cached. This unpredictable behavior can lead to unexpected problems, creating a recipe for all sorts of chaos! Let's discover how things can go terribly wrong with this approach!

Unleashing Zalgo

Now, let's look at how using an inconsistent function, one that may behave synchronously in some calls and asynchronously in others, such as the `inconsistentRead()` function defined earlier, can break an application in subtle, non-obvious ways. The following example illustrates one such unexpected side effect:

```
function createFileReader(filename) {
  const listeners = []
  inconsistentRead(filename, (value) => {
    for (const listener of listeners) {
      listener(value)
    }
  })

  return {
    onDataReady: listener => listeners.push(listener)
  }
}
```

When the preceding function is invoked, it creates a new object that acts as a notifier, allowing us to set multiple listeners for a file read operation. All the listeners will be invoked at once when the read operation completes, and the data is available. The preceding function uses our `inconsistentRead()` function to implement this functionality. Let's see how to use the `createFileReader()` function to read a `data.txt` file from the current working directory which contains the text `some data`:

```
const reader1 = createFileReader('data.txt')
reader1.onDataReady(data => {
  console.log(`First call data: ${data}`)

  // ...sometime later we try to read again from
```

```
// the same file
const reader2 = createFileReader('data.txt')
reader2.onDataReady(data => {
  console.log(`Second call data: ${data}`)
})
})
```

The preceding code will print the following text:

```
First call data: some data
```

... And nothing else! What happened to our second `console.log()`? Why don't we see that in our output? Let's review the code more carefully to find out why:

- During the creation of `reader1`, our `inconsistentRead()` function behaves asynchronously because there is no cached result available for `data.txt`. This means that any `onDataReady` listener will be invoked later in another cycle of the event loop, so we have all the time we need to register our listener.
- Then, `reader2` is created in a cycle of the event loop in which the cache for `data.txt` already exists. In this case, the inner call to `inconsistentRead()` will be synchronous. So, its callback will be invoked immediately, which means that all the listeners of `reader2` will be invoked synchronously as well. However, we are registering the listener after the creation of `reader2`, so it will never be invoked.

The callback behavior of our `inconsistentRead()` function is really unpredictable as it depends on many factors, such as the frequency of its invocation, the filename passed as an argument, and the amount of time taken to load the file.

This type of inconsistent behavior can lead to bugs that can be extremely complicated to identify and reproduce in a real application. Imagine using a similar function in a web server, where there can be multiple concurrent requests. Imagine seeing some of those requests hanging, without any apparent reason and without any error being logged.

Isaac Z. Schlueter, the creator of npm and former Node.js project lead, once compared using this kind of unpredictable function to *unleashing Zalgo* in one of his blog posts.

Zalgo is an internet legend about a terrifying entity said to bring chaos, insanity, and even the end of the world. If you haven't heard of Zalgo, I encourage you to look it up — it's definitely worth the read!



You can find Isaac Z. Schlueter's original post at nodejsdp.link/unleashing-zalgo.

Using synchronous APIs

The lesson we must learn from the unleashing Zalgo example is that it is imperative for an API to clearly define its nature: either synchronous or asynchronous — mixing the two can lead to all sorts of trouble.

One way to fix our `inconsistentRead()` function is by making it fully synchronous. This is doable because Node.js offers synchronous APIs for most basic I/O operations. For instance, we can use `readFileSync()` function instead of its asynchronous version. Here's the updated code:

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

const cache = new Map()

function consistentReadSync(filename) {
  if (cache.has(filename)) {
    return cache.get(filename)
  }

  const data = readFileSync(filename, 'utf8')
  cache.set(filename, data)
  return data
}
```

You'll notice the function now uses direct style, meaning it returns data directly instead of using callbacks. Since it's synchronous, there's no need for a callback. In fact, it's best practice to use direct style for synchronous APIs to keep things simple and clear.



Pattern

Always choose a direct style for purely synchronous functions



Keep in mind that switching an API from callback style (CPS) to direct style, or from asynchronous to synchronous (or vice versa), is a *breaking change*. This means any code using the API will also need to be updated.

However, using a synchronous API instead of an asynchronous one has some caveats:

- A synchronous API for a specific functionality might not always be available.
- A synchronous API will block the event loop and put any concurrent requests on hold. This conflicts with the Node.js concurrency model, slowing down the whole application. We will discuss this in greater detail in *Chapter 11, Advanced Recipes*.

In our `consistentReadSync()` function, the risk of blocking the event loop is partially mitigated because the synchronous I/O API is invoked only once per filename, while the cached value will be used for all the subsequent invocations. If we have a limited number of static files, then using `consistentReadSync()` won't have a big effect on our event loop. Things can change quickly if we have to read many files and only once.

Using synchronous I/O in Node.js is generally discouraged, but in some cases, it can be the easiest and most efficient option. If you're building a web server that handles many concurrent requests, it's important to use asynchronous APIs to avoid blocking the event loop, ensuring all requests are processed concurrently. On the other hand, if you're creating an automation script run from the command line and don't need concurrency, it's perfectly fine to use synchronous APIs—they can make the code simpler and easier to manage.

Sometimes, you might even use a mix of both. For example, you may need to load a configuration file before starting a web server. In this case, using a synchronous API to load the file makes sense, but your request-handling logic should use asynchronous, non-blocking code as much as possible.

Always evaluate your specific use case to choose the right approach.



Pattern

Use blocking APIs sparingly and only when they don't affect the ability of the application to handle concurrent asynchronous operations.

Guaranteeing asynchronicity with deferred execution

Another alternative for fixing our `inconsistentRead()` function is to make it purely asynchronous. The trick here is to schedule the synchronous callback invocation to be executed “in the future” instead of it being run immediately in the same event loop cycle. In Node.js, this is possible with `process.nextTick()`, which defers the execution of a function after the currently running operation completes. Its functionality is very simple: it takes a callback as an argument and pushes it to the top of the event queue, in front of any pending I/O event, and returns immediately. The callback will then be invoked as soon as the currently running operation yields control back to the event loop.

Let's apply this technique to fix the issues found in the `inconsistentRead()` function:

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

const cache = new Map()

function consistentReadAsync(filename, callback) {
  if (cache.has(filename)) {
    // deferred callback invocation
    process.nextTick(() => callback(cache.get(filename)))
  } else {
    // asynchronous function
    readFile(filename, 'utf8', (err, data) => {
      if (err) return callback(err)
      cache.set(filename, data)
      callback(null, data)
    })
  }
}
```

```

    readFile(filename, 'utf8', (_err, data) => {
      cache.set(filename, data)
      callback(data)
    })
  }
}

```

Now, thanks to `process.nextTick()`, our function is guaranteed to invoke its callback asynchronously, under any circumstances. Try to use it instead of the `inconsistentRead()` function and verify that, indeed, Zalgo has been eradicated.



Pattern

You can guarantee that a callback is invoked asynchronously by deferring its execution using `process.nextTick()`.

You might have seen another API for deferring code execution: `setImmediate()`. Although its name suggests that it runs callbacks immediately, this can be a bit misleading. In reality, `setImmediate()` gives callbacks lower priority than those scheduled with `process.nextTick()` or even `setTimeout(callback, 0)`. Yes, naming things is hard!

Callbacks deferred with `process.nextTick()` are called **microtasks** and they are executed just after the current operation completes, even before any other I/O event is fired. With `setImmediate()`, on the other hand, the execution is queued in an event loop phase that comes after all I/O events have been processed.

This means that callbacks scheduled with `process.nextTick()` will always run sooner, but in certain situations, like when using recursion, it can lead to **I/O starvation** — indefinitely delaying I/O callbacks. For example, if you recursively schedule callbacks inside a `process.nextTick()` callback, more callbacks will keep piling up in the queue. Since the event loop prioritizes this queue over other callbacks, it won't get a chance to process I/O event callbacks, like reading from a file. We can appreciate this concept in the following example:

```

import { readFile } from 'node:fs'

readFile('data.txt', 'utf8', (_err, data) => {
  console.log(`Data from file: ${data}`)
})

let scheduledNextTicks = 0
function recursiveNextTick() {
  if (scheduledNextTicks++ >= 1000) {
    return
  }
}

```

```
console.log('Keeping the event loop busy')
process.nextTick(() => recursiveNextTick())
}
recursiveNextTick()
```

In this example, we read the content of a file using `readFile()` and provide a callback to log the data once it's available. After that, we artificially keep the event loop busy by recursively scheduling 1000 tasks using `process.nextTick()`. Each task simply prints `Keeping the event loop busy` to the console.

If we run this code, the output will look like this:

```
Keeping the event loop busy
Keeping the event loop busy
Keeping the event loop busy
...
Data from file: some data
```

We've shortened the output, but in reality, you'll see `Keeping the event loop busy` printed 1000 times before the file's content appears! This demonstrates that while `process.nextTick()` is useful for running high-priority callbacks, it can become problematic if used too frequently.

We mentioned that another common way to schedule callbacks to run as soon as possible is `setImmediate(callback, 0)`, which behaves somewhat like `setImmediate()`. However, in typical scenarios, callbacks scheduled with `setImmediate()` are executed after those scheduled with `setTimeout(callback, 0)`.

To understand why, we need to look at how the event loop handles callbacks in different phases. For the events we're discussing, timers (`setTimeout()`) are executed before I/O callbacks, which in turn are processed before `setImmediate()` callbacks. We can borrow (with slight modification) a great example from the official Node.js website (nodejsdp.link/next-tick):

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

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

process.nextTick(() => {
  console.log('process.nextTick(cb)')
})

console.log('Sync operation')
```

Take a moment to think about the output.

Ready? Here's what you'll see:

```
Sync operation
process.nextTick(cb)
setTimeout(cb, 0)
setImmediate(cb)
```

This gives us a clear picture of the event loop's order of priority for different types of events.



If you still feel confused about how JavaScript and Node.js work under the hood, or if you are wondering how the stack, the event loop and the different queues are used at different times of the execution of a program, we'd recommend checking out this awesome interactive simulator: nodejsdp.link/js-visualizer.

The differences between these APIs will become even clearer later in the book when we discuss running synchronous, CPU-bound tasks (*Chapter 11, Advanced Recipes*).

Next, we'll explore the conventions for defining callbacks in Node.js.

Node.js callback conventions

In Node.js, CPS APIs and callbacks follow a set of specific conventions. These conventions aren't just found in the core library, most third-party modules and applications follow them too. To design an async API that uses callbacks effectively, it's crucial you understand these guidelines and stick to them.

The callback is the last argument

In all core Node.js functions, the standard convention is that when a function accepts a callback as input, this must be passed as the last argument.

Let's take the following Node.js core API as an example:

```
readFile(filename, [options], callback)
```

As you can see from the signature of the preceding function, the callback is always put in the last position, even in the presence of optional arguments such as options in this example. The reason for this convention is that the function call is more readable in case the callback is defined in place.

Any error always comes first

In CPS, errors are propagated like any other type of result, which means using callbacks. In Node.js, any error produced by a CPS function is always passed as the first argument of the callback, and any actual result is passed starting from the second argument. If the operation succeeds without errors, the first argument will be null or undefined.

The following code shows you how to define a callback that complies with this convention:

```
readFile('foo.txt', 'utf8', (err, data) => {  
  if (err) {  
    handleError(err)  
  } else {  
    processData(data)  
  }  
})
```



It is best practice to always check for the presence of an error, as not doing so will make it harder for you to debug your code and discover the possible points of failure. Another important convention to consider is that the error must always be an instance of the `Error` class. This means that simple strings or numbers should never be passed as error objects.

Propagating errors

Error propagation in synchronous, direct style functions is done with the well-known `throw` statement, which causes the error to jump up in the call stack until it is caught:

```
throw new Error('Something went wrong')
```

In asynchronous CPS, however, proper error propagation is done by simply passing the error to the next callback in the chain. The typical pattern looks as follows:

```
import { readFile } from 'node:fs'  
  
function readJson(filename, callback) {  
  readFile(filename, 'utf8', (err, data) => {  
    let parsed  
    if (err) {  
      // error reading the file  
      // propagate the error and exit the current function  
      return callback(err)  
    }  
  
    try {  
      // parse the file contents  
      parsed = JSON.parse(data)  
    } catch (err) {  
      // catch parsing errors  
      return callback(err)  
    }  
  })  
}
```

```
// no errors, propagate just the data
callback(null, parsed)
})
}
```

Notice how we don't throw or return errors. The first possible error can happen when we use the `readFile()` operation. Inside its callback, the first argument we receive is `err` and it represents an error related to reading from the given file. We do not throw it or return it; instead, we just invoke the callback with the error to propagate it back to the caller of `readJson()`. The `return` statement is used only to stop the function execution, not to return a specific value.

The next potential error that we need to deal with is when we call `JSON.parse()`. This is a synchronous function and therefore, if we try to parse invalid JSON, it will use the traditional `throw` instruction to propagate errors to the caller. This means that this time we need to use a `try/catch` block to capture errors. Again, on the catch branch we invoke `callback(err)` to propagate the error to the caller and we use `return` to stop the execution of the function. Finally, if everything went well, `callback` is invoked with `null` as the first argument to indicate that there are no errors.

It's also interesting to note how we refrained from invoking `callback` from within the `try` block. This is because doing so would catch any error thrown from the execution of the callback itself, which is usually not what we want.

Avoiding uncaught exceptions

Sometimes, it can happen that an error is thrown and not caught within the callback of an asynchronous function. This could happen if, for example, we forget to surround `JSON.parse()` with a `try...catch` and then our function happens to parse some invalid JSON at runtime.

Throwing an error inside an asynchronous callback would cause the error to jump up to the event loop, so it would never be propagated to the next callback. In Node.js, this is an unrecoverable state and the application would simply exit with a non-zero exit code, printing the stack trace to the `stderr` interface.

To see this in action, let's try to remove the `try...catch` block surrounding `JSON.parse()` from the `readJson()` function we defined previously:

```
function readJsonThrows(filename, callback) {
  readFile(filename, 'utf8', (err, data) => {
    if (err) {
      return callback(err)
    }
    callback(null, JSON.parse(data))
  })
}
```

Now, in the function we just defined, there is no way of catching an eventual exception coming from `JSON.parse()`. Let's try to parse an invalid JSON file with the following code:

```
readJsonThrows('invalid_json.json', (err) => console.error(err))
```

Although we have provided a callback that should be able to handle the errors, since the implementation of `readJsonThrows()` does not call the callback but just throws an error, this will result in the application being abruptly terminated, with a stack trace similar to the following being printed on the console:

```
SyntaxError: Unexpected token h in JSON at position 1
    at JSON.parse (<anonymous>)
    at file:///.../03-callbacks-and-events/10-uncaught-errors/index.js:8:25
    at FSReqCallback.readFileAfterClose [as oncomplete] (internal/fs/read/
context.js:68:3)
```

Now, if you look at the preceding stack trace from the bottom up, you will see that it starts from within the built-in `fs` module, and exactly from the point in which the native API has completed reading and returned its result back to the `fs.readFile()` function, via the event loop. This clearly shows that the exception traveled from our callback, up the call stack, and then straight into the event loop, where it was finally caught and thrown to the console.

This also means that wrapping the invocation of `readJsonThrows()` with a `try...catch` block will not help us to catch the error, because the stack in which the block operates is different from the one in which our callback is invoked. The following code shows the anti-pattern that was just described:

```
try {
  readJsonThrows('invalid_json.json', (err) => console.error(err))
} catch (err) {
  console.log('This will NOT catch the JSON parsing exception')
}
```

The preceding catch statement will never receive the JSON parsing error, as it will travel up the call stack in which the error was thrown, that is, in the event loop and not in the function that triggered the asynchronous operation.

As mentioned previously, the application will abort the moment an exception reaches the event loop. However, we still have the chance to perform some cleanup or logging before the application terminates. In fact, when an exception ends up in the event loop, Node.js will emit a special event called `uncaughtException`, just before exiting the process. The following code shows a sample use case:

```
process.on('uncaughtException', (err) => {
  console.error(`This will catch at last the JSON parsing exception: `+
    `${err.message}`)
  // Terminates the application with 1 (error) as exit code.
  // Without the following line, the application would continue
  process.exit(1)
})
```


It's important to understand that an uncaught exception leaves the application in a state that is not guaranteed to be consistent, which can lead to unforeseeable problems. For example, there might still be incomplete I/O requests running or closures might have become inconsistent. That's why it is always advised, especially in production, to never leave the application running after an uncaught exception is received. Instead, the process should exit immediately, optionally after having run some necessary cleanup tasks, and ideally, a supervising process should restart the application. This is also known as the **fail-fast** approach and it's the recommended practice in Node.js and it will be discussed in more detail in *Chapter 12, Scalability and Architectural Patterns*.

This concludes our gentle introduction to the Callback pattern. Now, it's time to meet the Observer pattern, which is another critical component of an event-driven platform such as Node.js.

The Observer pattern

Another important and fundamental pattern used in Node.js is the **Observer** pattern. Together with the Reactor pattern and callbacks, the Observer pattern is an absolute requirement for mastering the asynchronous world of Node.js.

The Observer pattern is the ideal solution for modeling the reactive nature of Node.js and a perfect complement for callbacks. Let's give a formal definition, as follows:

The Observer pattern defines an object (called subject) that can notify a set of observers (or listeners) when a change in its state occurs.

The main difference from the Callback pattern is that the subject can notify multiple observers, while a traditional CPS callback will usually propagate its result to only one listener, the callback.

The EventEmitter

In traditional object-oriented programming, the Observer pattern requires interfaces, concrete classes, and a hierarchy. In Node.js, all this becomes much simpler. The Observer pattern is already built into the core and is available through the `EventEmitter` class. The `EventEmitter` class allows us to register one or more functions as listeners, which will be invoked when a particular event type is fired. *Figure 3.2* visually explains this concept:

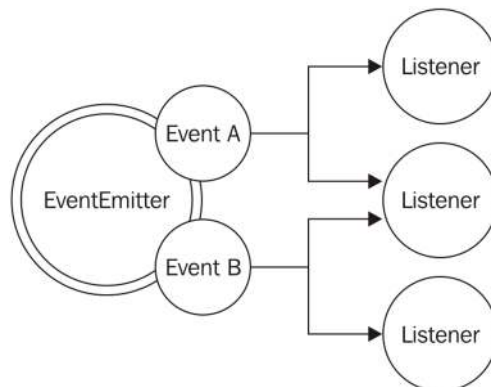


Figure 3.2: Listeners receiving events from an EventEmitter

The `EventEmitter` is exported from the `node:events` core module. The following code shows how we can obtain a reference to it:

```
import { EventEmitter } from 'node:events'
const emitter = new EventEmitter()
```

The essential methods of the `EventEmitter` are the following:

- `on(event, listener)`: This method allows us to register a new listener (a function) for the given event type (a string).
- `once(event, listener)`: This method registers a new listener, which is then removed after the event is emitted for the first time.
- `emit(event, [arg1], [...])`: This method produces a new event and provides additional arguments to be passed to the listeners.
- `removeListener(event, listener)`: This method removes a listener for the specified event type. Note that this method has also an alias: `off(event, listener)`.
- `removeAllListeners(event)`: Removes all the listeners registered for the given event.

All these methods will return the same `EventEmitter` instance to allow chaining. The listener function has the signature `function([arg1], [...])`, so it simply accepts the arguments provided at the moment the event is emitted.

You can already see that there is a big difference between a listener and a traditional Node.js callback. In fact, the first argument is not an error, but it can be any data passed to `emit()` at the moment of its invocation.

Creating and using the EventEmitter

Let's now see how we can use an `EventEmitter` in practice. The simplest way is to create a new instance and use it immediately. The following code shows us a function that uses an `EventEmitter` to notify its subscribers in real time when a particular regular expression (regex) is matched in a list of files:

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

function findRegex(files, regex) {
  const emitter = new EventEmitter()
  for (const file of files) {
    readFile(file, 'utf8', (err, content) => {
      if (err) {
        return emitter.emit('error', err)
      }

      emitter.emit('fileread', file)
      const match = content.match(regex)
    })
  }
}
```

```

    if (match) {
      for (const elem of match) {
        emitter.emit('found', file, elem)
      }
    }
  })
}
return emitter
}

```

The function we just defined returns an `EventEmitter` instance that will produce three events:

- `fileread`, when a file is being read
- `found`, when a match has been found
- `error`, when an error occurs during reading the file

Let's now see how our `findRegex()` function can be used:

```

findRegex(
  ['fileA.txt', 'fileB.json'],
  /hello [\w.]+/
)
.on('fileread', file => console.log(`${file} was read`))
.on('found', (file, match) => console.log(`Matched "${match}" in ${file}`))
.on('error', err => console.error(`Error emitted ${err.message}`))

```

In the code we just defined, we register a listener for each of the three event types produced by the `EventEmitter` that was created by our `findRegex()` function.



We kept this example intentionally simple to show how to use the `EventEmitter` class, but there's room for improvement, especially for production use. One issue is that we're loading all files into memory, which could lead to problems if the files are large, potentially filling up memory and crashing the program. In *Chapter 6, Coding with Streams*, we'll cover techniques to handle these situations more reliably. Additionally, we're using a simple synchronous `for...of` loop to read each file, triggering a `readFile()` operation for every file at once. This approach can overwhelm the system if there are many files to process, potentially exceeding system limits on open file descriptors. A better solution would be to process the files in batches using the limited parallel execution pattern, which we'll discuss in *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*.

Propagating errors

As with callbacks, the `EventEmitter` can't just throw an exception when an error condition occurs. Instead, the convention is to emit a special event, called `error`, and pass an `Error` object as an argument. That's exactly what we are doing in the `findRegex()` function that we defined earlier.



The `EventEmitter` class treats the error event in a special way. It will automatically throw an exception and exit from the application if such an event is emitted and no associated listener is found. For this reason, it is recommended to always register a listener for the error event.

Making any object observable

In the Node.js world, the `EventEmitter` is rarely used on its own, as you saw in the previous example. Instead, it is more common to see it extended by other classes. In practice, this enables any class to inherit the capabilities of the `EventEmitter`, hence becoming an observable object.

To demonstrate this pattern, let's try to implement the functionality of the `findRegex()` function in a class, as follows:

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

class FindRegex extends EventEmitter {
  constructor(regex) {
    super()
    this.regex = regex
    this.files = []
  }

  addFile(file) {
    this.files.push(file)
    return this
  }

  find() {
    for (const file of this.files) {
      readFile(file, 'utf8', (err, content) => {
        if (err) {
          return this.emit('error', err)
        }

        this.emit('fileread', file)

        const match = content.match(this.regex)
        if (match) {
          for (const elem of match) {
            this.emit('found', file, elem)
          }
        }
      })
    }
  }
}
```

```

    }
  })
}
return this
}
}

```

This code defines a `FindRegex` class that extends the `EventEmitter` class from the `node:events` module. This class has a constructor that uses `super()` to initialize the `EventEmitter` internals. It also takes in a `regex` argument and creates an array of files. The class has two methods: `addFile()` and `find()`. The `addFile()` method adds a file to the files array and returns the current instance of the `FindRegex` class to allow chained calls. The `find()` method reads each file in the files array, uses the `readFile()` function to read the contents of the file, and then uses the `match()` function to search for matches in the content using the `regex` passed as an argument. If a match is found, it emits the `found` event with the file path and the matched text.

The following is an example of how to use the `FindRegex` class we just defined:

```

const findRegexInstance = new FindRegex(/hello [\w.]+/)
findRegexInstance
  .addFile('fileA.txt')
  .addFile('fileB.json')
  .find()
  .on('found', (file, match) =>
    console.log(`Matched "${match}" in file ${file}`))
  .on('error', err => console.error(`Error emitted ${err.message}`))

```

You will now notice how the `FindRegex` object also provides the `on()` method, which is inherited from the `EventEmitter`. This is a common pattern in the Node.js ecosystem. For example, the `Server` object of the core HTTP module inherits from the `EventEmitter` function, thus allowing it to produce events such as `request` (when a new request is received), `connection` (when a new connection is established), or `closed` (when the server socket is closed).

Other notable examples of objects extending the `EventEmitter` are Node.js streams. We will analyze streams in more detail in *Chapter 6, Coding with Streams*.

The risk of memory leaks

A memory leak is a software defect whereby memory that is no longer needed is not released, causing the memory usage of an application to grow indefinitely.

When subscribing to observables with a long life span, it is extremely important that we **unsubscribe** our listeners once they are no longer needed. This allows us to release the memory used by the objects in a listener's scope and prevent **memory leaks**. Unreleased `EventEmitter` listeners are the main source of memory leaks in Node.js (and JavaScript in general).

For example, consider the following code:

```
const thisTakesMemory = 'A big string....'  
const listener = () => {  
  console.log(thisTakesMemory)  
}  
emitter.on('an_event', listener)
```

The variable `thisTakesMemory` is referenced in the listener and therefore its memory is retained until the listener is released from emitter, or until the emitter itself is garbage collected, which can only happen when there are no more active references to it, making it unreachable.



You can find a good explanation about garbage collection in JavaScript and the concept of reachability at nodejsdp.link/garbage-collection.

This means that if an `EventEmitter` remains reachable for the entire duration of the application, all its listeners do too, and with them all the memory they reference. If, for example, we register a listener to a “permanent” `EventEmitter` at every incoming HTTP request and never release it, then we are causing a memory leak. The memory used by the application will grow indefinitely, sometimes slowly, sometimes faster, but eventually it will crash the application. This is something that, under specific circumstances, an attacker could be able to exploit to trigger a Denial of Service (DoS) attack. To prevent such a dangerous situation, we can release the listener with the `removeListener()` method of the `EventEmitter`:

```
emitter.removeListener('an_event', listener)
```

An `EventEmitter` has a very simple built-in mechanism for warning the developer about possible memory leaks. When the count of listeners registered to an event exceeds a specific amount (by default, 10), the `EventEmitter` will produce a warning. Sometimes, registering more than 10 listeners is completely fine, so we can adjust this limit by using the `setMaxListeners()` method of the `EventEmitter`.



When we only want to listen to an event once, we can use the convenience method `once(event, listener)` in place of `on(event, listener)` to automatically unregister a listener after the event is received for the first time. However, be aware that if the event we specify is never emitted, then the listener is never released, causing a memory leak.

Synchronous and asynchronous events

As with callbacks, events can also be emitted synchronously or asynchronously with respect to the moment the tasks that produce them are triggered. It is crucial that we never mix the two approaches in the same `EventEmitter`, but even more importantly, we should never emit the same event type using a mix of synchronous and asynchronous code, to avoid producing the same problems described in the *Unleashing Zalgo* section. The main difference between emitting synchronous and asynchronous events lies in the way listeners can be registered.

When events are emitted asynchronously, we can register new listeners even after the task that produces the events is triggered, up until the current stack yields to the event loop. This is because the events are guaranteed not to be fired until the next cycle of the event loop, so we can be sure that we won't miss any events.

The `FindRegex()` class we defined previously emits its events asynchronously after the `find()` method is invoked. This is the reason why we can register the listeners *after* the `find()` method is invoked, without losing any events, as shown in the following code:

```
findRegexInstance
  .addFile(...)
  .find()
  .on('found', ...)
```

On the other hand, if we emit our events synchronously after the task is launched, we have to register all the listeners *before* we launch the task, or we will miss all the events. To see how this works, let's modify the `FindRegex` class we defined previously and make the `find()` method synchronous:

```
find() {
  for (const file of this.files) {
    let content
    try {
      content = readFileSync(file, 'utf8')
    } catch (err) {
      this.emit('error', err)
    }

    this.emit('fileread', file)
    const match = content.match(this.regex)
    if (match) {
      for (const elem of match) {
        this.emit('found', file, elem)
      }
    }
  }
}
```

```
    return this
  }
```

Now, let's try to register a listener before we launch the `find()` task, and then a second listener after that to see what happens:

```
const findRegexSyncInstance = new FindRegexSync(/hello [\w.]+/)
findRegexSyncInstance
  .addFile('fileA.txt')
  .addFile('fileB.json')
  // this listener is invoked
  .on('found', (file, match) =>
    console.log(`[Before] Matched "${match}"`))
  .find()
  // this listener is never invoked
  .on('found', (file, match) => console.log(`[After] Matched "${match}"`))
```

As expected, the listener that was registered after the invocation of the `find()` task is never called; in fact, the preceding code will print:

```
[Before] Matched "hello world"
[Before] Matched "hello Node.js"
```

There are some (rare) situations in which emitting an event in a synchronous fashion makes sense, but the very nature of the `EventEmitter` lies in its ability to deal with asynchronous events. Most of the time, emitting events synchronously is a telltale sign that we either don't need the `EventEmitter` at all or that, somewhere else, the same observable is emitting another event asynchronously, potentially causing a `Zalgo` type of situation.



The emission of synchronous events can be deferred with `process.nextTick()` to guarantee that they are emitted asynchronously.

EventEmitter versus callbacks

When designing an asynchronous API, a common question is whether to use an `EventEmitter` or a callback. A simple guideline is to use callbacks for returning results asynchronously, and events when you want to notify that something has happened, leaving it up to the consumer to decide whether to handle that event.

The confusion comes from the fact that both approaches are quite similar in practice and can often produce the same results. For instance, let's compare 2 different code examples.

The first one illustrates the events API:

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

function helloEvents () {
  const eventEmitter = new EventEmitter()
  setTimeout(() => eventEmitter.emit('complete', 'hello world'), 100)
  return eventEmitter
}

helloEvents().on('complete', message => console.log(message))
```

The second one illustrates the usage of callbacks:

```
function helloCallback (cb) {
  setTimeout(() => cb(null, 'hello world'), 100)
}

helloCallback((_err, message) => console.log(message))
```

The two functions `helloEvents()` and `helloCallback()` can be considered equivalent in terms of functionality. They both use `setTimeout()` to simulate a delay before producing a result (the string `hello world` in this case). They differ in the way they signal the completion of the timeout. `helloEvents()` uses an event, while `helloCallback()` relies on a callback. The key differences between these two approaches lie in readability, semantics, and the amount of code required to implement or use them.

While there are no strict rules for choosing one approach over the other, here are a few tips to guide your decision:

- Callbacks can be limiting when supporting different types of events. You can pass the event type as an argument to the callback or use multiple callbacks for each event, but this approach can result in a less elegant API. In such cases, an `EventEmitter` provides a cleaner interface and more concise code.
- Use `EventEmitter` when an event may happen multiple times or not at all. Callbacks are typically expected to run once, regardless of success or failure. If an event can recur, it's better treated as something to be communicated rather than a result to be returned.
- An `EventEmitter` allows multiple listeners to be registered for the same event. This flexibility can be useful in certain scenarios.

Combining callbacks and events

In some cases, you can use an `EventEmitter` alongside a callback. This pattern is very powerful and useful when you need to handle the final result of an asynchronous operation with a callback, but also want to emit progress events as the operation is ongoing. This approach gives you the best of both worlds, allowing for granular event tracking while still providing a way to propagate a final result.

A traditional example is downloading a file from a URL. We can create a function that uses a callback to signal when the download is complete. At the same time, the function can return an `EventEmitter` to track the download's progress. This allows us to display real-time updates, such as the percentage of completion, on the screen.

Before diving into the code, it's important to note that we'll be building an HTTPS client that follows a request/response flow. Here's a breakdown of the main steps:

1. **Client request:** The client sends a request to the server, specifying a method (in our case, GET) and a path (the part of the URL after the protocol and domain name). This is an asynchronous operation since multiple bytes are transmitted over the network gradually.
2. **Server response:** The server processes the request and sends back a response. This response typically has two parts: a response head (which includes the status code and headers) and a response body, which, in this case, will be the content of the file being downloaded. This is also an asynchronous operation.
3. **Closing the connection:** Once the response is fully received, the connection is closed on both ends, completing the flow. We also need to account for potential errors, such as failing to establish a connection, or the server dropping the response midway through the transfer.

Now let's see how we can implement this concept:

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

function download(url, cb) { // 1
  const emitter = new EventEmitter() // 2

  const req = get(url, resp => { // 3
    const chunks = [] // 4
    let downloadedBytes = 0
    const fileSize = Number.parseInt(resp.headers['content-length'], 10)
    resp
      .on('error', err => { // 5
        cb(err)
      })
      .on('data', chunk => { // 6
        chunks.push(chunk)
        downloadedBytes += chunk.length
        emitter.emit('progress', downloadedBytes, fileSize)
      })
      .on('end', () => { // 7
        const data = Buffer.concat(chunks)
        cb(null, data)
      })
  })
}
```

```
  })

  req.on('error', err => { // 8
    cb(err)
  })

  return EventEmitter // 9
}
```

There's a lot going on here, so let's walk through it step by step:

1. Our `download()` function takes a URL (`url`) to download from and a callback (`cb`) to handle the result once the download completes or if an error occurs.
2. It creates a new `EventEmitter` instance to emit events during the download process.
3. We use the Node's core module `node:https` to make an HTTPS request, specifically we use the `get` function (`const req = get(url, resp => { ... })`). This function sends an HTTPS request to the provided URL and starts processing the response once the server responds. It returns a request object (`req`), an `EventEmitter` that we can use to keep track of the progress of forwarding the request to the server. The callback receives a response object (`resp`), which is another `EventEmitter` that can notify us of various events related to the response that we are receiving from the server. This includes events such as errors (`'error'`), new data received (`'data'`), and completion (`'end'`).
4. The response object is created once the server successfully sends the response headers, allowing us to access those headers. At this point, we initialize the `chunks` array to store the bytes that will be received over time (in chunks), along with the `downloadedBytes` counter to track the total number of bytes downloaded. By reading the `'Content-Length'` header, we can determine the total number of bytes expected in the response.
5. We add a listener for the `'error'` event on the response object to catch any issues while receiving the response body. If an error occurs, we pass it back to the caller through the callback.
6. Progress tracking is done by using the `'data'` event listener which is triggered each time a chunk of data is received. Each chunk is pushed into the `chunks` array, and `downloadedBytes` is updated with the size of the received chunk. The `eventEmitter` emits a `'progress'` event with the current number of downloaded bytes and the total file size, which can be used to calculate and display download progress.
7. When the response body has been fully sent, the `'end'` event fires. In our event handler, all chunks are concatenated into a single `Buffer` using `Buffer.concat(chunks)`, which represents the complete file data. The callback `cb(null, data)` is also called to signal completion and provide the downloaded data back to the caller.

8. If an error occurs during the request phase, the `req.on('error', err => { ... })` listener is triggered, and the callback is called with the error. This allows us to capture and propagate errors that can happen during the request phase.
9. The function returns the `eventEmitter`, allowing the caller to listen for 'progress' events to track download.

Yes, that was quite a bit of detail, but now let's see how to use the new `download()` function:

```
download(
  'https://example.com/somefile.zip',
  (err, data) => {
    if (err) {
      return console.error(`Download failed: ${err.message}`)
    }
    console.log('Download completed', data)
  })
).on('progress', (downloaded, total) => {
  console.log(
    `${downloaded}/${total}` +
    `(${((downloaded / total) * 100).toFixed(2)}%)`
  )
})
```

Here's how it works:

- **Calling `download()`:** We provide the URL of the file to download and a callback function. This callback handles the final result of the download — either receiving the buffer with the downloaded data or an error.
- **Progress Event Listener:** We can also optionally attach a 'progress' event listener. This listener prints the amount of data downloaded so far and the percentage of completion, providing nice progress feedback for the use in case the download can take some time.

An example output showing real-time progress updates and the final result could look like this:

```
280/127454 (0.22%)
1649/127454 (1.29%)
...
126747/127454 (99.45%)
127454/127454 (100.00%)
Download completed <Buffer ff d8 ff e0 00 10 4a 46 49 46 00 01 01 00 00 01 00
01 00 00 ff db 00 84 00 04 04 04 04 04 04 04 04 06 06 05 06 06 08 07 07
07 07 08 0c 09 09 09 ... 127404 more bytes>
```



Please note that this implementation is for illustrative purposes and not production-ready. It is simplified to focus on demonstrating the pattern of combining a callback with an `EventEmitter` but lacks several important features. It doesn't handle chunked transfers, redirects, or plain HTTP requests, and it stores all data in a single buffer, which can lead to memory issues with large files. Error handling for HTTP response codes is also missing, and there's no support for download resumption or retries. For production scenarios, you might want to use a more comprehensive abstraction or library like `fetch` or third-party options such as `axios` (nodejsdp.link/axios) offer advanced features and better error handling, making them more suitable for real-world applications.

This example highlights the effectiveness of combining a callback with an `EventEmitter`. The key takeaway is that this pattern is especially useful for managing long-running tasks where you need to track both the completion and real-time progress. It's also applicable to various other practical scenarios, including file uploads, data processing pipelines, database backups and migrations, port scanning, brute-force applications, and more. Interestingly, the `get()` function from the `node:https` module leverages this pattern as well: it provides a callback for accessing the response while also returning an `EventEmitter` to monitor the ongoing request.



Although combining callbacks and event emitters is common in the Node.js ecosystem, this pattern is being gradually replaced by combining promises and async iterators. We will dive into these modern approaches in *Chapter 9, Behavioral Design Patterns*. Additionally, you can combine event emitters with other asynchronous mechanisms like promises. For example, you can return an object containing both a promise and an `EventEmitter`, which can then be destructured by the caller, like this: `{ promise, events } = foo()`

Summary

In this chapter, we made our first contact with the practical aspects of writing asynchronous code. We discovered the two pillars of the entire Node.js asynchronous infrastructure—the callback and the `EventEmitter`—and we explored in detail their use cases, conventions, and patterns. We also explored some of the pitfalls of dealing with asynchronous code and we learned about the ways to avoid them. Mastering the content of this chapter paves the way toward learning the more advanced asynchronous techniques that will be presented throughout the rest of this book.

In the next chapter, we will learn how to deal with complex asynchronous control flows using callbacks.

Exercises

- **3.1 A simple event:** Modify the asynchronous `FindRegex` class so that it emits an event when the find process starts, passing the input files list as an argument. Hint: beware of *Zalgo*!
- **3.2 Ticker:** Write a function that accepts a number and a callback as the arguments. The function will return an `EventEmitter` that emits an event called `tick` every 50 milliseconds until the number of milliseconds is passed from the invocation of the function. The function will also call the callback when the number of milliseconds has passed, providing, as the result, the total count of tick events emitted. Hint: you can use `setTimeout()` to schedule another `setTimeout()` recursively.
- **3.3 A simple modification:** Modify the function created in exercise 3.2 so that it emits a tick event immediately after the function is invoked.
- **3.4 Playing with errors:** Modify the function created in exercise 3.3 so that it produces an error if the timestamp at the moment of a tick (including the initial one that we added as part of exercise 3.3) is divisible by 5. Propagate the error using both the callback and the event emitter. Hint: use `Date.now()` to get the timestamp and the remainder (%) operator to check whether the timestamp is divisible by 5.
- **3.5 Disk bloat finder:** Create a function that accepts the path to a folder on the local file system and identifies the largest file within that folder. For extra credit, enhance the function to search recursively through subfolders. Hint: you can use the `node:fs` module for this task, specifically the `stats()` function to determine if a path is a directory or file and to get the file size in bytes, and the `readdir()` function to list the contents of a directory.

4

Asynchronous Control Flow Patterns with Callbacks

Transitioning from a synchronous programming style to a platform like Node.js, where **continuation-passing style (CPS)** and asynchronous APIs are the standard, can be challenging. Asynchronous code makes it difficult to predict the order in which statements will execute. Simple tasks such as iterating through a set of files, executing tasks sequentially, or waiting for multiple operations to complete require developers to adopt new approaches and techniques to avoid writing inefficient and hard-to-read code.

When using callbacks to handle asynchronous control flow, the most common mistake is to fall into the trap of “callback hell,” where code grows horizontally with excessive nesting, making even simple routines difficult to read and maintain. However, by applying discipline and utilizing certain patterns, it’s possible to tame callbacks and write clean, manageable asynchronous code.

In this chapter, we will deep dive into callbacks, what’s good about them, what’s bad, and how to handle them effectively. Specifically, we’ll cover:

- The challenges of asynchronous programming.
- Best practices for avoiding callback hell and writing efficient asynchronous code.
- Common asynchronous patterns: Sequential execution (executing tasks one after another), sequential iteration (processing items in a sequence without parallelizing tasks), concurrent execution (running multiple tasks concurrently), limited concurrent execution (controlling the number of concurrent operations).

By mastering these concepts, you’ll be well on your way to writing efficient and readable asynchronous code.

The challenges of asynchronous programming

Asynchronous programming in JavaScript may seem straightforward, but it's not without its pitfalls. The use of closures and in-place definitions of anonymous functions allows for a smooth coding experience that doesn't require the developer to jump to other points in the codebase – an approach aligned with the **KISS principle** (**Keep It Simple, Stupid** or, as we prefer, “**Keep It Super Simple**”).



The KISS principle, is a friendly reminder to prioritize simplicity in software design. Coined by engineer Kelly Johnson at Lockheed, the idea was that things should be easy to fix, even with limited tools. For us developers, it means writing clear, understandable code over complex solutions. Aim for readability, use clear names, and stick to well-known patterns. This not only reduces bugs but also makes teamwork smoother and code easier to maintain.

However, this simplicity often comes at the cost of modularity, reusability, and maintainability. The risk is to end up with code that is made up of multiple nested callbacks, large functions and, overall, poor code organization.

While there's no single “perfect” solution for dealing with the challenges of asynchronous code, we will equip you with the skills to evaluate tradeoffs and choose approaches that balance simplicity with maintainability. We'll learn how to recognize potential issues before they arise, anticipate problems, and implement solutions that promote modularity, reusability, and maintainability. By mastering these concepts, you'll be able to write efficient, readable, and well-structured asynchronous code that meets the demands of modern JavaScript development – not by eliminating all trade-offs but by understanding how to make informed decisions about when to prioritize simplicity over complexity.

Creating a simple web spider

To work on a practical and realistic example that can illustrate some of the challenges of asynchronous programming, we will create a little web spider: a command-line application that takes in a web URL as input and downloads its contents locally into one or more files. This kind of software is something that can be useful if you want to be able to browse a website offline or to take a snapshot of it at a certain point in time.

I had to build something pretty similar once for work! I needed a program that could go through a website and find any links that were broken (404 errors). Using asynchronous programming was the key to handle the scale of the sites it was used on, often composed by hundreds or even thousands of pages.

These kinds of programs are really common when you're working with websites, and the web spider we are going to build will be a great starting point for your own projects. You can easily change how it works or add new features if you need to build something similar. For example, you might want to make a program that can crawl and extract structured information from multiple sites concurrently (i.e. a *web scraper*).

In our implementation, we will often refer to a local module named `./utils.js`, which contains some helpers that we will be using in our application. We will omit the contents of this file for brevity, but you can find the full implementation, along with a `package.json` file containing the full list of dependencies, in the official repository at nodejsdp.link/repo4.

Our application's core functionality is contained within the `spider.js` module. Let's take a look inside.

First, we import the necessary dependencies:

```
import { writeFile } from 'node:fs'
import { dirname } from 'node:path'
import { exists, get, recursiveMkdir, urlToFilename } from './utils.js'
```

Here's a brief explanation of each utility function imported from `./utils.js`:

- `exists`: A callback-based function that checks if a file exists in the filesystem. It takes a file path and invokes the callback with a boolean value indicating whether the file exists.
- `get`: A callback-based function that retrieves the body of an HTTP response for a given URL. It takes the URL as input and returns a Buffer representing the response body.
- `recursiveMkdir`: Another callback-based function that creates all necessary directories recursively within a specified path.
- `urlToFilename`: A synchronous function that converts a URL into a valid file system path.

Next, we define a new function named `spider()`, which takes two parameters: the URL to download and a callback function invoked when the download process completes:

```
export function spider(url, cb) {
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => { // 1
    if (err) { // 1.1
      cb(err)
    } else if (alreadyExists) { // 1.2
      cb(null, filename, false)
    } else { // 1.3
      console.log(`Downloading ${url} into ${filename}`)
      get(url, (err, content) => { // 2
        if (err) {
          cb(err)
        } else {
          recursiveMkdir(dirname(filename), err => { // 3
            if (err) {
              cb(err)
            } else {
              writeFile(filename, content, err => { // 4
                if (err) {
```



```
if (downloaded) {  
  console.log(`Completed the download of "${filename}"`)  
} else {  
  console.log(`${filename} was already downloaded`)  
}  
})
```

If you've copied the `utils.js` and `package.json` files from our code samples repository, run `npm install` to download all necessary dependencies. You're now ready to test your web spider application.

To run it, navigate to your project directory in your terminal or command prompt and execute:

```
node spider-cli.js https://www.nodejsdesignpatterns.com/
```

This should create a file called `www.nodejsdesignpatterns.com/index.html` starting from the current working directory. Feel free to check the content of this file, compare it with the source HTML code of the remote website, and to play around with other URLs.



Note that our simple web spider application requires that we always include the protocol (for example, `http://`) in the URL we provide. Also, do not expect HTML links to be rewritten or resources such as images, videos, scripts, and stylesheets to be downloaded. We are intentionally keeping this example lean, because we want to focus on demonstrating how asynchronous programming works. Feel free to try adding these extra features yourself as a fun challenge!

In the next section, you will learn how to improve the readability of this code and, in general, how to keep callback-based code as clean and readable as possible.

Callback hell

Take a closer look at the `spider()` function that we implemented in the previous section. You'll see that even with a simple algorithm, the code becomes deeply nested and hard to follow. This is a common issue with asynchronous code.

Now, imagine if we could use a blocking API instead. The code would be much simpler and easier to read. To give you a better idea, here's what the implementation could look assuming we had equivalent blocking APIs for all our helper functions:

```
export function spider(url) {  
  const filename = urlToFilename(url)  
  if (exists(filename)) {  
    return false  
  } else {  
    const content = get(url)  
    recursiveMkdir(dirname(filename))  
  }  
}
```

```
    writeFile(filename, content)
  }
}
```

Notice how we don't even need to handle errors explicitly; any synchronous exception will automatically propagate up the call stack.

However, things are different when it comes to using asynchronous CPS. Defining callbacks in place can quickly lead to messy and unreadable code.

This problem, where an overload of closures and inline callbacks turns the code into an unmanageable tangle, is commonly known as **callback hell**. It's one of the most notorious anti-patterns in Node.js and JavaScript development. Code caught in this trap typically looks something like this:

```
asyncFoo(err => {
  asyncBar(err => {
    asyncFooBar(err => {
      //...
    })
  })
})
```

You can see how code written in this way assumes the shape of a pyramid (pointing to the right) due to deep nesting, and that's why it is also colloquially known as the **pyramid of doom**.

The most obvious problem with code like this is its poor readability. The deep levels of nesting can make it really hard to tell where one function ends and another begins.

Another issue arises from overlapping variable names across different scopes. Often, we end up using similar, or even identical, names for variables in each scope. A common example is the error argument passed to each callback. Some developers try to differentiate these by using slight variations, like `err`, `error`, `err1`, `err2`, and so on. Others prefer to use the same name, such as `err`, across all scopes, shadowing the variable from the outer scope. Neither of these approaches is ideal, as both increase confusion and the risk of introducing bugs.

Additionally, closures come with some performance and memory overhead. They can also cause memory leaks, which aren't always easy to track down. In fact, any context referenced by an active closure is retained and won't be cleaned up by garbage collection.



Remember that, when a closure is created, it “closes over” its surrounding lexical environment, meaning it retains access to variables and data in that environment, even after the outer function has completed. This is the core of how closures work, but it also means that the referenced context cannot be garbage collected as long as the closure is still in use. In essence, any variable, object, or other data within the closure's scope will remain in memory.

If you take another look our `spider()` function, you'll see it's a prime example of callback hell, displaying all the issues we just discussed. Fortunately, we'll tackle these problems using the patterns and techniques covered in the upcoming sections of this chapter.

Callback best practices

Now that you've encountered your first example of callback hell, you know what to avoid. But managing asynchronous code comes with more challenges than just preventing deeply nested callbacks. In fact, there are many situations where controlling the flow of multiple asynchronous tasks requires specific patterns and techniques, especially if you're working with plain JavaScript without any external libraries.

For example, consider the common scenario where you have an array of URLs and you need to fetch the content of each URL **in sequence**, one after the other. In real life, one reason why you might want to do something like this sequentially could be due to rate-limiting restrictions (and prevent doing more than one request at the time). To solve this problem, you might be tempted to use a simple `forEach()` loop, like the following:

```
const urls = ['url1', 'url2', 'url3']
urls.forEach(url => {
  fetch(url, response => {
    console.log(`Fetched ${url}`)
    // ... process the response
  })
})
```

However, this code won't work as you might expect. The `forEach()` loop will execute synchronously, initiating all the `fetch` calls very quickly, one after the other. Because `fetch` is asynchronous, the callbacks will be executed at an unspecified point in the future without regard to the order in the array. This would lead to requests being made concurrently and to results that are not in order, with no clear control on execution.

To process these asynchronous operations sequentially, we need a mechanism that awaits the completion of each operation before moving on to the next. A common technique for this type of problem is to implement a pattern similar to recursion, which is something we will explore later in this chapter.

In this section, you'll not only learn how to steer clear of callback hell, but also how to implement some of the most common control flow patterns — using nothing but simple, plain JavaScript.

Implementing various control flows using callbacks can become cumbersome quickly if we are not careful. Before we dive into the details of how we can use callbacks to implement different control flows, let's take a short break to explore a few techniques that will help us write higher quality, more maintainable, callback-based code. Applying these techniques is key to making working with callbacks more manageable and enjoyable.

Callback discipline

When writing asynchronous code, the first rule is to avoid overusing in-place function definitions for callbacks. While it might seem convenient, as it saves you from worrying about modularity or reusability, you've seen how it often leads to more problems than it solves. In most cases, fixing callback hell doesn't require libraries, complex techniques, or a shift in paradigms — it just takes following some simple ideas.

Here are a few basic principles that can help reduce nesting and improve code organization overall:

- **Exit early:** Use `return`, `continue`, or `break` as needed to exit a statement immediately, instead of nesting full `if...else` blocks. This keeps your code shallow and easier to follow.
- **Use named functions for callbacks:** Move callbacks out of closures, and pass intermediate results as arguments. Named functions also provide clearer stack traces, which is helpful for debugging.
- **Modularize your code:** Break your code into smaller, reusable functions whenever possible to promote clarity and maintainability.

Now, let's apply these principles in practice.

Applying the callback discipline

To illustrate the effectiveness of the callback principles we mentioned, let's apply them to resolve the callback hell in our web spider application.

First, we can refactor our error-checking pattern by removing the `else` statement. This works because we can return from the function immediately when an error is detected. Instead of writing:

```
if (err) {  
  cb(err)  
} else {  
  // code to execute when there are no errors  
}
```

We can streamline the code like this:

```
if (err) {  
  return cb(err)  
}  
// code to execute when there are no errors
```

This is often referred to as the **early return principle**. With this simple trick, we immediately have a reduction in the nesting level of our functions. It is easy and doesn't require any complex refactoring.

A common mistake when executing the optimization just described is forgetting to terminate the function after the callback is invoked. For an error-handling scenario, the following code is a typical source of defects:

```
if (err) {
  cb(err)
}
// code to execute when there are no errors.
```



We should never forget that the execution of our function will continue even after we invoke the callback. It is then important to insert a `return` instruction to block the execution of the rest of the function. Also, note that it doesn't really matter what value is returned by the function; the real result (or error) is produced asynchronously and passed to the callback. The return value of the asynchronous function is usually ignored. This property allows us to write shortcuts such as the following:

```
return cb(...)
```

Otherwise, we'd have to write slightly more verbose code, such as the following:

```
cb(...)
return
```

As a second optimization for our `spider()` function, we can try to identify reusable pieces of code. For example, the functionality that writes a given string to a file (also making sure that the directory path is created if needed) can be easily factored out into a separate function, as follows:

```
function saveFile(filename, content, cb) {
  recursiveMkdir(dirname(filename), err => {
    if (err) {
      return cb(err)
    }
    writeFile(filename, content, cb)
  })
}
```

Following the same principle, we can create a generic function named `download()` that takes a URL and a filename as input, and downloads the URL into the given file. Internally, we can use the `saveFile()` function we created earlier:

```
function download(url, filename, cb) {
  console.log(`Downloading ${url} into ${filename}`)
  get(url, (err, content) => {
    if (err) {
      return cb(err)
    }
  })
}
```



```
    }
    saveFile(filename, content, err => {
      if (err) {
        return cb(err)
      }
      cb(null, content)
    })
  })
}
```

Now, our code is also more modular and explicit. To finish the refactoring, we modify the `spider()` function, which will now look like the following:

```
export function spider(url, cb) {
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    if (err) {
      return cb(err)
    }
    if (alreadyExists) {
      return cb(null, filename, false)
    }
    download(url, filename, err => {
      if (err) {
        return cb(err)
      }
      cb(null, filename, true)
    })
  })
}
```

The functionality and interface of the `spider()` function remain unchanged; what we've improved is how the code is organized. By applying the early return principle and other callback discipline techniques, we significantly reduced code nesting while increasing both reusability and testability. In fact, we could consider exporting both `saveFile()` and `download()` and moving them in our shared utility library to make them reusable across different modules. This would also allow us to test these functions in isolation, improving our code quality.

This refactoring shows that, more often than not, all it takes is some discipline to avoid overusing closures and anonymous functions. It's a simple, effective approach that requires minimal effort and doesn't rely on external libraries.

Now that you know how to write clean, asynchronous code with callbacks, we're ready to dive into common asynchronous patterns, such as sequential and concurrent execution.

Control flow patterns

In this section, we will look at asynchronous control flow patterns and start by analyzing the sequential execution flow.

Sequential execution

Executing a set of tasks in sequence means running them one at a time, one after the other. The order of execution matters and must be preserved, because the result of a task in the list may affect the execution of the next. *Figure 4.1* illustrates this concept:



Figure 4.1: An example of sequential execution flow with three tasks

There are different variations of this flow:

- Executing a set of known tasks in sequence, without propagating data across them.
- Using the output of a task as the input for the next (also known as *chain*, *pipeline*, or *waterfall*).
- Iterating over a collection while running an asynchronous task on each element, one after the other.

Sequential execution is usually the main cause of the callback hell problem when using asynchronous CPS.

Executing a known set of tasks in sequence

You may not have realized it, but in the previous section, we already delved into the concept of sequential execution flow while implementing the `spider()` function. If you take a moment to think about it, you'll see that our spider performs several asynchronous tasks in a specific order, with each task completing before the next begins: checking if the file already exists, downloading content from a remote URL, and saving that content to a file. By following some simple rules, we were able to organize these tasks into a sequential flow. Now, using that code as a reference, we can generalize our solution with the following pattern:

```
function task1(cb) {
  asyncOperation(() => {
    task2(cb) // calls task2 with the current callback
  })
}

function task2(cb) {
  asyncOperation(() => {
    task3(cb) // calls task3 with the current callback
  })
}
```

```
}

function task3(cb) {
  asyncOperation(() => {
    cb() // finally completes and executes the callback
  })
}

task1(() => {
  // executed when task1, task2, and task3 are completed
  console.log('tasks 1, 2, and 3 executed')
})
```

The pattern above demonstrates how each task triggers the next one upon the completion of a generic asynchronous operation. It highlights the modularization of tasks and shows that closures aren't always necessary for handling asynchronous code.

Sequential iteration

The sequential execution pattern works great when we know in advance which tasks need to be executed and how many there are, especially if the number of tasks is relatively small. This lets us hardcode the invocation of each subsequent task in the sequence. But what happens when we want to perform an asynchronous operation for every item in a collection? In situations like this, we can't hardcode the task sequence anymore; instead, we need to build it dynamically.

Web spider version 2

To illustrate sequential iteration, let's introduce a new feature to our web spider application: the ability to download all links on a web page recursively, as long as the links stay within the same domain. After all, it's a spider—it can crawl through the web, following links into its depths! To accomplish this, we'll extract all the links from the current page and have our web spider follow each one, recursively downloading them in sequence.

The first step is to modify our `spider()` function to trigger a recursive download of the page's links by using a new function called `spiderLinks()`, which we'll create shortly.

A key design decision here is ensuring the spider doesn't get stuck in an endless loop. To prevent this, we'll introduce a `maxDepth` parameter that limits the recursion depth. Here's the updated code for the `spider()` function:

```
export function spider(url, maxDepth, cb) {
  const filename = urlToFilename(url)

  exists(filename, (err, alreadyExists) => {
    if (err) {
      // error checking the file
    }
  })
}
```

```
    return cb(err)
  }

  if (alreadyExists) {
    if (!filename.endsWith('.html')) {
      // ignoring non-HTML resources
      return cb()
    }

    // If the page was already downloaded, read the contents and download
    // the links
    return readFile(filename, 'utf8', (err, fileContent) => {
      if (err) {
        // error reading the file
        return cb(err)
      }
      return spiderLinks(url, fileContent, maxDepth, cb)
    })
  }

  // The file does not exist, download it
  download(url, filename, (err, fileContent) => {
    if (err) {
      // error downloading the file
      return cb(err)
    }
    // if the file is an HTML file, spider it
    if (filename.endsWith('.html')) {
      return spiderLinks(url, fileContent.toString('utf8'),
        maxDepth, cb)
    }
    // otherwise, stop here
    return cb()
  })
})
}
```

We've added comments throughout the code, so it should be easy to follow what this new version does.

In the next section, we'll explore how the `spiderLinks()` function is implemented.

Sequential crawling of links

Now, we can build the core of this new version of our web spider: the `spiderLinks()` function. This function downloads all the links on an HTML page using a sequential asynchronous iteration algorithm. Take a close look at how we define it in the following code block:

```
function spiderLinks(currentUrl, body, maxDepth, cb) {  
  if (maxDepth === 0) { // 1  
    return process.nextTick(cb)  
  }  
  
  const links = getPageLinks(currentUrl, body) // 2  
  if (links.length === 0) {  
    return process.nextTick(cb)  
  }  
  
  function iterate(index) { // 3  
    if (index === links.length) {  
      return cb()  
    }  
  
    spider(links[index], maxDepth - 1, err => { // 4  
      if (err) {  
        return cb(err)  
      }  
      iterate(index + 1)  
    })  
  }  
  
  iterate(0) // 5  
}
```

Here are the key steps to understand in this new function:

1. Base case: If the `maxDepth` variable has reached 0, we stop crawling. Notice how we call the callback asynchronously to avoid the infamous Zalgo.
2. We extract all the links from the page using the `getPageLinks()` function, which is synchronous and returns only links pointing to the same hostname (this function is implemented in the `utils.js` file). If there are no links, we stop the process.
3. We use a local function called `iterate()` to go over the links. This function takes the index of the next link to process. The first thing it does is check if the index equals the length of the links array. If it does, it means we've processed all items, and we can stop by invoking `cb()`.

4. At this stage, everything is ready for processing the link. We call the `spider()` function, reducing the maximum recursion depth (`maxDepth - 1`), and continue with the next step of the iteration once the operation completes.
5. As the final step in `spiderLinks()`, we start the iteration by calling `iterate(0)`.

The algorithm we just introduced allows us to iterate over an array by executing an asynchronous operation sequentially, which, in our case, is the `spider()` function.

Now, let's update `spider-cli.js` to allow us to specify the nesting level as an additional command-line interface (CLI) argument:

```
import { spider } from './spider.js'

const url = process.argv[2]
const maxDepth = Number.parseInt(process.argv[3], 10) || 1

spider(url, maxDepth, err => {
  if (err) {
    console.error(err)
    process.exit(1)
  }

  console.log('Downloaded complete')
})
```

We can now test this new version of our spider application and watch as it downloads all the links on a web page recursively, one after the other. To stop the process—since it can take a while if there are many links—remember that you can always press *Ctrl* + *C*. If you want to resume later, you can restart the spider with the same URL you used in the first run.



Now that our web spider has the potential to download an entire website, please use it with care. For example, avoid setting a high max depth level or leaving the spider running for too long. Overloading a server with thousands of requests is not only impolite, but in some cases, it may even be illegal. For example, it could be considered a DoS (Denial of Service) attack. Spider responsibly!

The pattern

The code of the `spiderLinks()` function from the previous section is a clear example of how it's possible to iterate over a collection while applying an asynchronous operation. You may also notice that it's a pattern that can be adapted to any other situation where we need to iterate asynchronously over the elements of a collection or, in general, over a list of tasks. This pattern can be generalized as follows:

```
function iterate (index) {
  if (index === tasks.length) {
    return finish()
  }
  const task = tasks[index]
  task(() => iterate(index + 1))
}

function finish () {
  // iteration completed
}

iterate(0)
```



It's important to notice that these types of algorithms become recursive if `task()` is a synchronous operation. In such a case, the stack will not unwind at every cycle and there might be a risk of hitting the maximum call stack size limit.

The pattern that was just presented is very powerful and can be extended or adapted to address several common needs. Just to mention some examples:

- We can map the values of an array asynchronously.
- We can pass the results of an operation to the next one in the iteration to implement an asynchronous version of the reduce algorithm.
- We can quit the loop prematurely if a particular condition is met (asynchronous implementation of the `Array.some()` helper).
- We can even iterate over an infinite number of elements.

We could also choose to generalize the solution even further by wrapping it in a function with a signature such as the following:

```
iterateSeries(collection, iteratorCallback, finalCallback)
```

Here, `collection` is the actual dataset you want to iterate over, `iteratorCallback` is the function to execute over every item, and `finalCallback` is the function that gets executed when all the items are processed or in case of an error. The implementation of this helper function is left to you as an exercise.



The Sequential Iterator pattern

Enables the execution of tasks within a collection, one after the other, ensuring a controlled flow. This is achieved by using an `iterator` function that automatically triggers the next task in the sequence as soon as the current one completes.

In the next section, we will explore the concurrent execution pattern, which is more convenient when the order of the various tasks is not important.

Concurrent execution

There are some situations where the order of execution of a set of asynchronous tasks is not important and there is no logical correlation or any data dependency between these tasks. All we want is to be notified when all those running tasks are completed. Such situations are better handled using a concurrent execution flow.

At this point in the book, we have used the words **parallel** and **concurrent** a few times. They might sound similar, but the difference between them is an important concept to understand, especially in the context of Node.js.

As we discussed in *Chapter 1, The Node.js Platform*, Node.js is single-threaded, which makes understanding the concept of concurrency particularly relevant. Before we continue, let's try to define these two concepts with an analogy.

Imagine you run a restaurant and receive two orders at the same time. In the kitchen, there are two ways to handle this:

- **Parallelism:** If you have two chefs, each can prepare an order simultaneously, working completely independently. This is like having multiple CPU cores executing tasks in parallel.
- **Concurrency:** If there's only one chef, they need to divide their time efficiently. While waiting for water to boil for one dish, they can start chopping ingredients for the other. The chef isn't working on both tasks simultaneously but is making progress on both. This is how an event loop works, switching between tasks efficiently while waiting on slower operations like I/O.

Node.js is built on JavaScript, which runs on a single-threaded event loop. This means that, by default, Node.js excels at concurrent execution, handling multiple tasks without blocking the main thread.

While parallelism can be achieved using worker threads or by spawning multiple processes, concurrency is often more efficient and lightweight for many real-world applications, such as handling multiple network requests or database queries.

A parallel execution of multiple tasks might be represented as in *Figure 4.2*:

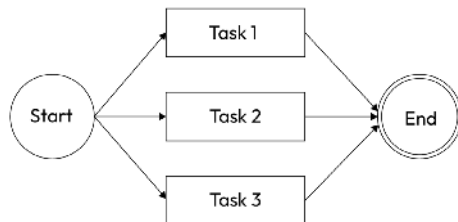


Figure 4.2: An example of parallel execution with three tasks

Although this picture is generally used to represent parallel execution of multiple tasks, it can also help describe concurrency at a high level, where multiple tasks are actively progressing within the event loop. The key distinction between parallel and concurrent execution lies in how progress is made: parallel execution runs tasks simultaneously, while concurrency involves efficiently switching between tasks to keep things moving.

So to further clarify our understanding, let's look at another diagram that shows how two asynchronous tasks can run concurrently in a Node.js program:

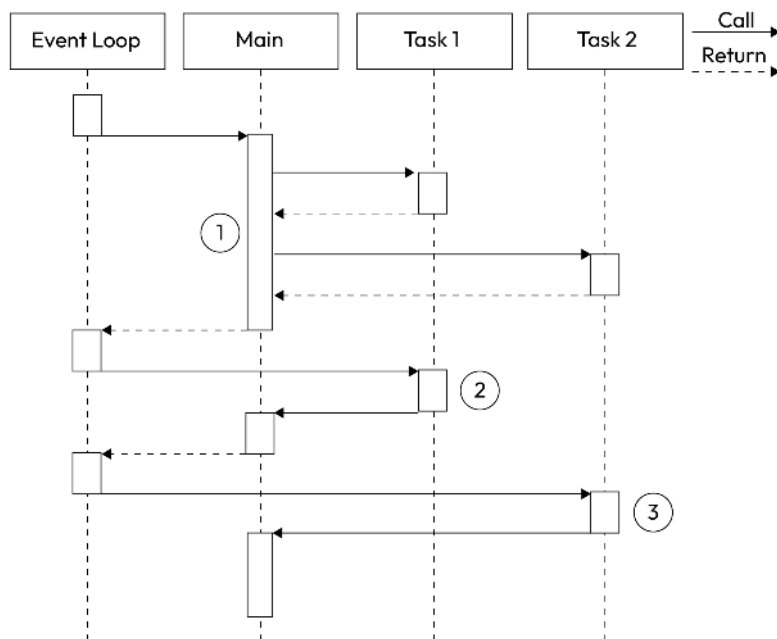


Figure 4.3: An example of how asynchronous tasks run concurrently

In *Figure 4.3*, we have a **Main** function that executes two asynchronous tasks:

1. The **Main** function triggers the execution of **Task 1** and **Task 2**. As they trigger an asynchronous operation, they immediately return control back to the **Main** function, which then returns it to the event loop.
2. When the asynchronous operation of **Task 1** is completed, the event loop gives control to it. When **Task 1** completes its internal synchronous processing as well, it notifies the **Main** function.
3. When the asynchronous operation triggered by **Task 2** is complete, the event loop invokes its callback, giving control back to **Task 2**. At the end of **Task 2**, the **Main** function is notified once more. At this point, the **Main** function knows that both **Task 1** and **Task 2** are complete, so it can continue its execution or return the results of the operations to another callback.

In short, this means that in Node.js, we generally execute asynchronous operations concurrently, because their concurrency is handled internally by the non-blocking APIs. In Node.js, synchronous (blocking) operations can't be easily parallelized or run concurrently unless their execution is interleaved with an asynchronous operation, or interleaved with `setTimeout()` or `setImmediate()`. You will see these techniques in more detail in *Chapter 11, Advanced Recipes*.

Web spider version 3

Our web spider application seems like a perfect candidate to apply the concept of concurrent execution. So far, our application is executing the recursive download of the linked pages in a sequential fashion. We can easily improve the performance of this process by downloading all the linked pages concurrently.

To do that, we just need to modify the `spiderLinks()` function to make sure we spawn all the `spider()` tasks at once, and then invoke the final callback only when all of them have completed their execution. So, let's modify our `spiderLinks()` function as follows:

```
function spiderLinks(currentUrl, body, maxDepth, cb) {
  if (maxDepth === 0) {
    return process.nextTick(cb)
  }

  const links = getPageLinks(currentUrl, body)
  if (links.length === 0) {
    return process.nextTick(cb)
  }

  let completed = 0
  let hasErrors = false // 3

  function done(err) { // 2
    if (err) {
      hasErrors = true
    }
  }
```

```

    return cb(err)
  }
  if (++completed === links.length && !hasErrors) {
    return cb()
  }
}

for (const link of links) { // 1
  spider(link, maxDepth - 1, done)
}
}

```

Let's discuss what we changed:

1. As mentioned earlier, the `spider()` tasks are now started all at once. This is possible by simply iterating over the `links` array and starting each task without waiting for the previous one to finish
2. Then, the trick to make our application wait for all the tasks to complete is to provide the `spider()` function with a special callback, which we call `done()`. The `done()` function increases a counter when a `spider` task completes. When the number of completed downloads reaches the size of the `links` array, the final callback (`cb`) is invoked.
3. The `hasErrors` variable is necessary because if one concurrent task fails, we want to immediately call the callback with the given error. Also, we need to make sure that other concurrent tasks that might still be running won't invoke the callback again.

With these changes in place, if we now try to run our spider against a web page, we will notice a huge improvement in the speed of the overall process, as every download will be carried out concurrently, without waiting for the previous link to be processed.

The pattern

Finally, we can extract our nice little pattern for the concurrent execution flow. Let's represent a generic version of the pattern with the following code:

```

const tasks = [ /* ... */ ]

let completed = 0
for (const task of tasks) {
  task(() => {
    if (++completed === tasks.length) {
      finish()
    }
  })
}

```

```
function finish () {  
  // all the tasks completed  
}
```

With small modifications, we can adapt the pattern to accumulate the results of each task into a collection, to filter or map the elements of an array, or to invoke the `finish()` callback as soon as one or a given number of tasks complete (this last situation in particular is called **competitive race**).



The Unlimited Concurrent Execution pattern

This pattern involves running a set of asynchronous tasks concurrently by launching them all at once and waiting for their completion. All tasks are started immediately, and completion is tracked by counting how many times their callbacks are invoked.

When we have multiple tasks running concurrently, we might have race conditions, that is, contention to access external resources (for example, files or records in a database). In the next section, we will talk about race conditions in Node.js and explore some techniques to identify and address them.

Fixing race conditions with concurrent tasks

Running a set of tasks in parallel can cause issues when using blocking I/O in combination with multiple threads. However, you have just seen that, in Node.js, this is a totally different story. Running multiple asynchronous tasks concurrently is, in fact, straightforward and cheap in terms of resources.

This is one of the most important strengths of Node.js, because it makes running multiple tasks through concurrency a common practice rather than a complex technique to only use when strictly necessary.

Another important characteristic of Node.js's concurrency model is how it handles task synchronization and race conditions. In a multithreaded environment, managing shared resources typically requires synchronization mechanisms such as locks, mutexes, semaphores, and monitors. These constructs help coordinate access to shared data but can introduce significant complexity and performance overhead.

To bring back our kitchen analogy, imagine two chefs working in parallel, each preparing their own dish. If they both need to use the same sink at the same time to wash ingredients, they can't proceed simultaneously; they must either take turns or risk getting in each other's way. In multithreaded programming, mechanisms like locks act as a way to manage this contention, ensuring that only one task accesses the shared resource at a time. However, poorly managed synchronization can lead to inefficiencies, such as one chef blocking the sink for too long, slowing down the entire kitchen.

In Node.js, we usually don't need a fancy synchronization mechanism, as everything runs on a single thread. However, this doesn't mean that we can't have race conditions; on the contrary, they can be quite common. The root of the problem is the delay between the invocation of an asynchronous operation and the notification of its result.



A race condition is a situation where the behaviour of a program depends on the timing of concurrent operations accessing shared resources.

To see a concrete example, we will refer again to our web spider application, and in particular, the last version we created, which contains a race condition. Can you spot it? Take a few minutes to think about it and see if you can guess what's the issue. We'll be here waiting for you!

The problem we are talking about lies in the `spider()` function, where we check whether a file already exists before we start to download the corresponding URL:

```
export function spider(url, maxDepth, cb) {
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    // ...
    if (alreadyExists) {
      // ...
    } else {
      download(url, filename, (err, fileContent) => {
        // ...
      })
    }
  })
}
```

The problem is that two spider tasks operating on the same URL might invoke `exists()` on the same file before one of the two tasks completes the download and creates a file, causing both tasks to start a download. *Figure 4.4* explains this situation:

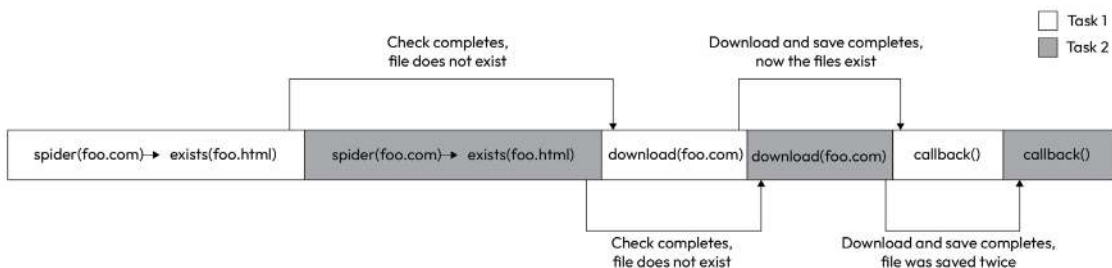


Figure 4.4: An example of a race condition in our `spider()` function

Figure 4.4 shows how **Task 1** and **Task 2** are interleaved in the single thread of Node.js, as well as how an asynchronous operation can actually introduce a race condition. In our case, the two spider tasks end up downloading the same file.

How can we fix that? The answer is much simpler than you might think. In fact, all we need is a variable to mutually exclude multiple `spider()` tasks running on the same URL. This can be achieved with some code, such as the following:

```
const spidering = new Set()
function spider (url, nesting, cb) {
  if (spidering.has(url)) {
    return process.nextTick(cb)
  }
  spidering.add(url)
  // ...
}
```

We simply exit the function right away if the URL is already in the global spidering set (I know *spidering* isn't technically a word, but it sounds cool, doesn't it?). Otherwise, we add the URL to the set and proceed with the download. Since we don't want to download a URL more than once, we don't need to remove URLs from the set.



If you are building a spider that might have to download hundreds of thousands of web pages, removing the downloaded url from the set once a file is downloaded will help you to keep the set cardinality, and therefore the memory consumption, from growing indefinitely.

Race conditions can cause all sorts of issues, even in a single-threaded environment like Node.js. In some cases, they can lead to data corruption and are notoriously difficult to debug due to their fleeting nature. That's why it's always a good idea to double-check for potential race conditions when running tasks concurrently.

Speaking of concurrent tasks, running too many concurrent tasks at once is usually a bad idea—you could quickly exhaust system memory or run into limits like the maximum number of open file descriptors. In the next section, we'll dive into why this can be a problem and how to manage the number of concurrent tasks effectively.

Limited concurrent execution

Spawning concurrent tasks without any control can easily lead to excessive load. Imagine trying to read thousands of files, access multiple URLs, or execute numerous database queries all at once. The most common issue in such cases is running out of resources. For example, an application might attempt to open too many files at the same time, quickly exhausting the available file descriptors.



A server that launches an unlimited number of concurrent tasks in response to user requests can also become vulnerable to a **denial-of-service (DoS)** attack. In this type of attack, a malicious actor can craft one or more requests that somehow force the server to use up all its resources and become unresponsive. Limiting the number of concurrent tasks is a good practice that helps build more resilient applications.

Currently, Version 3 of our web spider doesn't impose any limits on concurrent tasks, making it prone to crashing in several scenarios. For instance, if we run it against a large website, it might run for a few seconds before failing with an `ECONNREFUSED` error. This happens because, when too many pages are being downloaded concurrently, the web server might start rejecting new connections from the same IP. In such cases, our spider would crash, and we'd have to restart the process to continue crawling the site. While we could handle the `ECONNREFUSED` error to prevent the crash, we'd still risk overloading the system by allocating too many concurrent tasks, leading to other problems.

In this section, we'll explore how to make our spider more resilient by limiting concurrency.

The following diagram shows a scenario where we have 5 tasks to run concurrently, but we've set a concurrency limit of 2:

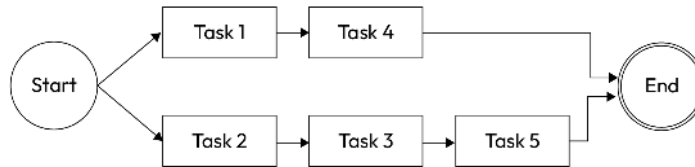


Figure 4.5: An example of how concurrency can be limited to a maximum of two tasks at the time

Looking at Figure 4.5, we can see that, with this particular example, the algorithm works like this:

- First, we start as many tasks as possible without going over the concurrency limit. Since the limit is set to two, the Start node launches **Task 1** and **Task 2**.
- As soon as a task finishes, we launch the next one in line, always keeping the limit in mind. For example, when **Task 2** finishes, **Task 3** is started. When **Task 1** finishes, **Task 4** is started. Once **Task 3** finishes, we move on to the final task, **Task 5**. When **Task 5** completes, and there are no more tasks to run, the entire process is done.

In the next section, we will explore a possible implementation of the limited concurrent execution pattern.

Limiting concurrency

We will now look at a pattern that will execute a set of given tasks concurrently with limited concurrency:

```

const tasks = [
  // ...
]

const concurrency = 2
let running = 0
let completed = 0
let nextTaskIndex = 0

function next() { // 1

```

```
while (running < concurrency && nextTaskIndex < tasks.length) {
  const task = tasks[nextTaskIndex++]
  task(() => { // 2
    if (++completed === tasks.length) {
      return finish()
    }
    running--
    next()
  })
  running++
}
next()

function finish() {
  // all the tasks completed
}
```

This algorithm can be considered a mixture of sequential execution and concurrent execution. In fact, you might notice similarities with both patterns:

1. We have a function called `next()` that acts as an iterator. Inside it, there's a loop that starts as many tasks as possible, while keeping the number of running tasks within the set concurrency limit.
2. Each time a task is started, we pass it a callback. In that callback, we decide what to do when the task completes. If there are more tasks left, we call `next()` again to keep things going. If all tasks are finished, we call `finish()` to signal that everything is done. Along the way, we make sure to properly update our counters: increasing the count of completed tasks and adjusting the number of running tasks (which gets increased each time `next()` starts a new task).

Pretty simple, isn't it?



In this pattern, we control concurrency by tracking how many tasks are running at any given time using the `running` variable. Since Node.js runs JavaScript in a single-threaded event loop, there's no risk of multiple tasks modifying `running` at the same time; each update happens in sequence, making synchronization unnecessary. As tasks start, `running` increases, and as they complete, it decreases before triggering `next()` to launch more tasks if needed. Because execution is always ordered within the event loop, `running` is always accurate, without the complexity of locks or race conditions. This is a key advantage of Node.js's concurrency model: efficient task management without the pitfalls of multithreading.

Globally limiting concurrency

Our web spider application is a great example of where we can apply what we've just learned about limiting the number of tasks running at once. Without such a limit, we might end up crawling thousands of links simultaneously, which could overwhelm our system. By controlling the number of concurrent downloads, we can add some much-needed predictability to the process.

Now, we could apply the limited concurrency pattern to our `spiderLinks()` function. However, this would only limit the number of tasks for the links found on a single page. For instance, if we set the concurrency limit to two, we'd have at most two links downloading at the same time *per page*. But since each page might spawn two more downloads, this could still cause the total number of downloads to grow quickly, potentially out of control.

In general, this particular implementation of the limited concurrency pattern works well when you have a predefined set of tasks, or when tasks increase at a manageable rate. However, in cases like our web spider—where a task can spawn multiple new tasks—it's not effective for limiting overall concurrency. To fix this, we need to introduce a mechanism that allows us to control concurrency on a global level.

Queues to the rescue

What we really want is to limit the total number of download operations running concurrently. A good way to achieve this is by introducing queues to manage the concurrency of multiple tasks. Let's see how this works.

We'll now implement a simple class called `TaskQueue`, which combines a queue with the limited concurrency algorithm we just discussed. Let's create a new module called `taskQueue.js`:

```
export class TaskQueue {
  constructor(concurrency) {
    this.concurrency = concurrency
    this.running = 0
    this.queue = []
  }

  pushTask(task) {
    this.queue.push(task)
    process.nextTick(this.next.bind(this))
    return this
  }

  next() {
    while (
      this.running < this.concurrency &&
      this.queue.length > 0
    ) {
      const task = this.queue.shift()
```

```
task() => {  
  this.running--  
  process.nextTick(this.next.bind(this))  
})  
this.running++  
}  
}  
}
```

The constructor of this class takes, as input, only the concurrency limit, but besides that, it initializes the instance variables `running` and `queue`. The former variable is a counter used to keep track of all the running tasks, while the latter is the array that will be used as a queue to store the pending tasks.

The `pushTask()` method simply adds a new task to the queue and then bootstraps the execution of the worker by asynchronously invoking `this.next()`.

The `next()` method spawns a set of tasks from the queue, ensuring that it does not exceed the concurrency limit.

You may notice that this method has some similarities with the pattern presented at the beginning of the *Limiting concurrency* section. It essentially starts as many tasks from the queue as possible, without exceeding the concurrency limit. When each task is complete, it updates the count of running tasks and then starts another round of tasks by asynchronously invoking `next()` again. The interesting property of the `TaskQueue` class is that it allows us to dynamically add new items to the queue. The other advantage is that, now, we have a central entity responsible for the limitation of the concurrency of our tasks, which can be shared across all the instances of a function's execution. In our case, it's the `spider()` function, as you will see in a moment.

Refining the TaskQueue

The previous implementation of `TaskQueue` is sufficient to demonstrate the queue pattern, but in order to be used in real-life projects, it needs a couple of extra features. For instance, how can we tell when one of the tasks has failed? How do we know whether all the work in the queue has been completed?

Let's bring back some of the concepts we discussed in *Chapter 3, Callbacks and Events*, and let's turn the `TaskQueue` into an `EventEmitter` so that we can emit events to propagate task failures and to inform any observer when the queue is empty.

The first change we have to make is to import the `EventEmitter` class and let our `TaskQueue` extend it:

```
import { EventEmitter } from 'node:events'  
  
export class TaskQueue extends EventEmitter {  
  constructor (concurrency) {  
    super()  
    // ...  
  }  
}
```

```
// ...  
}
```

At this point, we can use `this.emit` to fire events from within the `TaskQueue next()` method:

```
next () {  
  if (this.running === 0 && this.queue.length === 0) { // 1  
    return this.emit('empty')  
  }  
  
  while (this.running < this.concurrency && this.queue.length) {  
    const task = this.queue.shift()  
    task((err) => { // 2  
      if (err) {  
        this.emit('error', err)  
      }  
      this.running--  
      process.nextTick(this.next.bind(this))  
    })  
    this.running++  
  }  
}
```

Comparing this implementation with the previous one, there are two additions here:

1. Every time the `next()` function is called, we check that no task is running and whether the queue is empty. In such a case, it means that the queue has been drained and we can fire the empty event.
2. The completion callback of every task can now be invoked by passing an error. We check whether an error is actually passed, indicating that the task has failed, and in that case, we propagate such an error with an error event.

Notice that in case of an error, we are deliberately keeping the queue running. We are not stopping other tasks in progress, nor removing any pending tasks. This is quite common with queue-based systems. Errors are expected to happen and rather than letting the system crash on these occasions, it is generally better to identify errors and to think about retry or recovery strategies. We will discuss these concepts a bit more in *Chapter 13, Messaging and Integration Patterns*.



Hypothetically, if you wanted to gracefully halt all tasks upon encountering an error, you might consider introducing a mechanism that signals the `TaskQueue` to stop accepting and processing new tasks. Imagine a scenario where we add a `this.stopped` flag to the `TaskQueue` class. In case of an error, this flag is immediately set to `true`, and the queue is cleared. This would prevent any further tasks waiting in the queue from being initiated. We could also add logic to stop accepting new tasks (e.g. by throwing an error in the `pushTask()` method when `this.stopped` is `true`) from being pushed into the queue. While this approach would prevent the queue from starting any new tasks, it's important to note that it wouldn't be able to stop tasks that are already running or waiting on some asynchronous call to complete.

Web spider version 4

Now that we have our generic queue to execute tasks in a limited concurrent flow, let's use it straight-away to refactor our web spider application.

We are going to use an instance of `TaskQueue` as a work backlog; every URL that we want to crawl needs to be appended to the queue as a task. The starting URL will be added as the first task, then every other URL discovered during the crawling process will be added as well. The queue will manage all the scheduling for us, making sure that the number of tasks in progress (that is, the number of pages being downloaded or read from the filesystem) at any given time is never greater than the concurrency limit configured for the given `TaskQueue` instance.

We have already defined the logic to crawl a given URL inside our `spider()` function. We can consider this to be our generic crawling task. For more clarity, it's best to rename this function `spiderTask`:

```
function spiderTask(url, maxDepth, queue, cb) { // 1
  const filename = urlToFilename(url)
  exists(filename, (err, alreadyExists) => {
    if (err) {
      // error checking the file
      return cb(err)
    }
    if (alreadyExists) {
      if (!filename.endsWith('.html')) {
        // ignoring non-HTML resources
        return cb()
      }
    }
    return readFile(filename, 'utf8', (err, fileContent) => {
      if (err) {
        // error reading the file
        return cb(err)
      }
      spiderLinks(url, fileContent, maxDepth, queue) // 2
    })
  })
}
```

```

        return cb()
    })
}
// The file does not exist, download it
download(url, filename, (err, fileContent) => {
    if (err) {
        // error downloading the file
        return cb(err)
    }
    // if the file is an HTML file, spider it
    if (filename.endsWith('.html')) {
        spiderLinks(url, fileContent.toString('utf8'), maxDepth, queue) // 2
        return cb()
    }
    // otherwise, stop here
    return cb()
})
})
}

```

Other than renaming the function, you might have noticed that we applied some other small changes:

1. The function signature now accepts a new parameter called `queue`. This is an instance of `TaskQueue` that we need to carry over to be able to append new tasks when necessary.
2. The function responsible for adding new links to crawl is `spiderLinks()`, so we need to make sure that we pass the `queue` instance when we call this function after downloading a new page.

In just a moment, we'll see that the `spiderLinks()` function will be simplified to a synchronous function, as it will only need to append tasks to the queue. We won't need to pass the callback to it anymore; instead, we'll just call `return cb()` after invoking it.

Now, let's take a look at the `spiderLinks()` function. This function can be significantly simplified since it no longer has to track task completion—the queue now handles that. Its execution will effectively become synchronous; it simply needs to call the new `spider()` function (which we'll define shortly) to push a new task to the queue for each discovered link:

```

function spiderLinks(currentUrl, body, maxDepth, queue) {
    if (maxDepth === 0) {
        return
    }

    const links = getPageLinks(currentUrl, body)
    if (links.length === 0) {
        return
    }
}

```

```
    }

    for (const link of links) {
      spider(link, maxDepth - 1, queue)
    }
  }
}
```

Let's now revisit the `spider()` function, which needs to act as the *entry point* for the first URL; it will also be used to add every new discovered URL to the queue:

```
const spidering = new Set() // 1
export function spider(url, maxDepth, queue) {
  if (spidering.has(url)) {
    return
  }
  spidering.add(url)
  queue.pushTask(done => { // 2
    spiderTask(url, maxDepth, queue, done)
  })
}
```

As you can see, this function now has two main responsibilities:

1. It manages the bookkeeping of the URLs already visited or in progress by using the `spidering` set.
2. It pushes a new task to the queue. Once executed, this task will invoke the `spiderTask()` function, effectively starting the crawling of the given URL.

Finally, we can update the `spider-cli.js` script, which allows us to invoke our spider from the command line:

```
import { spider } from './spider.js'
import { TaskQueue } from './TaskQueue.js'

const url = process.argv[2] // 1
const maxDepth = Number.parseInt(process.argv[3], 10) || 1
const concurrency = Number.parseInt(process.argv[4], 10) || 2

const spiderQueue = new TaskQueue(concurrency) // 2
spiderQueue.on('error', console.error)
spiderQueue.on('empty', () => console.log('Download complete'))

spider(url, maxDepth, spiderQueue) // 3
```

This script is now composed of three main parts:

1. CLI arguments parsing. Note that the script now accepts a third additional parameter that can be used to customize the concurrency level.
2. A `TaskQueue` object is created, and listeners are attached to the error and empty events. When an error occurs, we simply want to print it. When the queue is empty, that means that we've finished crawling the website.
3. Finally, we start the crawling process by invoking the `spider` function.

After we have applied these changes, we can try to run the spider module again. When we run the following command:

```
node spider-cli.js https://loige.co 1 4
```

We should notice that no more than four downloads will be active at the same time.

With this final example, we've concluded our exploration of callback-based patterns.

Summary

At the start of this chapter, we mentioned that Node.js can be challenging due to its asynchronous nature, especially for developers coming from other platforms. However, as you've seen, you can make asynchronous APIs work to your advantage. The tools you've learned about are flexible and provide solid solutions to many common problems, while also allowing for different programming styles to suit individual preferences.

We've also continued refactoring and improving our web crawler example throughout the chapter. When working with asynchronous code, it can sometimes be tricky to find the right balance between simplicity and effectiveness, so take your time to digest the concepts covered and experiment with them.

Our journey with asynchronous Node.js programming is just beginning. In the next few chapters, you'll dive into other widely used techniques like promises and `async/await`. Once you're familiar with all these approaches, you'll be able to pick the best one for your needs—or even combine multiple techniques within the same project.

Before moving on, we highly recommend giving the exercises below a try to solidify your understanding. They'll help you practice the key concepts and prepare you for the chapters ahead.

Exercises

1. **File concatenation:** Write the implementation of `concatFiles()`, a callback-style function that takes two or more paths to text files in the filesystem and a destination file:

```
function concatFiles (srcFile1, srcFile2, srcFile3, ... ,
                      dest, cb) {
  // ...
}
```

This function must copy the contents of every source file into the destination file, respecting the order of the files, as provided by the arguments list. For instance, given two files, if the first file contains *foo* and the second file contains *bar*, the function should write *foobar* (and not *barfoo*) in the destination file. Note that the preceding example signature is not valid JavaScript syntax: you need to find a different way to handle an arbitrary number of arguments. For instance, you could use the **rest parameters** syntax (nodejsdp.link/rest-parameters).

2. **List files recursively:** Write `listNestedFiles()`, a callback-style function that takes, as the input, the path to a directory in the local filesystem and that asynchronously iterates over all the subdirectories to eventually return a list of all the files discovered. Here is what the signature of the function should look like:

```
function listNestedFiles (dir, cb) { /* ... */ }
```

Bonus points if you manage to avoid callback hell. Feel free to create additional helper functions if needed.

3. **Recursive find:** Write `recursiveFind()`, a callback-style function that takes a path to a directory in the local filesystem and a keyword, as per the following signature:

```
function recursiveFind(dir, keyword, cb) { /* ... */ }
```

The function must find all the text files within the given directory that contain the given keyword in the file contents. The list of matching files should be returned using the callback when the search is completed. If no matching file is found, the callback must be invoked with an empty array. As an example, test case, if you have the files `foo.txt`, `bar.txt`, and `baz.txt` in `myDir` and the keyword `'batman'` is contained in the files `foo.txt` and `baz.txt`, you should be able to run the following code:

```
recursiveFind('myDir', 'batman', console.log)
// should print ['foo.txt', 'baz.txt']
```

Bonus points if you make the search recursive (it looks for text files in any subdirectory as well). Extra bonus points if you manage to perform the search within different files and subdirectories concurrently but be careful to keep the number of concurrent tasks under control!

4. **Broken links checker:** Write a `checkBrokenLinks()` function that takes a URL and a maximum depth level and reports any broken links (links returning a 404 status code) it encounters during the crawling process. You could start from the code we wrote for the web crawler and modify it so that, instead of downloading the content of the pages, it only checks for the HTTP status of each link. If the status code is 404, it should log the URL of the broken link and continue crawling other links. Tip: You could try using the HEAD method instead of GET when checking for links, as it only fetches the headers, which can speed up the process and reduce bandwidth usage.

5

Asynchronous Control Flow Patterns with Promises and Async/Await

Callbacks are the low-level building blocks of asynchronous programming in Node.js and, as such, they are one of the most important concepts that we need to master, but they are far from being developer-friendly. No wonder that, in the last chapter, we spent a significant amount of time learning various techniques to enforce callback discipline and keep our code manageable. We also learned about different control flow constructs using callbacks, and we have to acknowledge that they can be complex and verbose to the point that we needed to create higher-level abstractions to keep this complexity under control and to simplify the reuse of these patterns. A particular mention goes to the serial execution flow, which is the predominant control flow structure in most of the code we write. Trying to implement this control flow can easily lead an untrained developer to write code affected by the callback hell problem. On top of that, even if properly implemented, a serial execution flow seems needlessly complicated and error-prone. Let's also remember how fragile error management with callbacks is; if we forget to forward an error, then it just gets lost, and if we forget to catch any exception thrown by some synchronous code, then the program crashes. And all of this without considering that *Zalgo* is always breathing down our necks.

Node.js and JavaScript have been criticized for years for the lack of a native solution to a problem so common and ubiquitous. Luckily, over the years, the community has worked on new solutions to the problem, and finally, after many iterations, discussions, and years of waiting, today we have a proper solution to the “callback issue.”

The first step toward a better asynchronous code experience is the **promise**, an object that “carries” the status and the eventual result of an asynchronous operation. A promise can be easily chained to implement serial execution flows and can be moved around like any other object. Promises simplify asynchronous code a lot; however, there was still room for improvement. So, in an attempt to make the ubiquitous serial execution flow as simple as possible, a new construct was introduced, called **async/await**, which can finally make asynchronous code look as simple and easy to reason about as synchronous code.

In today’s modern Node.js programming, **async/await** is the preferred construct to use when dealing with asynchronous code. However, **async/await** is built on top of promises, as much as promises are built on top of callbacks. So, it’s important that we know and master all of them in order. To truly grasp how **async/await** works and use it correctly, it’s essential to have a solid understanding of callbacks and promises first. Skipping over these fundamental concepts can lead to a gross misunderstanding of how **async/await** operates and result in writing buggy or inefficient code.

If you carefully walked through *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*, you shouldn’t fear callbacks anymore. Now, in this chapter, we will cover the following topics:

- How promises work and how to use them effectively to implement the main control flow constructs we already know about
- The **async/await** syntax, which will become our main tool for dealing with asynchronous code in Node.js
- Practical **async/await** patterns, common pitfalls, and how to avoid them.

By the end of the chapter, you will have learned about the two most important components that we have in JavaScript for taming asynchronous code. So, let’s get started by discovering promises.

Promises

Promises are part of the ECMAScript 2015 standard (or ES6, which is why they are also called ES6 promises) and have been natively available in Node.js since version 4. But the history of promises goes back a few years earlier, when there were dozens of implementations around, initially with different features and behavior. Eventually, the majority of those implementations settled on a standard called **Promises/A+**.

Promises constitute a big step toward providing a robust alternative to continuation-passing-style callbacks for propagating an asynchronous result. As we will see, the use of promises will make asynchronous control flow constructs easier to read, less verbose, and more robust compared to their callback-based alternatives.

What is a promise?

A promise is an object that represents the eventual result (or error) of an asynchronous operation. In promises jargon, we say that a **Promise** is **pending** when the asynchronous operation is not yet complete, it’s **fulfilled** when the operation successfully completes, and it’s **rejected** when the operation terminates with an error. Once a **Promise** is either fulfilled or rejected, it’s considered **settled**.

To receive the fulfillment **value** or the error (**reason**) associated with the rejection, we can use the `then()` method of a `Promise` instance. The following is its signature:

```
promise.then(onFulfilled, onRejected)
```

In the preceding signature, `onFulfilled` is a callback that will eventually receive the fulfillment value of the `Promise`, and `onRejected` is another callback that will receive the reason for the rejection (if any). Both are optional.

To have an idea of how promises can transform our code, let's consider the following callback-based code:

```
asyncOperation(arg, (err, result) => {  
  if(err) {  
    // handle the error  
  }  
  // do stuff with the result  
})
```

Promises allow us to transform this typical continuation-passing-style code into better-structured and more elegant code, such as the following:

```
asyncOperationPromise(arg)  
  .then(result => {  
    // do stuff with result  
  }, err => {  
    // handle the error  
  })
```

In the preceding code, `asyncOperationPromise()` is returning a `Promise`, which we can then use to receive the fulfillment value or the rejection reason of the eventual result of the function. So far, it seems that this is just a minor syntactical difference when we compare this code with the callback-based code, but one crucial property of the `then()` method is that it *synchronously* returns another `Promise`.

Moreover, if any of the `onFulfilled` or `onRejected` functions return a value *x*, the `Promise` returned by the `then()` method will do the following:

- Fulfill with *x* if *x* is a value
- Fulfill with the fulfillment value of *x* if *x* is a `Promise` in the fulfilled state
- Reject with the eventual rejection reason of *x* if *x* is a `Promise` in the rejected state

This behavior—the synchronous return of another Promise and the handling of return values—allows us to build *chains* of promises, allowing easy aggregation and arrangement of asynchronous operations into several configurations. Moreover, if we don't specify an `onFulfilled` or `onRejected` handler, the fulfillment value or rejection reason is automatically forwarded to the next promise in the chain. This allows us, for example, to automatically propagate errors across the whole chain until they are caught by an `onRejected` handler. With a Promise chain, the sequential execution of tasks suddenly becomes a simple operation:

```
asyncOperationPromise(arg)
  .then(result1 => {
    // returns another promise
    return asyncOperationPromise(arg2)
  })
  .then(result2 => {
    // returns a value
    return 'done'
  })
  .then(undefined, err => {
    // any error in the chain is caught here
  })
```

The following diagram provides another perspective on how a Promise chain works:

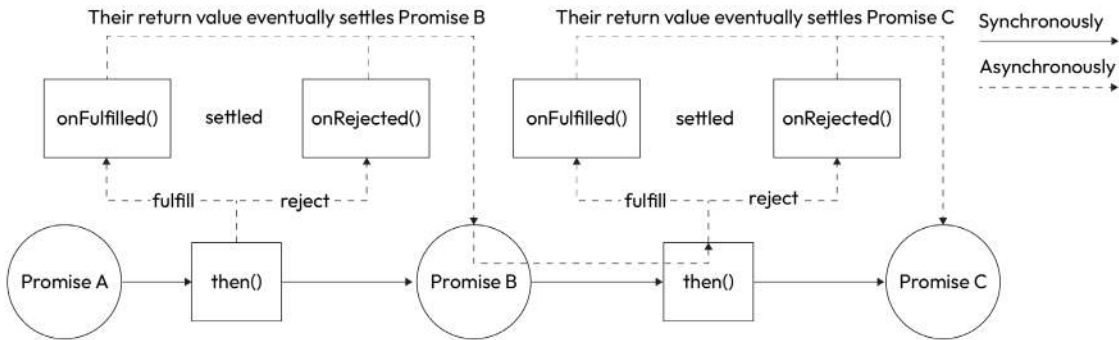


Figure 5.1: Promise chain execution flow

Figure 5.1 shows how our program flows when we use a chain of promises. When we invoke `then()` on **Promise A**, we synchronously receive **Promise B** as a result, and when we invoke `then()` on **Promise B**, we synchronously receive **Promise C** as a result. Eventually, when **Promise A** settles, it will either fulfill or reject, which results in the invocation of either the `onFulfilled()` or the `onRejected()` callback, respectively. The result of the execution of such a callback will then fulfill or reject **Promise B** and such a result is, in turn, propagated to the `onFulfilled()` or the `onRejected()` callback passed to the `then()` invocation on **Promise B**. The execution continues similarly for **Promise C** and any other promise that follows in the chain.

From a user's perspective, this means that once the entire expression is evaluated (e.g., `PromiseA.then(...).then(...)`), the result is the last promise in the chain—in this case, **Promise C**. This value is returned *synchronously* at the time the expression is evaluated, even though the underlying asynchronous operations have not completed yet. Let's say we assign this value to a variable called `promiseC`; we can now attach some completion handlers using `promiseC.then(onFulfilled, onRejected)` to be notified *asynchronously* when `promiseC` eventually settles, allowing us to handle the final outcome of the entire promise chain.

An important property of promises is that the `onFulfilled()` and `onRejected()` callbacks are guaranteed to be invoked asynchronously and at most once, even if the promise is already settled when `.then()` is called. Here's an example:

```
Promise.resolve(someValue).then(onFulfilled, onRejected)
```

In this case, `Promise.resolve(someValue)` returns a promise that is already fulfilled. However, `onFulfilled` will not be called immediately. Instead, it will be scheduled to run asynchronously, after the current call stack is cleared.

This scheduling is handled using the **microtask queue**, a mechanism that ensures that promise callbacks are run as soon as possible, but only after the current synchronous code has finished executing. This behavior helps prevent unexpected interleaving of synchronous and asynchronous logic, a common issue known as *Zalgo* (see *Chapter 3, Callbacks and Events*). Thanks to this, promises provide a consistent, predictable model for asynchronous control flow, making it easier to reason about the timing of code execution.

Now comes the best part. If an exception is thrown (using the `throw` statement) in the `onFulfilled()` or `onRejected()` handler, the `Promise` returned by the `then()` method will automatically reject, with the exception that was thrown. This is a tremendous advantage over CPS, as it means that with promises, exceptions will propagate automatically across the chain, and the `throw` statement becomes finally usable in asynchronous code.

Promises/A+ and thenables

Historically, there have been many different implementations of promises, and most of them were not compatible with each other, meaning that it was not possible to create chains between `Promise` objects coming from libraries that were using different `Promise` implementations.

The JavaScript community worked very hard to address this limitation, and those efforts led to the creation of the **Promises/A+** specification. This specification details the behavior of the `then()` method, providing an interoperable base, which makes `Promise` objects from different libraries able to work with each other out of the box. Today, the majority of `Promise` implementations use this standard, including the native `Promise` object of JavaScript and Node.js.



For a detailed overview of the **Promises/A+** specification, you can refer to the official website at nodejsdp.link/promises-aplus.

As a result of the adoption of the Promises/A+ standard, many Promise implementations, including the native JavaScript Promise API, will consider any object with a `then()` method a Promise-like object, also called **thenable**. This behavior allows different Promise implementations to interact with each other seamlessly.



The technique of recognizing (or typing) objects based on their external behavior, rather than their actual type, is called **duck typing** (nodejsdp.link/duck) and is widely used in JavaScript.

The promise API

Let's now take a quick look at the API of the native JavaScript Promise. This is just a theoretical overview to give you an idea of what we can do with promises. Try to make an effort to understand the semantics, but don't worry if things are not so clear at this point yet; we will have the chance to use most of these APIs throughout the book, and practice is often the best teacher.

The Promise constructor (`new Promise((resolve, reject) => {})`) creates a new Promise instance that fulfils or rejects based on the behavior of the function provided as an argument. The function provided to the constructor will receive two arguments:

- `resolve(obj)`: This is a function that, when invoked, will fulfil the Promise with the provided fulfillment value, which will be `obj` if `obj` is a value. It will be the fulfillment value of `obj` if `obj` is a Promise or a thenable.
- `reject(err)`: This rejects the Promise with the reason `err`. It is a convention for `err` to be an instance of `Error`.

Now, let's look at the most important static methods of the Promise object:

- `Promise.resolve(obj)`: This method creates a new Promise from another Promise, a thenable, or a value. If a Promise is passed, then that Promise is returned as it is. If a thenable is provided, then it's *converted* to the Promise implementation in use. If a value is provided, then the Promise will be fulfilled with that value.
- `Promise.reject(err)`: This method creates a Promise that rejects with `err` as the reason.
- `Promise.all(iterable)`: This method creates a Promise that fulfils with an array of fulfillment values when every item in the input `iterable` (such as an `Array`) object fulfils. If any Promise in the `iterable` object rejects, then the Promise returned by `Promise.all()` will reject with the first rejection reason. Note that if any promise in the `iterable` rejects, the returned promise from `Promise.all()` will reject immediately, but any other pending promises in the `iterable` will continue to execute and settle independently; they are *not* automatically cancelled or aborted.

- `Promise.allSettled(iterable)`: This method waits for all the input promises to fulfil or reject and then returns an array of objects containing the fulfillment value or the rejection reason for each input Promise. Each output object has a `status` property, which can be equal to `'fulfilled'` or `'rejected'`, and a `value` property containing the fulfillment value, or a `reason` property containing the rejection reason. The difference with `Promise.all()` is that `Promise.allSettled()` will always wait for each Promise to either fulfil or reject, instead of immediately rejecting when one of the promises rejects.
- `Promise.race(iterable)`: This method returns a Promise that is equivalent to the first Promise in `iterable` that settles (including if the first Promise is rejected, in that case, the whole operation results in a rejection). While `Promise.race()` is used less frequently, it does have practical use cases. For example, if you have several servers available, you can ping them all at once and use `Promise.race()` to pick the first one that responds. The winner is the server with the lowest observed latency right now, so you can prefer it for subsequent requests.
- `Promise.withResolvers()`: This returns an object containing a new Promise object and two functions to resolve or reject it, corresponding to the two parameters passed to the executor of the `Promise()` constructor. It is roughly equivalent to the following function:

```
function withResolvers() {  
  let resolve  
  let reject  
  const promise = new Promise((res, rej) => {  
    resolve = res  
    reject = rej  
  })  
  return { promise, resolve, reject }  
}
```

When you call this function, it gives you access to the promise `resolve()` and `reject()` functions, allowing you to control the promise's outcome externally. This is especially useful when integrating with callback-based or event-driven code, where you need to manually resolve or reject a promise. We'll see a practical use case for this in *Chapter 10, Testing: Patterns and Best Practices*.

The following are the main methods available on a Promise instance:

- `promise.then(onFulfilled, onRejected)`: This is the essential method of a Promise to handle continuation. Its behavior is compatible with the Promises/A+ standard that we mentioned before.
- `promise.catch(onRejected)`: This method is just syntactic sugar (nodejsdp.link/syntactic-sugar) for `promise.then(undefined, onRejected)`.

- `promise.finally(onFinally)`: This method allows us to set up an `onFinally` callback, which is invoked when the Promise is settled (either fulfilled or rejected). Unlike `onFulfilled` and `onRejected`, the `onFinally` callback will not receive any argument as input, and any value returned from it will be ignored. The Promise returned by `finally` will settle with the same fulfillment value or rejection reason of the current Promise instance. There is only one exception to all this, which is the case in which we throw inside the `onFinally` callback or return a rejected Promise. In this case, the returned Promise will reject with the error that is thrown or the rejection reason of the rejected Promise returned.

Let's now see an example of how we can create a Promise from scratch using its constructor.

Creating a promise

Creating a Promise from scratch is a low-level operation, and it's usually required when we need to convert an API that uses another asynchronous style (such as a callback-based style). Most of the time, you are going to work with promises produced by other libraries, and most of the promises we create will come from the `then()` method. Nonetheless, in some advanced scenarios, we need to manually create a Promise using its constructor.

To demonstrate how to use the Promise constructor, let's create a function that returns a Promise that fulfills with the current Unix timestamp after a specified number of milliseconds. A Unix timestamp is defined as the number of seconds that have elapsed since January 1, 1970, at 00:00:00 UTC (the Unix epoch). In JavaScript, we can get the current Unix timestamp with `Date.now()`, but keep in mind that it will return the value in milliseconds rather than in seconds.

Let's look at a possible implementation of this function:

```
function delay(milliseconds) {  
  return new Promise((resolve, _reject) => {  
    setTimeout(() => {  
      resolve(Date.now())  
    }, milliseconds)  
  })  
}
```

As you might have guessed, we used `setTimeout()` to invoke the `resolve()` function of the Promise constructor.

Note that the **executor function** (the function passed to the Promise constructor) is executed synchronously as soon as the promise is created. We'll come back to this later in the chapter when we discuss how to create *lazy* promises.

We can notice how the entire body of the function is wrapped by the Promise constructor; this is a frequent code pattern you will see when creating a Promise from scratch.

We can now use the `delay()` function. Let's see an example:

```
console.log(`Delaying... (${Date.now()})`)

delay(1000).then(newDate => {
  console.log(`Done (${newDate})`)
})
```

The `console.log()` within the `then()` handler will be executed approximately 1 second after the invocation of `delay()`. You can see that the two Unix timestamps will differ by about (but not exactly!) 1,000 milliseconds.



This simple example illustrates an often-surprising property of Node.js timers: scheduling tasks with `setTimeout()` is considered best effort, not guaranteed accurate timing. In fact, the callback may not execute exactly after the specified timeout has expired. This discrepancy arises because there can be delays between when a timer completes and when its completion callback is picked up and executed by the event loop.

Also note that, here, we created our custom promise-based function to be able to dynamically control the creation of the value the promise resolves with. If you just want to wait for a certain amount of time, you can simply use the built-in `setTimeout()` Node.js function from the `'node:timers/promises'` module, as follows:

```
import { setTimeout } from 'node:timers/promises'
await setTimeout(100) // or setTimeout(100).then(/* ... */)
```



The Promises/A+ specification states that the `onFulfilled` and `onRejected` callbacks of the `then()` method have to be invoked only once and exclusively (only one or the other is invoked). A compliant promise implementation makes sure that even if we call `resolve` or `reject` multiple times, the Promise is either fulfilled or rejected only once.

Promisification

When some characteristics of a callback-based function are known in advance, it's possible to create a function that transforms such a callback-based function into an equivalent function returning a Promise. This transformation is called **promisification**.

For example, let's consider the conventions used in Node.js-style callback-based functions:

- The callback is the last argument of the function
- The error (if any) is always the first argument passed to the callback
- Any return value is passed after the error to the callback

Based on these rules, we can easily create a generic function that *promisifies* a Node.js-style callback-based function. Let's see what this function looks like:

```
function promisify(callbackBasedFn) {  
  return function promisifiedFn(...args) {  
    return new Promise((resolve, reject) => { // 1  
      const newArgs = [ // 2  
        ...args, // 3  
        (err, result) => { // 4  
          if (err) {  
            return reject(err)  
          }  
          resolve(result)  
        },  
      ],  
      callbackBasedFn(...newArgs) // 5  
    })  
  }  
}
```

The preceding function returns another function called `promisifiedFn()`, which represents the promisified version of `callbackBasedFn` given as input. This is how it works:

1. The `promisifiedFn()` function creates a new `Promise` using the `Promise` constructor and immediately returns it to the caller.
2. In the function passed to the `Promise` constructor, we build a new set of arguments to pass to the original callback-based function (`callbackBasedFn`).
3. This list of arguments includes all the arguments received by the caller of `promisifiedFn(...args)`.
4. The list of arguments also includes: a callback function that is used to capture the result of `callbackBasedFn` and resolve the promise accordingly. If this callback function has received an error (`err`), then we call `reject()`; otherwise, we call `resolve()` with the received result.
5. Finally, we simply invoke `callbackBasedFn` with the list of arguments we have built to trigger the execution of the original business logic that we have now wrapped in a promise-based interface.

Now, let's promisify a Node.js function using our newly created `promisify()` function. We can use the `randomBytes()` function of the core `crypto` module, which produces a buffer containing the specified number of random bytes. The `randomBytes()` function accepts a callback as the last argument, and it follows the callback conventions we already know very well (the callback will be called with two arguments: an error and a value).

Let's see what this looks like:

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

const randomBytesP = promisify(randomBytes)
randomBytesP(32)
  .then(buffer => {
    console.log(`Random bytes: ${buffer.toString()}`)
  })
```

The previous code should print some random gibberish sequence to the console; that's because most of the randomly generated bytes don't correspond to a printable character.



The promisification function we created here is just for educational purposes, and it's missing a few features, such as the ability to deal with callbacks returning more than one result. In real life, we would use the `promisify()` function from the `util` core module to promisify our Node.js-style callback-based functions. You can take a look at its documentation at nodejsdp.link/promisify.

Sequential execution and iteration

At this point, we now know enough to convert the web spider application that we created in the previous chapter to use promises. Let's start directly from version 2, the one downloading the links of a web page in sequence.



We can access an already promisified version of the core `fs` API through the `promises` object of the `fs` module—for example: `import { promises } from 'node:fs'`. Alternatively, you can also import from `'node:fs/promises'`. This pattern is present in many other Node.js core modules that were originally written with only callbacks in mind and later got extended with an alternative promisified version of their callback-based functions. Some other notable examples are `'node:dns/promises'`, `'node:timers/promises'`, `'node:inspector/promises'`, `'node:readline/promises'`, and `'node:stream/promises'`.

Be aware that in this version of the code, we have manually promisified all the helper functions that you can find in the `utils.js` file in the code repository; therefore, we are not using callbacks anymore. This means, for example, that we can now call `get()` without a callback function and get back a promise that tracks the progress of the asynchronous operation.

Now, let's start by converting the `download()` function to use promises:

```
function download(url, filename) {  
  console.log(`Downloading ${url} into ${filename}`)  
  return get(url)  
    .then(content => saveFile(filename, content))  
}
```

For the sake of comparison, let's review the previous callback-based version of this function as implemented in the previous chapter:

```
function download(url, filename, cb) {  
  console.log(`Downloading ${url} into ${filename}`)  
  get(url, (err, content) => {  
    if (err) {  
      return cb(err)  
    }  
    saveFile(filename, content, err => {  
      if (err) {  
        return cb(err)  
      }  
      cb(null, content)  
    })  
  })  
}
```

The first thing that we should appreciate is that we reduced the code to almost a third! The new code is not just shorter but it should also feel easier to follow. It almost reads in plain English:

get (the content of) url, then (take the) content (and) saveFile (with) filename (and) content.

Also, note how we are returning the promise chain to the caller. This makes it so that the caller receives another promise that will eventually resolve to the result of the `saveFile()` call (which is, in turn, a promise that resolves to the content that was written to the file). In other words, this function doesn't just download the content of a given URL to a file, but it also exposes that very content to the caller. This is to make sure that the semantics of our new `download()` function haven't changed from the previous version.

In the `download()` function that we've just seen, we have executed a known set of asynchronous operations in sequence: get the content, save it to a file. However, in the `spiderLinks()` function, we will have to deal with a sequential iteration over a dynamic set of asynchronous tasks. Let's see how we can achieve that:

```
function spiderLinks(currentUrl, body, maxDepth) {  
  let promise = Promise.resolve() // 1  
  if (maxDepth === 0) {
```

```
    return promise
  }

  const links = getPageLinks(currentUrl, body)
  for (const link of links) {
    promise = promise.then(() => spider(link, maxDepth - 1)) // 2
  }

  return promise
}
```

We will spare you the comparison this time, but if you were to go back to the previous chapter and compare this implementation with the previous one, you would see that this one is also significantly shorter.

The interesting new bit of knowledge brought by this version is that, in order to iterate over all the links of a web page asynchronously, we had to dynamically build a chain of promises. This is achieved as follows:

1. First, we defined an “empty” Promise, which resolves to undefined. This Promise is used just as the starting point for our chain.
2. Then, in a loop, we update the promise variable with a new Promise obtained by invoking `then()` on the previous promise in the chain. This is effectively an implementation of our asynchronous iteration pattern using promises.

At the end of the for loop, the promise variable will contain the promise of the last `then()` invocation, so it will resolve only when all the promises in the chain have been resolved.



Pattern (sequential iteration with promises)

Dynamically build a chain of promises using a loop.

An alternative to the sequential iteration pattern with promises makes use of the `reduce()` function, for an even more compact implementation:

```
const promise = tasks.reduce((prev, task) => {
  return prev.then(() => {
    return task()
  })
}, Promise.resolve())
```

Now, we can finally convert the `spider()` function:

```
export function spider(url, maxDepth) {
  const filename = urlToFilename(url)

  return exists(filename).then(alreadyExists => {
    if (alreadyExists) {
      if (!filename.endsWith('.html')) {
        // ignoring non-HTML resources
        return
      }

      return readFile(filename, 'utf8').then(fileContent =>
        spiderLinks(url, fileContent, maxDepth)
      )
    }

    // if file does not exist, download it
    return download(url, filename).then(fileContent => {
      // if the file is an HTML file, spider it
      if (filename.endsWith('.html')) {
        return spiderLinks(url, fileContent.toString('utf8'), maxDepth)
      }

      // otherwise, stop here
      return
    })
  })
}
```

In this new version of the `spider()` function, we are leveraging a new promise-based version of the `exists()` helper function. This function now returns a promise that resolves to the Boolean value: `true` if the given file path exists, `false` otherwise. We can capture this value with `then()` and then handle the two different code branches: if the file exists (and it's an HTML file), we can read its content and continue spidering from there; otherwise, we have to download it before we can continue with the spidering. The basic logic of the function hasn't changed, but we have now managed to remove all the callbacks from the code. One important detail to underline is that the conditional nature of this code forces us to retain some level of nesting. In fact, we cannot simply inline everything in a single, nice chain of events, because we effectively have to deal with two distinct code paths. We will see in the second part of this chapter how `async/await` makes handling asynchronous conditional logic as easy as handling it in synchronous code.

Now that we have converted our `spider()` function as well, we can finally modify the `spider-cli.js` module:

```
spider(url, maxDepth)
  .then(() => console.log('Downloaded complete'))
  .catch(err => {
    console.error(err)
    process.exit(1)
  })
```

This one should be quite straightforward, but it's important to note that this is the first place in this new version where we are explicitly handling errors with a `catch()` handler. This handler will intercept any error originating from the entire `spider()` process, which highlights how promises simplify the propagation of errors and reduce the amount of code needed for error handling. This is clearly an enormous advantage, as it greatly reduces the boilerplate in our code and the chances of missing any asynchronous errors.

This completes the implementation of version 2 of our web spider application with promises.

Concurrent execution

Another execution flow that becomes simple with promises is the concurrent execution flow. In fact, all that we need to do is use the built-in `Promise.all()` method. This helper function creates another Promise that fulfils only when all the promises received as input are fulfilled. If there is no causal relationship between those promises (for example, they are not part of the same chain of promises), then they will be executed concurrently.

To demonstrate this, let's consider version 3 of our web spider application, which downloads all the links of a page concurrently. Let's just update the `spiderLinks()` function again to implement a concurrent execution flow using promises:

```
function spiderLinks(currentUrl, body, maxDepth) {
  if (maxDepth === 0) {
    return Promise.resolve()
  }

  const links = getPageLinks(currentUrl, body)
  const promises = links.map(link => spider(link, maxDepth - 1))

  return Promise.all(promises)
}
```


The pattern here consists of starting the tasks all at once in the `links.map()` loop. At the same time, each Promise returned by invoking `spider()` is collected in the final `promises` array. The critical difference in this loop—as compared to the sequential iteration loop—is that we are not waiting for the previous `spider()` task in the list to complete before starting a new one. All the `spider()` tasks are started in the loop at once, in the same event loop cycle.

Once we have all the promises, we pass them to the `Promise.all()` method, which returns a new Promise that will be fulfilled when all the promises in the array are fulfilled. In other words, it fulfills when all the download tasks have completed. In addition to that, the Promise returned by `Promise.all()` will reject immediately if any of the promises in the input array reject. This is exactly what we wanted for this version of our web spider.

Limited concurrent execution

So far, promises have not disappointed our expectations. We were able to greatly improve our code for both serial and concurrent execution. Now, with limited concurrent execution, things should not be that different, considering that this flow is just a combination of serial and concurrent execution.

In this section, we will go straight to implementing a solution that allows us to *globally* limit the concurrency of our web spider tasks.



If you are just interested in a simple solution to locally limit the concurrent execution of a set of tasks, you can still apply the same principles that we will see in this section to implement a special asynchronous version of `Array.map()`. We leave this to you as an exercise; you can find more details and hints at the end of this chapter (*Exercise 5.4 An asynchronous map()*).

For a ready-to-use, production-ready implementation of a `map()` function supporting promises and limited concurrency, you can rely on the `p-map` package. Find out more at nodejsdp.link/p-map.

Implementing the TaskQueue class with promises

To globally limit the concurrency of our spider download tasks, we'll reuse the `TaskQueue` class we originally implemented in version 4 of our crawler, described in the previous chapter. Back then, the queue was designed to handle tasks using callbacks. In this version, however, we want it to handle tasks that return promises. To support this, we need to update the `next()` method, which is responsible for dispatching tasks until we haven't reached the concurrency limit.

For a quick reference, here's the previous version of the `next()` function:

```
next() {
  if (this.running === 0 && this.queue.length === 0) {
    return this.emit('empty')
  }

  while (this.running < this.concurrency && this.queue.length > 0) {
```

```
const task = this.queue.shift()
task(err => {
  if (err) {
    this.emit('error', err)
  }
  this.running--
  process.nextTick(this.next.bind(this))
})
this.running++
}
```

And here's the updated version:

```
next() {
  if (this.running === 0 && this.queue.length === 0) {
    return this.emit('empty')
  }

  while (this.running < this.concurrency && this.queue.length > 0) {
    const task = this.queue.shift()
    task() // 1
      .catch(err => { // 4
        this.emit('error', err)
      })
      .finally(() => { // 2
        this.running--
        this.next() // 3
      })

    this.running++
  }
}
```

Let's break down what's happening here by comparing it to the previous callback-based implementation:

1. In the original version, each task accepted a callback and called it when done. Now, each task is a function that returns a promise. This enables us to await asynchronous operations within each task without needing to manually manage callbacks. We use `.then()` instead.
2. Previously, we manually decremented the running count inside the callback. Now, we use the `.finally()` method of the promise to ensure that running is decremented once the task finishes, regardless of whether it resolved or rejected.

3. After a task finishes, we schedule the next iteration of `next()`, just like before.
4. In the original version, we needed to check for errors by inspecting the `err` argument passed to the callback function, which required us to use a dedicated `if` statement. Now, we can use `.catch()` to listen for promise rejections and emit an error event if something goes wrong. This makes error handling cleaner and more consistent.

These are the only changes we need to adapt our `TaskQueue` class to use promises rather than rely on callbacks. Next, we'll use this new version of the `TaskQueue` class to implement version 4 of our web spider.

Updating the web spider

Now it's time to adapt our web spider to implement a limited concurrent execution flow using the `TaskQueue` class we have just created. We will also take this opportunity to apply some optimizations. For example, we will include a set that allows us to track which URLs are being processed or have been processed already, so that we can skip them in case we find them again during the crawling process.

With this idea in mind, let's start by copying the `spider.js` code from version 3 of our promise-based spider and let's update the `spiderLinks()` function first:

```
const spidering = new Set() // 1
function spiderLinks(currentUrl, body, maxDepth, queue) { // 2
  if (maxDepth === 0) {
    return
  }
  const links = getPageLinks(currentUrl, body)
  for (const link of links) { // 3
    if (!spidering.has(link)) {
      queue.pushTask(() => spider(link, maxDepth - 1, queue))
      spidering.add(link)
    }
  }
}
```

Here are the changes:

1. First of all, we created the `spidering` set to keep track of which URLs are in progress or have been completed already.
2. We added `queue` (an instance of our `TaskQueue` class) to the list of arguments in the `spiderLinks()` function.

3. Instead of using `Array.map()` and `Promise.all()` like we did in the previous version, we now use a regular for loop to go through all the links found in the current URL. For each link that isn't already in the spidering set, we create a new task, add it to the queue, and update the spidering set with this link. The task we create is an arrow function that calls the `spider()` function with the correct parameters (note how `maxDepth` is decreased by one). This approach defers the execution of `spider()` until the task is picked up from the queue. This method allows us to control concurrency, as we can keep adding tasks without starting them immediately; they run only when they're picked from the queue.



In this version, we've brought back the concept of the spidering set, which we first introduced in *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*. This is an optimization technique that helps reduce the number of tasks and prevents unnecessary duplicate work. Observant readers may notice that this time we implemented it slightly differently. In *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*, we checked for duplicates within the `spider()` function, but now we've moved that check to the `spiderLinks()` function. This additional optimization allows us to avoid creating tasks that would otherwise be pointless when pulled from the queue. This keeps the system more efficient by limiting the queue's growth, something that can be important for websites with a large number of links.

We can now update the `spider()` function:

```
export function spider(url, maxDepth, queue) { // 1
  const filename = urlToFilename(url)
  return exists(filename).then(alreadyExists => {
    if (alreadyExists) {
      if (!filename.endsWith('.html')) {
        return
      }
      return readFile(filename, 'utf8').then(fileContent =>
        spiderLinks(url, fileContent, maxDepth, queue) // 2
      )
    }
    return download(url, filename).then(fileContent => {
      if (filename.endsWith('.html')) {
        return spiderLinks(url, fileContent.toString('utf8'), maxDepth,
          queue) // 3
      }
    })
  })
}
```

Nothing much has changed here compared to the previous version of our `spider()` function. We just needed to introduce our queue instance in the function signature (1) and then make sure we propagate it when we call `spiderLinks()` (2,3).

The last thing we need to do is to update our CLI script (`spider-cli.js`):

```
const url = process.argv[2]
const maxDepth = Number.parseInt(process.argv[3], 10) || 1
const concurrency = Number.parseInt(process.argv[4], 10) || 2 // 1

const queue = new TaskQueue(concurrency) // 2
queue.pushTask(() => spider(url, maxDepth, queue)) // 3
queue.on('error', console.error) // 4
queue.on('empty', () => {
  console.log('Download complete')
})
```

This code isn't very different from what we did in *Chapter 4, Asynchronous Control Flow Patterns with Callbacks* when we implemented the limited concurrent execution pattern using callbacks. Here's a quick summary of what's changed from our promise-based web spider v3:

1. We are now accepting an additional CLI argument to allow the user to specify the desired maximum number of concurrent tasks.
2. We create an instance of `TaskQueue` with the given concurrency.
3. We add the first task to the queue, which will start the *spidering* of the given URL. Note that we have to make sure to pass the queue instance we just created to the `spider` function so that it will use it internally to keep scheduling new tasks into that queue.
4. Finally, we attach some event listeners to the queue to know if there's an error (and log it to the standard error) and to know when all the tasks have been processed.

That's it! Now you can finally try this new version. Try to use it against a significantly big website and see how it behaves. You can play with different values for the `concurrency` and `maxDepth` parameters to see how fast you can download pages from the website. Of course, always remember the legal implications of scraping, so it's best if you can perform these tests against a website you own, or—even better—a website you can run on localhost.



In production code, you can use the `p-limit` package (available at nodejsdp.link/p-limit) to limit the concurrency of a set of tasks. The package essentially implements the pattern we have just shown, but wrapped in a slightly different API. If you are curious, you can check the implementation. You probably won't be too surprised to find out that this library also internally uses a queue to manage the scheduling of tasks.

Lazy promises

So far, we have learned that we can create a Promise instance by using its canonical constructor:

```
new Promise((resolve, reject) => {  
  // promise execution logic ...  
}))
```

The function we pass to the Promise constructor is called the **executor function**, and it's where we define the lifecycle of a custom Promise. It's important to know that the executor function is invoked immediately (*synchronously*) as soon as the Promise object is created.

This detail matters because, sometimes, we might need to run resource-intensive operations (such as reading a large file or accessing a remote resource), or we might need to create a large number of Promise instances that we only need to use later in the code. In other cases, we may want to create a Promise but have conditional logic that could end up not needing it (for instance, loading a default configuration file if the user hasn't provided an explicit configuration at runtime).

In such cases, it can be beneficial to postpone creating the Promise instance (and thus the execution of its executor function) until we actually need to interact with it, such as when calling `then()`, `catch()`, or `finally()`. By deferring the executor function's execution to this point, we can avoid unnecessary resource use, helping to keep our code more efficient.

The Promise class doesn't support this kind of deferred instantiation natively, but we can implement this ourselves in a few ways. In fact, we've already implemented a form of lazy or *deferred* Promise in version 4 of our promise-based web scraper. There, we needed a way to schedule a task (crawling a URL) without starting it immediately. To do this, we simply wrapped the Promise instantiation in an arrow function, a pattern which we can generalize as follows:

```
const lazyPromise = () => new Promise((resolve, reject) => {  
  // ...  
})
```

This code doesn't create the Promise instance immediately. Instead, it creates a function that, when called, will create the Promise instance. We can now use this function as follows:

```
lazyPromise().then((value) => {  
  // ...  
})
```

Notice that we need to explicitly invoke `lazyPromise()` before we can use `then()` on it. While this approach works, it doesn't create true lazy Promise objects; instead, it just gives us a function that needs to be called to produce a Promise.



Technically speaking, this approach should be considered an implementation of the factory function pattern (covered in more detail in *Chapter 7, Creational Design Patterns*) rather than a true lazy promise object.

When working with third-party libraries that expect plain Promise objects, this approach to deferring the Promise initialization won't work. The third-party code will eventually try to call `then()` directly on our function without invoking it first, which will lead to the following error:

```
Uncaught TypeError: lazyPromise.then is not a function
```

An alternative approach might be to implement an extension of the Promise class that behaves lazily. Let's call it LazyPromise:

```
export class LazyPromise extends Promise { // 1
  #resolve // 2
  #reject
  #executor
  #promise

  constructor(executor) { // 3
    let _resolve
    let _reject
    super((resolve, reject) => {
      _resolve = resolve
      _reject = reject
    })
    this.#resolve = _resolve
    this.#reject = _reject
    this.#executor = executor
    this.#promise = null
  }

  #ensureInit() { // 4
    if (!this.#promise) {
      this.#promise = new Promise(this.#executor)
      this.#promise.then(
        v => this.#resolve(v),
        e => this.#reject(e)
      )
    }
  }

  then(onFulfilled, onRejected) { // 5
    this.#ensureInit()
    return this.#promise.then(onFulfilled, onRejected)
  }
}
```

```
catch(onRejected) {
  this.#ensureInit()
  return this.#promise.catch(onRejected)
}

finally(onFinally) {
  this.#ensureInit()
  return this.#promise.finally(onFinally)
}
}
```

Let's see how this works:

1. The LazyPromise class extends the built-in Promise class, inheriting all its functionalities. This also allows instances of this class to be considered instances of the original Promise class, which means that writing something such as `lazyPromise instanceof Promise` (where `lazyPromise` is an instance of `LazyPromise`) will evaluate to `true`.
2. We are defining some private properties of the class. These will be used to track the lifecycle of the lazy promise, but they won't be accessible from outside the class.



The `#` notation is a relatively new syntax introduced in ECMAScript 2022. If you want to find out more about it, check out nodejsdp.link/private-properties.

3. We redefine the original Promise constructor. This is where most of the magic happens. We need to respect the same interface as in the original Promise constructor, which means that we receive only one argument: the executor function. But in this implementation, we don't immediately invoke the executor function we receive, but we just store it as a private property, together with its `resolve` and `reject` functions. We also create a `promise` property that is initially set to `null`. This is the value that we will populate only when we eventually execute the executor function to create the real Promise instance.
4. We define a private function called `ensureInit()`. This function is responsible for initializing the internal promise instance when needed and wiring it to the original `resolve()` and `reject()` functions of executor. Note that this function will perform the initialization only if the internal promise isn't initialized already.
5. Finally, we reimplement the `then()`, `catch()`, and `finally()` methods. These are the methods that the user calls when they are interested in the result of the promise. Therefore, this is where we need to initialize the internal promise by calling `ensureInit()` and wire the received callback to the internal promise instance.

Let's see a simple example that illustrates how we can use this new class:

```
const lazyPromise = new LazyPromise(resolve => { // 1
  console.log('Executor Started!')
  // simulate some async work to be done
  setTimeout(() => {
    resolve('Completed!')
  }, 1000)
})

console.log('Lazy Promise instance created!')
console.log(lazyPromise)
lazyPromise.then(value => { // 2
  console.log(value)
  console.log(lazyPromise)
})
```

This code is quite simple. We are just instantiating a `LazyPromise` instance and providing an executor function that will resolve the promise after 1 second (1). Later, we use `lazyPromise` by calling `then()` on it (2). This action triggers the execution of the executor function we defined previously. To demonstrate that this works as expected, let's observe the logs produced by this simple snippet:

```
Lazy Promise instance created!
LazyPromise [Promise] { <pending> }
Executor Started!
< 1 second pause... >
Completed!
LazyPromise [Promise] { 'Completed!' }
```

As we can see from the logs, `Lazy Promise instance created!` is logged before `Executor Started!`. This proves that the executor function is not being invoked straight away, but only later on when we call `then()`.

It's also interesting to note how, when we call `console.log(lazyPromise)`, the promise state is correctly tracking the internal state of our lazy promise. Once created, the promise is reported as `<pending>`. Later on, when the promise settles, its state is correctly reported as resolved to the `'Completed!'` value.

Now that you know how to write true lazy promises, you could take a little challenge and try to rewrite version 4 of our promise-based web crawler by using the `LazyPromise` class as a way to define tasks.



One popular library that implements the idea of lazy promises is `p-lazy`. Check it out at nodejsdp.link/p-lazy and feel free to compare its implementation with the one we provided here. You'll find many similarities, but you will also notice that `p-lazy` offers a few additional utilities, so it's probably a better option for production use.

This concludes our exploration of JavaScript promises. Next, we are going to learn about the `async/await` pair, which will completely revolutionize the way we deal with asynchronous code.

Async/await

As we have just seen, promises are a quantum leap ahead of callbacks. They allow us to write clean and readable asynchronous code and provide a set of safeguards that can only be achieved with boilerplate code when working with callback-based asynchronous code. However, promises are still suboptimal when it comes to writing sequential asynchronous code. The `Promise` chain is indeed much better than having callback hell, but still, we have to invoke a `then()` and create a new function for each task in the chain. This is still too much for a control flow that is definitely the most commonly used in everyday programming. JavaScript needed a proper way to deal with the ubiquitous asynchronous sequential execution flow, and the answer arrived with the introduction in the ECMAScript standard of **async functions** and the **await expression** (`async/await` for short).

The `async/await` pair allows us to write functions that appear to block at each asynchronous operation, waiting for the results before continuing with the following statement. As we will see, any asynchronous code using `async/await` has readability comparable to traditional synchronous code.

Today, `async/await` is the recommended construct for dealing with asynchronous code in both Node.js and JavaScript. However, `async/await` does not replace all that we have learned so far about asynchronous control flow patterns; on the contrary, as we will see, `async/await` piggybacks heavily onto promises.

Async functions and the await expression

An `async` function is a special type of function in which it's possible to use the `await` expression to “pause” the execution on a given `Promise` until it resolves. Let's consider a simple example and use the `delay()` function we implemented in the *Creating a promise* subsection. The `Promise` returned by `delay()` resolves with the current date as the value after the given number of milliseconds. Let's use this function with the `async/await` pair:

```
async function playingWithDelays() {
  console.log('Delaying...', Date.now())

  const timeAfterOneSecond = await delay(1000)
  console.log(timeAfterOneSecond)

  const timeAfterThreeSeconds = await delay(3000)
  console.log(timeAfterThreeSeconds)

  return 'done'
}
```

As we can see from the previous function, `async/await` seems to work like magic. The code doesn't even look like it contains any asynchronous operation. However, don't be mistaken; this function does not run synchronously (they are called *async functions* for a reason!). At each `await` expression, the execution of the function is put on hold, its state is saved, and the control is returned to the event loop. Once the Promise that has been *awaited* resolves, the control is given back to the async function, returning the fulfillment value of the Promise.



The `await` expression can be used with a Promise, a thenable object (an object with a `then()` method), or any other value. If the value is not a thenable, JavaScript automatically wraps it in a resolved Promise using `Promise.resolve()`. For example, `await 5` is equivalent to `await Promise.resolve(5)`.

Let's now see how we can invoke our new async function:

```
playingWithDelays()
  .then(result => {
    console.log(`After 4 seconds: ${result}`)
  })
```

From the preceding code, it's clear that async functions can be invoked just like any other function. However, the most observant of you may have already spotted another important property of async functions: they always return a Promise. It's like if the return value of an async function was passed to `Promise.resolve()` and then returned to the caller.



Calling an async function happens immediately, just like any other regular function call. However, instead of returning a value directly, it returns a Promise right away. That Promise will later settle, either resolving with the function's return value or rejecting if an error is thrown.

From this first encounter with `async/await`, we can see how dominant promises still are in our discussion. In fact, we can consider `async/await` just as syntactic sugar for a simpler consumption of promises. As we will see, all the asynchronous control flow patterns with `async/await` use promises and their API for most of the heavy-lifting operations.

Top-level await

Starting with Node.js 14, we gain access to a powerful feature called **top-level await** when working with **ECMAScript Modules (ESM)**. This feature allows us to use the `await` keyword directly at the top level of our module, outside of any async function.



Remember that, to enable ESM, we need to do is make sure our file has an `.mjs` extension or we have to set `"type": "module"` in `package.json`. This was discussed in *Chapter 2, The Module System*.

With top-level `await`, we can make our asynchronous code simpler and more readable. No more calling `then()` on promises or complex wrappers; just clean, direct access to asynchronous operations. Let's say we want to call the `playingWithDelays()` asynchronous function that we wrote in the previous section. In the previous section, we saw how we can invoke the function and use `then()` on the returned promise, but we could have also called it like this:

```
(async () => {  
  const result = await playingWithDelays()  
  console.log(`After 4 seconds: ${result}`)  
})();
```

This is what we call an **async IIFE (Immediately Invoked Function Expression)**. Within the first pair of parentheses, we are defining an inline `async` arrow function and invoking it immediately with the final pair of parentheses.

Now, with top-level `await`, we can cut out these extra steps and simplify our code even more, like this:

```
const result = await playingWithDelays()  
console.log(`After 4 seconds: ${result}`)
```

This code is much cleaner, less verbose, and easier to read.

This technique can be very useful in a few practical use cases when you need to do some `async` initialization before your main business logic can run. Here are some examples:

- **Connecting to a database:** If our app needs to connect to a database, we can now set up that connection at the very start, ensuring everything is ready before handling requests.
- **Fetching configurations:** You might need to fetch the configuration from a file or secrets from a remote secret management service. You could use top-level `await` to pull this configuration before the application logic is executed.
- **Initializing other remote dependencies:** When our module depends on third-party services or external APIs, we can initialize them right at the top level, making it clear and easy to see what our app relies on before it runs.

Error handling with `async/await`

`Async/await` doesn't just improve the readability of asynchronous code under standard conditions, but it also helps when handling errors. In fact, one of the biggest gains of `async/await` is the ability to normalize the behavior of the `try...catch` block, to make it work seamlessly with both synchronous *throws* and asynchronous Promise rejections. Let's demonstrate that with an example.

A unified try...catch experience

Let's define a function that returns a Promise that rejects with an error after a given number of milliseconds. This is very similar to the `delay()` function that we already know very well:

```
function delayError(milliseconds) {  
  return new Promise((_resolve, reject) => {  
    setTimeout(() => {  
      reject(new Error(`Error after ${milliseconds}ms`))  
    }, milliseconds)  
  })  
}
```

Next, let's implement an async function that can both throw an error synchronously or await a Promise that will reject. This function demonstrates how both the synchronous throw and the Promise rejection are caught by the same catch block:

```
async function playingWithErrors(throwSyncError) {  
  try {  
    if (throwSyncError) {  
      throw new Error('This is a synchronous error')  
    }  
    await delayError(1000)  
  } catch (err) {  
    console.error(`We have an error: ${err.message}`)  
  } finally {  
    console.log('Done')  
  }  
}
```

Now, invoke the function like this:

```
playingWithErrors(true)
```

This will print the following to the console:

```
We have an error: This is a synchronous error  
Done
```

Invoke the function with `false` as the input, like this:

```
playingWithErrors(false)
```

This will produce the following output:

```
We have an error: Error after 1000ms  
Done
```

If we remember how we had to deal with errors in *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*, we will surely appreciate the giant improvements introduced by both promises and `async/await`. Now, error handling is just as it should be: simple, readable, and most importantly, it works the same for both synchronous and asynchronous errors.

The “return” versus “return await” trap

One common antipattern when dealing with errors with `async/await` is returning a `Promise` that rejects to the caller and expecting the error to be caught by the local `try...catch` block of the `async` function.

For example, consider the following code:

```
async function errorNotCaught() {
  try {
    return delayError(1000)
  } catch (err) {
    console.error('Error caught by the async function: ' +
      err.message)
  }
}

errorNotCaught()
  .catch(err => console.error('Error caught by the caller: ' +
    err.message))
```

The `Promise` returned by `delayError()` is not awaited locally, which means that it's returned as it is to the caller. Consequently, the local `catch` block will never be invoked. In fact, the previous code will output the following:

```
Error caught by the caller: Error after 1000ms
```

If our intention is to locally catch any error generated by the asynchronous operation that produces the value that we want to return, then we have to use the `await` expression on that `Promise` *before* we return the value to the caller. The following code demonstrates this:

```
async function errorCaught() {
  try {
    return await delayError(1000)
  } catch (err) {
    console.error('Error caught by the async function: ' +
      err.message) }
}

errorCaught()
  .catch(err => console.error('Error caught by the caller: ' +
    err.message))
```

All we did was add an `await` after the `return` keyword. This is enough to cause the `async` function to “deal” with the `Promise` locally and therefore also catch any rejection locally. As a confirmation, when we run the previous code, we should see the following output:

```
Error caught by the async function: Error after 1000ms
```



Another scenario where `return await` can be useful is during debugging. When you use `return await` inside an `async` function, it keeps that function's frame on the call stack until the awaited `Promise` settles. Although this introduces a slight performance cost (an extra microtask before the outer `Promise` resolves), it can give you a more complete call stack, making it easier to trace where the `Promise` originated. If you're curious to dive deeper into this trade-off, check out this excellent blog post by the V8 team: nodejsdp.link/fast-async.

Sequential execution and iteration

Our exploration of control flow patterns with `async/await` starts with sequential execution and iteration. We already mentioned a few times that the core strength of `async/await` lies in its ability to make asynchronous serial execution easy to write and straightforward to read. This was already apparent in all the code samples we have written so far; however, it will become even more obvious now that we will start converting our web spider version 2. `Async/await` is so simple to use and understand that there are really no patterns here to study. We will get straight to the code, without any preamble.

So, let's start with our `saveFile()` and `download()` functions of our web spider; this is how it looks with `async/await`:

```
async function saveFile(filename, content) {
  await recursiveMkdir(dirname(filename))
  return writeFile(filename, content)
}

async function download(url, filename) {
  console.log(`Downloading ${url} into ${filename}`)
  const content = await get(url)
  await saveFile(filename, content)
  return content
}
```

Let's appreciate for a moment how simple and compact these two functions have become. Let's just consider that the same functionality was implemented with callbacks in two different functions using more than 20 lines of code. Now we just have 10. Plus, the code is now completely flat, with no nesting at all. This tells us a lot about the enormous positive impact that `async/await` has on our code.

Now, let's see how we can iterate asynchronously over an array using `async/await`. This is exemplified in the `spiderLinks()` function:

```
async function spiderLinks(currentUrl, body, maxDepth) {
  if (maxDepth === 0) {
    return
  }

  const links = getPageLinks(currentUrl, body)
  for (const link of links) {
    await spider(link, maxDepth - 1)
  }
}
```

Even here, there is no pattern to learn. We just have a simple iteration over a list of links, and for each item, we await the Promise returned by `spider()`.

The next code fragment shows the `spider()` function implemented using `async/await`:

```
export async function spider(url, maxDepth) {
  const filename = urlToFilename(url)

  let content
  if (!(await exists(filename))) {
    // if the file does not exist, download it
    content = await download(url, filename)
  }

  // if the file is not an HTML file, stop here
  if (!filename.endsWith('.html')) {
    return
  }

  // if file content is not already loaded, load it from disk
  if (!content) {
    content = await readFile(filename)
  }

  // spider the links in the file
  return spiderLinks(url, content.toString('utf8'), maxDepth)
}
```


If you compare this implementation with the previous one using promises, you can see that here we were able to restructure the logic a little bit. With promises, it's hard to express conditional logic, so we had some repetition in our code (we were calling `spiderLinks()` in two different branches). Here, since `async/await` makes it easy to deal with conditions, we could achieve a better code structure and remove the duplicated call. The comments in the code should clarify exactly what the intended behavior is. This makes the code easier to maintain in the long term.

With the `spider()` function, we have completed the conversion of our web spider application to `async/await`. As you can see, it has been quite a smooth process, but the results are quite impressive.

Antipattern – using `async/await` with `Array.forEach` for serial execution

It's worth mentioning that there is a common antipattern whereby developers will try to use `Array.forEach()` or `Array.map()` to implement a sequential asynchronous iteration with `async/await`, which, won't work as expected.

To see why, let's take a look at the following alternate implementation (which is wrong!) of the asynchronous iteration in the `spiderLinks()` function:

```
links.forEach(async function iteration(link) {  
    await spider(link, nesting - 1)  
})
```

In the previous code, the iteration function is invoked once for each element of the `links` array. Then, in the iteration function, we use the `await` expression on the Promise returned by `spider()`. However, the Promise returned by the iteration function is just ignored by `forEach()`. The result is that all the `spider()` functions are invoked in the same round of the event loop, which means they are started concurrently, and the execution continues immediately after invoking `forEach()`, without waiting for all the `spider()` operations to complete.

Concurrent execution

There are mainly two ways to run a set of tasks concurrently using `async/await`; one purely uses the `await` expression and the other relies on `Promise.all()`. They are both very simple to implement; however, be advised that the method relying on `Promise.all()` is the recommended (and optimal) one to use.

Let's see an example of both. Let's consider the `spiderLinks()` function of our web spider (v3). If we wanted to purely use the `await` expression to implement an unlimited concurrent asynchronous execution flow, we would do it with some code like the following:

```
async function spiderLinks(currentUrl, body, maxDepth) {  
    if (maxDepth === 0) {  
        return  
    }  
}
```

```
const links = getPageLinks(currentUrl, body)
const promises = links.map(link => spider(link, maxDepth - 1))

return Promise.all(promises)
}
```

That's it! Very simple. In the previous code, we first start all the `spider()` tasks concurrently, collecting their promises with a `map()`. Then, we loop, and we await on each one of those promises.

At first, it seems neat and functional; however, it has a small, undesired effect. If a `Promise` in the array rejects, we have to wait for all the preceding promises in the array to resolve before the `Promise` returned by `spiderLinks()` will also reject. This is not optimal in most situations, as we usually want to know if an operation has failed as soon as possible.

Luckily, we already have a built-in function that behaves exactly the way we want, and that's `Promise.all()`. In fact, `Promise.all()` will reject as soon as any of the promises provided in the input array rejects. Therefore, we can simply rely on this method even for all our `async/await` code. And, since `Promise.all()` returns just another `Promise`, we can simply invoke an `await` on it to get the results from multiple asynchronous operations. The following code shows an example:

```
const results = await Promise.all(promises)
```

So, to wrap up, our recommended implementation of the `spiderLinks()` function with concurrent execution and `async/await` will look almost identical to that using promises. The only visible difference is the fact that we are now using an `async` function, which always returns a `Promise`:

```
async function spiderLinks (currentUrl, content, nesting) {
  if (nesting === 0) {
    return
  }

  const links = getPageLinks(currentUrl, content)
  const promises = links.map(link => spider(link, nesting - 1))

  return Promise.all(promises)
}
```



At the beginning of this chapter, we mentioned the `Promise.allSettled()` function. This function is similar to `Promise.all()` but instead of rejecting when a promise in the sequence rejects, it will continue to process all the other promises and resolve only when all the given promises are settled. The result of this operation is an array that describes the state of every promise (fulfilled or rejected) and the respective value or error that every promise settled to. This is a great option when you want to perform concurrent execution while tolerating errors. If you want to make our crawler continue in the presence of occasional failures, you can swap `Promise.all()` with `Promise.allSettled()` in the previous code snippet.

What we just learned about concurrent execution and `async/await` simply reiterates the fact that `async/await` is inseparable from promises. Most of the utilities that work with promises will also seamlessly work with `async/await`, and we should never hesitate to take advantage of them in our `async` functions.

Limited concurrent execution

To implement a limited concurrent execution pattern with `async/await`, we can simply reuse the `TaskQueue` class that we created in the *Limited concurrent execution* subsection within the *Promises* section. We can either use it as it is or convert its internals to `async/await` (as we already did for version 3 of our `async/await`-based web crawler). Converting the `TaskQueue` class to `async/await` is a simple operation, and we'll leave this to you as a quick challenge. Either way, the `TaskQueue` external interface shouldn't change.

You can consult the code repository to see our full implementation, and it shouldn't be too surprising. One thing that might be worth highlighting is our use of the `once()` function from the `core 'node:events' module` in the `spider-cli.js` file.

In version 4 of our promise-based version of the web spider, we were doing something like this:

```
// ...
const queue = new TaskQueue(concurrency)
queue.pushTask(() => spider(url, maxDepth, queue))
queue.on('error', console.error)
queue.on('empty', () => {
  console.log('Download complete')
})
```

Note the use of `queue.on('empty', ...)` at the end. We expect this event to happen only once, when all the tasks have been processed.

In this new async/await-based version, we have done something slightly different that looks like this:

```
import { once } from 'node:events'
// ...
const queue = new TaskQueue(concurrency)
queue.pushTask(() => spider(url, maxDepth, queue))
queue.on('taskError', console.error)
await once(queue, 'empty')
console.log('Download complete')
```

We replaced the `queue.on('empty', ...)` call with `await once(queue, 'empty')`.

The `once()` function creates a `Promise` instance that is fulfilled when the given `EventEmitter` (queue in our case) emits the given event ('empty' in our case) or is rejected if `EventEmitter` emits 'error' while waiting.

This approach allows us to use top-level `await` to wait for the completion of the crawling from our CLI helper. This is not a fundamental change, but a nice helper worth knowing when working with events and `async/await`.

Also, note how we had to rename the 'error' event to 'taskError' since the 'error' event has a special meaning. In fact, it is considered a global unrecoverable error (and therefore rejects the promise returned by `once()`). In our case, we want to continue the crawling even if a URL fails, so we had to introduce a different name to handle that.

The problem with infinite recursive promise resolution chains

At this point in the chapter, you should have a strong understanding of how promises work and how to use them to implement the most common control flow constructs. This is the right time to discuss an advanced topic that every professional Node.js developer should know and understand. This advanced topic is about a memory leak caused by infinite `Promise` resolution chains. The bug affects the actual `Promises/A+` specification, so no compliant implementation is immune.

It is quite common in programming to have tasks that don't have a predefined ending or take as an input a potentially infinite array of data. We can include in this category things such as the encoding/decoding of live audio/video streams, the processing of live cryptocurrency market data, and the monitoring of IoT sensors. But you don't need such complex scenarios to run into the same challenges. In fact, these situations can arise even in more ordinary code. For instance, when using functional programming patterns extensively, it's easy to create recursive or self-repeating operations that never terminate unless explicitly stopped.

To take a simple example, let's consider the following code, which defines a simple infinite operation using promises:

```
function leakingLoop() {  
  return delay(1)  
    .then(() => {  
      console.log(`Tick ${Date.now()}`)  
      return leakingLoop()  
    })  
}
```

The `leakingLoop()` function that we just defined uses the `delay()` function (which we created at the beginning of this chapter) to simulate an asynchronous operation. When the given number of milliseconds has elapsed, we print the current timestamp and we invoke `leakingLoop()` recursively to start the operation over again. The interesting part is that the `Promise` returned by `leakingLoop()` never resolves because its status depends on the next invocation of `leakingLoop()`, which in turn depends on the next invocation of `leakingLoop()`, and so on. This situation creates a chain of promises that never settle, and it will cause a memory leak in `Promise` implementations that strictly follow the `Promises/A+` specification, including JavaScript ES6 promises.

To demonstrate the leak, we can try running the `leakingLoop()` function many times to accentuate the effects of the leak:

```
for (let i = 0; i < 1e6; i++) {  
  leakingLoop()  
}
```

Then we can take a look at the memory footprint of the process using our favorite process inspector and notice how it grows indefinitely until (after a few minutes) the process crashes entirely.

The solution to the problem is to break the chain of the `Promise` resolution. We can do that by making sure that the status of the `Promise` returned by `leakingLoop()` does not depend on the promise returned by the next invocation of `leakingLoop()`.

We can ensure that by simply removing a return instruction:

```
function nonLeakingLoop() {  
  delay(1)  
    .then(() => {  
      console.log(`Tick ${Date.now()}`)  
      nonLeakingLoop()  
    })  
}
```

Now, if we use this new function in our sample program, we should see that the memory footprint of the process will go up and down, following the schedule of the various runs of the garbage collector, which means that there is no memory leak.

However, the solution we have just proposed radically changes the behavior of the original `leakingLoop()` function. In particular, this new function won't propagate eventual errors produced deeply within the recursion, since there is no link between the status of the various promises. This inconvenience may be mitigated by adding some extra logging within the function. But sometimes the new behavior itself may not be an option. So, a possible solution involves wrapping the recursive function with a Promise constructor, such as in the following code sample:

```
function nonLeakingLoopWithErrors() {
  return new Promise((_resolve, reject) => {
    (function internalLoop () {
      delay(1)
        .then(() => {
          console.log(`Tick ${Date.now()}`)
          internalLoop()
        })
        .catch(err => {
          reject(err)
        })
    })()
  })
}
```

In this case, we still don't have any link between the promises created at the various stages of the recursion; however, the Promise returned by the `nonLeakingLoopWithErrors()` function will still reject if any asynchronous operation fails, no matter at what depth in the recursion that happens.

A third solution makes use of `async/await`. In fact, with `async/await`, we can *simulate* a recursive Promise chain with a simple infinite while loop, such as the following:

```
async function nonLeakingLoopAsync() {
  while (true) {
    await delay(1)
    console.log(`Tick ${Date.now()}`)
  }
}
```

In this function too, we preserve the behavior of the original recursive function, whereby any error thrown by the asynchronous task (in this case, `delay()`) is propagated to the original function caller.

We should note that we would still have a memory leak if, instead of a while loop, we chose to implement the async/await solution with an actual asynchronous recursive step, such as the following:

```
async function leakingLoopAsync() {  
  await delay(1)  
  console.log(`Tick ${Date.now()}`)  
  return leakingLoopAsync()  
}
```

The preceding code would still create an infinite chain of promises that never resolve, and therefore it's still affected by the same memory leak issue of the equivalent promise-based implementation.



If you are interested in knowing more about the memory leak discussed in this section, you can check the related Node.js issue at nodejsdp.link/node-6673 or the related issue in the Promises/A+ GitHub repository at nodejsdp.link/promisesapplus-memleak.

So, the next time you are building an infinite promise chain, remember to double-check whether there are the conditions for creating a memory leak, as you learned in this section. If that's the case, you can apply one of the proposed solutions, making sure to choose the one that is best suited to your context.

Summary

In this chapter, we learned how to use promises and async/await syntax to write asynchronous code that is more concise, cleaner, and easier to read.

As we've seen, promises and async/await greatly simplify the serial execution flow, which is the most commonly used control flow. In fact, with async/await, writing a sequence of asynchronous operations is almost as easy as writing synchronous code. Running some asynchronous operations concurrently is also very easy thanks to `Promise.all()` and `Promise.allSettled()`.

But the advantages of using promises and async/await don't stop here. We learned that they provide a transparent shield against tricky situations such as code with mixed synchronous/asynchronous behavior (a.k.a. *Zalgo*, which we discussed in *Chapter 3, Callbacks and Events*). On top of that, error management with promises and async/await is much more intuitive and leaves less room for mistakes (such as forgetting to forward errors, which is a serious source of bugs in code using callbacks).

In terms of patterns and techniques, we should definitely keep in mind the chain of promises (to run tasks in series), promisification, and the Producer-Consumer pattern (with our task queue example). Also, pay attention when using `Array.forEach()` with async/await (you are probably doing it wrong), and keep in mind the difference between a simple `return` and `return await` in async functions.

Callbacks are still widely used in the Node.js and JavaScript world. We find them in legacy APIs, in code that interacts with native libraries, or when there is a need to micro-optimize particular routines. That's why they are still relevant to us, Node.js developers; however, for most of our day-to-day programming tasks, promises and `async/await` are a huge step ahead compared to callbacks, and therefore they are now the de facto standard for dealing with asynchronous code in Node.js. That's why we will be using promises and `async/await` throughout the rest of the book too, to write our asynchronous code.

In the next chapter, we will explore another fascinating topic relative to asynchronous code execution, which is also another fundamental building block in the whole Node.js ecosystem—that is, streams.

Exercises

- **5.1 Dissecting `Promise.all()`:** Implement your own version of `Promise.all()` leveraging promises, `async/await`, or a combination of the two. The function must be functionally equivalent to its original counterpart.
- **5.2 `TaskQueue` with promises:** Migrate the `TaskQueue` class internals from promises to `async/await` where possible. Hint: you won't be able to use `async/await` everywhere.
- **5.3 Producer-consumer with promises:** Update the `TaskQueuePC` class internal methods so that they use just promises, removing any use of the `async/await` syntax. Hint: the infinite loop must become an asynchronous recursion. Beware of the recursive `Promise` resolution memory leak!
- **5.4 An asynchronous `map()`:** Implement a concurrent asynchronous version of `Array.map()` that supports promises and a concurrency limit. The function should not directly leverage the `TaskQueue` or `TaskQueuePC` classes we presented in this chapter, but it can use the underlying patterns. The function, which we will define as `mapAsync(iterable, callback, concurrency)`, will accept the following as inputs:
 - An iterable, such as an array.
 - A callback, which will receive as the input each item of the iterable (exactly like in the original `Array.map()`) and can return either a `Promise` or a simple value.
 - A concurrency, which defines how many items in the iterable can be processed by callback concurrently at each given time.

6

Coding with Streams

We briefly touched on streams in *Chapter 3, Callbacks and Events* and *Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await* as an option to make some of our code a bit more robust. Now, it's finally time to dive in! We are here to talk about streams: one of the most important components and patterns of Node.js. There is a motto in the community that goes, “stream all the things!”, and this alone should be enough to describe the role of streams in Node.js. Dominic Tarr, an early contributor to the Node.js community, defined streams as “Node’s best and most misunderstood idea.” There are different reasons that make Node.js streams so attractive; it’s not just related to technical properties, such as performance or efficiency, but it’s more about their elegance and the way they fit perfectly into the Node.js philosophy. Yet, despite their potential, streams remain underutilized in the broader developer community. Many find them intimidating and choose to avoid them altogether. This chapter is here to change that. We’ll explore streams in depth, highlight their advantages, and present them in a clear and approachable way, making their power accessible to all developers.



But before we dive in, let’s take a short break for an author’s note (Luciano here). Streams are one of my favourite topics in Node.js, and I can’t help but share a story from my career where streams truly saved the day.

I was working for a network security company on a team developing a cloud application. The application’s purpose was to collect network metadata from physical devices monitoring traffic in corporate environments. Imagine recording all the connections between hosts in the network, which protocols they’re using, and how much data they’re transferring. This data could help spot the movement of an attacker in the network or uncover attempts at data exfiltration. The idea was simple yet powerful: in the event of a security incident, our customers could log into our platform, browse through the recorded metadata, and figure out exactly what happened, enabling them to take action quickly.

As you might imagine, this required continuously streaming a significant amount of data from devices at customer sites to our cloud-based web server. In the spirit of keeping things simple and shipping fast, our initial implementation of the data *collector* (the HTTP server receiving and storing metadata) used a buffered approach.



Devices would send network metadata in frames every minute, each containing all the observations from the previous 60 seconds.

Here's how it worked: we'd load the entire frame into memory as it arrived, and only after receiving the complete frame would we write it to persistent storage. This worked well in the beginning because we were only serving small customers who generated relatively modest amounts of metadata, even during peak traffic.

But when we rolled out the solution to a larger customer, things started to break down. We noticed occasional failures in the collector and, worse, gaps in the stored data. After digging into the issue, we discovered that the collector was crashing due to excessive memory usage. If a customer generated a particularly large frame, the system couldn't handle it, leading to data loss.

This was a serious problem. Our entire value proposition depended on being able to reliably store and retrieve network metadata for forensic analysis. If customers couldn't trust us to preserve their data, the platform was effectively useless.

We needed a fix, and fast. The root of the problem was clear: buffering entire frames in memory was a rookie mistake. The solution? Keep the memory footprint low by processing data in smaller chunks and writing them to storage incrementally.

Enter Node.js streams. With streams, we could process data piece by piece as it arrived, rather than waiting for the entire frame. After refactoring our code to use streams, we were able to handle terabytes of data daily without breaking a sweat. The system's latency improved dramatically: customers could see their data in the cloud in under two minutes. We also cut costs by using smaller machines with less memory, and the new implementation was far more elegant and maintainable, thanks to the composable nature of the Node.js streams API.

While this might sound like a specific use case, the lessons here apply broadly. Any time you're moving data from A to B, especially when dealing with unpredictable volumes or when early results are valuable, Node.js streams are an invaluable tool.

I promise you that once you learn the fundamentals of streams, you'll appreciate their power and see many opportunities to leverage them in your applications!

This chapter aims to provide a complete understanding of Node.js streams. The first half of this chapter serves as an introduction to the main ideas, the terminology, and the libraries behind Node.js streams. In the second half, we will cover more advanced topics and, most importantly, we will explore useful streaming patterns that can make your code more elegant and effective in many circumstances.

In this chapter, you will learn about the following topics:

- Why streams are so important in Node.js
- Understanding, using, and creating streams
- Streams as a programming paradigm: leveraging their power in many different contexts and not just for I/O
- Streaming patterns and connecting streams together in different configurations

Without further ado, let's discover together why streams are one of the cornerstones of Node.js.

Discovering the importance of streams

In an event-based platform such as Node.js, the most efficient way to handle I/O is in real time, consuming the input as soon as it is available and sending the output as soon as the application produces it.

In this section, we will give you an initial introduction to Node.js streams and their strengths. Please bear in mind that this is only an overview, as a more detailed analysis on how to use and compose streams will follow later in this chapter.

Buffering versus streaming

Almost all the asynchronous APIs that we've seen so far in this book work using *buffer mode*. For an input operation, buffer mode causes all the data coming from a resource to be collected into a buffer until the operation is completed; it is then passed back to the caller as one single blob of data. The following diagram shows a visual example of this paradigm:



Figure 6.1: Buffering

In *Figure 6.1*, we aim to transfer data containing the string “Hello Node.js” from a resource to a consumer. This process illustrates the concept of buffer mode, where all data is accumulated in a buffer before being consumed. At time $t1$, the first chunk of data, “Hello N,” is received from the resource and stored in the buffer. At $t2$, the second chunk, “ode.js,” arrives, completing the read operation. With the entire string now fully accumulated in the buffer, it is sent to the consumer at $t3$.

Streams provide a different approach, allowing data to be processed incrementally as it arrives from the resource. This is shown in the following diagram:

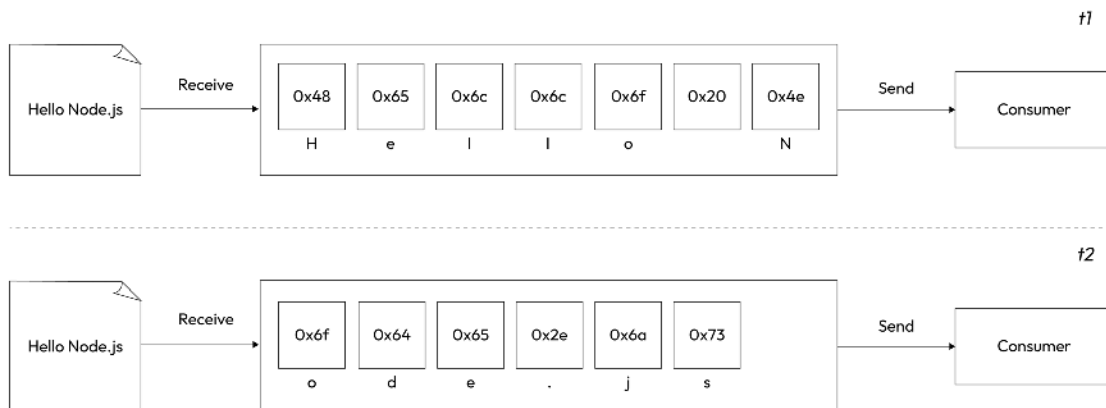


Figure 6.2: Streaming

This time, *Figure 6.2* shows that, as soon as each new chunk of data is received from the resource, it is immediately passed to the consumer, who now has the chance to process it straight away, without waiting for all the data to be collected in the buffer.

But what are the differences between these two approaches? Purely from an efficiency perspective, streams are generally more efficient in terms of space (memory usage) and sometimes even in terms of computation clock time. However, Node.js streams have another important advantage: **composability**. Let’s now see what impact these properties have on the way we design and write our applications.

Spatial efficiency

First of all, streams allow us to do things that would not be possible by buffering data and processing it all at once. For example, consider the case in which we have to read a very big file, let’s say, in the order of hundreds of megabytes or even gigabytes. Clearly, using an API that returns a big buffer when the file is completely read is not a good idea. Imagine reading a few of these big files concurrently; our application would easily run out of memory. Besides that, buffers in V8 are limited in size. You cannot allocate more than a few gigabytes of data, so we may hit a wall way before running out of physical memory.



The actual maximum size of a buffer changes across platforms and versions of Node.js. If you are curious to find out what the limit in bytes is in a given platform, you can run this code:

```
import buffer from 'node:buffer'
console.log(buffer.constants.MAX_LENGTH)
```

Gzipping using a buffered API

To make a concrete example, let's consider a simple command-line application that compresses a file using the GZIP format. Using a buffered API, such an application will look like the following in Node.js (error handling is omitted for brevity):

```
// gzip-buffer.js
import { readFile, writeFile } from 'node:fs/promises'
import { gzip } from 'node:zlib'
import { promisify } from 'node:util'
const gzipPromise = promisify(gzip) // note: gzip is a callback-based function

const filename = process.argv[2]

const data = await readFile(filename)
const gzippedData = await gzipPromise(data)
await writeFile(`${filename}.gz`, gzippedData)

console.log('File successfully compressed')
```

Now, we can try to run it with the following command:

```
node gzip-buffer.js <path to file>
```

If we choose a file that is big enough (for instance, 8 GB or more), we will most likely receive an error message saying that the file we are trying to read is bigger than the maximum allowed buffer size:

```
RangeError [ERR_FS_FILE_TOO_LARGE]: File size is greater than possible Buffer
```

That's exactly what we expected, and it's a symptom of the fact that we are using the wrong approach.



Note that the error happens when we execute `readFile()`. This is where we are taking the entire content of the file and loading it into a buffer in memory. Node.js will check the file size before starting to load its content. If the file is too big to fit in a buffer, then we will be presented with the `ERR_FS_FILE_TOO_LARGE` error.

Gzipping using streams

The simplest way we have to fix our Gzip application and make it work with big files is to use a streaming API. Let's see how this can be achieved. Let's write a new module with the following code:

```
// gzip-stream.js
import { createReadStream, createWriteStream } from 'node:fs'
import { createGzip } from 'node:zlib'

const filename = process.argv[2]

createReadStream(filename)
  .pipe(createGzip())
  .pipe(createWriteStream(`${filename}.gz`))
  .on('finish', () => console.log('File successfully compressed'))
```

“Is that it?” you may ask. Yes! As we said, streams are amazing because of their interface and composability, thus allowing clean, elegant, and concise code. We will see this in a while in more detail, but for now, the important thing to realize is that the program will run smoothly against files of any size and with constant memory utilization. Try it yourself (but consider that compressing a big file may take a while).



Note that, in the previous example, we omitted error handling for brevity. We will discuss the nuances of proper error handling with streams later in this chapter. Until then, be aware that most examples will be lacking proper error handling.

Time efficiency

We could talk about the time efficiency of streams in abstract terms, but it's probably much easier to understand why streams are so advantageous by seeing them in action. Let's work on something practical to appreciate how streams save both time and resources in real-world scenarios.

Let's build a new client-server application! Our goal is to create a client that reads a file from the file system, compresses it, and sends it to a server over HTTP. The server will then receive the file, decompress it, and save it to a local folder. This way, we're creating our very own homemade file transfer utility!

To achieve this, we have two options: we can use a **buffer-based API** or leverage **streams**. If we don't expect to transfer large files, both approaches will get the job done, but they differ significantly in how the data is processed and transferred.

If we were to use a buffered API for this, the client would first need to load the entire file into memory as a buffer. Once the file is fully loaded, it will compress the data, creating a second buffer containing the compressed version. Only after these steps can the client send the compressed data to the server.

On the server side, a buffered approach would involve accumulating all the incoming data from the HTTP request into a buffer. Once all the data has been received, the server would decompress it into another buffer containing the uncompressed data, which would then be saved to disk.

While this works, a better approach uses **streams**. With streams, the client can start compressing and sending chunks of data as soon as they are read from the file system. Similarly, the server can decompress each chunk of data as soon as it arrives, eliminating the need to wait for the entire file. As a bonus, we have already seen how streams give us the ability to handle arbitrarily large files.

Let's dive into how we can build a simple version of this stream-based approach, starting with the server:

```
// gzip-receive.js
import { createServer } from 'node:http'
import { createWriteStream } from 'node:fs'
import { createGunzip } from 'node:zlib'
import { basename, join } from 'node:path'

const server = createServer((req, res) => {
  const filename = basename(req.headers['x-filename'])
  const destFilename = join(import.meta.dirname, 'received_files',
    filename)
  console.log(`File request received: ${filename}`)
  req
    .pipe(createGunzip())
    .pipe(createWriteStream(destFilename))
    .on('finish', () => {
      res.writeHead(201, { 'content-type': 'text/plain' })
      res.end('OK\n')
      console.log(`File saved: ${destFilename}`)
    })
})

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

In the preceding example, we are setting up an HTTP server that listens for incoming file uploads, decompresses them, and saves them to disk. The key part of this server is the handler function (the one passed to the `createServer()` function), where two important objects, `req` (the request) and `res` (the response), come into play. These objects are both streams:

- `req` represents the incoming request from the client to the server. In this case, it carries the compressed file data being sent by the client.
- `res` represents the outgoing response from the server back to the client.

The focus here is on `req`, which acts as the source stream. The code processes `req` by:

- Decompressing it using `createGunzip()`.
- Saving it to disk with `createWriteStream()` in a directory named `received_files` (in the same folder as this code example).

The `pipe()` calls link these steps together, creating a smooth flow of data from the incoming request, through decompression, to the file on disk. Don't worry too much about the `pipe()` syntax for now—we'll cover it in more detail later in the chapter.

When all the data has been written to disk, the `finish` event is triggered. At this point, the server responds to the client with a status code of 201 (Created) and a simple "OK" message, indicating that the file has been successfully received and saved.

Finally, the server listens for connections on port 3000, and a message is logged to confirm it's running.



In our server application, we use `basename()` to remove any path from the name of a received file (e.g., `basename("/path/to/file")` would give us "file"). This is an important security measure to ensure that files are saved within our `received_files` folder. Without `basename()`, a malicious user could create a request that escapes the application's folder, leading to potentially serious consequences like being able to overwrite system files and inject malicious code. For example, imagine if the provided filename was something like `../../usr/bin/node`. An attacker could eventually guess a relative path to overwrite `/usr/bin/node`, replacing the Node.js interpreter with any executable file they want. Scary, right? This type of attack is called a **path traversal attack** (or directory traversal). You can read more about it here: nodejsdp.link/path-traversal.



Note that here we are not following the most conventional way to perform file uploads over HTTP. In fact, generally, this feature is implemented using a slightly more advanced and standard protocol that requires encoding the source data using the `multipart/form-data` specification (nodejsdp.link/multipart). This specification allows you to send one or more files and their respective file names using fields encoded in the body. In our simpler implementation, the body of the request contains no metadata, but only the gzipped bytes of the original file; therefore, we must specify the filename somewhere else. That's why we provide a custom header called `x-filename`.

Now that we are done with the server, let's write the corresponding client code:

```
// gzip-send.js
import { request } from 'node:http'
import { createGzip } from 'node:zlib'
import { createReadStream } from 'node:fs'
import { basename } from 'node:path'
```

```
const filename = process.argv[2]
const serverHost = process.argv[3]

const httpRequestOptions = {
  hostname: serverHost,
  port: 3000,
  path: '/',
  method: 'POST',
  headers: {
    'content-type': 'application/octet-stream',
    'content-encoding': 'gzip',
    'x-filename': basename(filename),
  },
}

const req = request(httpRequestOptions, res => {
  console.log(`Server response: ${res.statusCode}`)
})

createReadStream(filename)
  .pipe(createGzip())
  .pipe(req)
  .on('finish', () => {
    console.log('File successfully sent')
  })
```

In the preceding code, we implement the client side of our file transfer system. Its goal is to read a file from the local file system, compress it, and send it to the server using an HTTP POST request. Here's how it works:

The client reads the filename (to be sent) and the server's hostname (serverHost) from the command-line arguments. These values are then used to configure the `httpRequestOptions` object, which defines the details of the HTTP request, including:

- The server hostname and port
- The request path and method
- The headers, including information about the file name (x-filename), content type, and the fact that the content is gzip-compressed.
- The actual HTTP request (req) that is created using the `request()` function. This object is a stream that represents an HTTP request going from the client to the server.

The source file is read using `createReadStream()`, compressed with `createGzip()`, and then sent to the server by piping the resulting stream into `req`. This creates a continuous flow of data from the file on disk, through compression, and finally to the server.

When all the data has been sent, the `finish` event is triggered on the request stream. At this point, a confirmation message (“*File successfully sent*”) is logged.

Meanwhile, the server’s response is handled in the callback provided to `request()`. Once the server responds, its status code is logged to the console, allowing the client to confirm that the operation was completed successfully.

Now, to try out the application, let’s first start the server using the following command:

```
node gzip-receive.js
```

Then, we can launch the client by specifying the file to send and the address of the server (for example, `localhost`):

```
node gzip-send.js <path to file> localhost
```

If we choose a sufficiently large file, we can observe how the data flows from the client to the server. The target file will appear in the `received_files` folder before the “*File successfully sent*” message is displayed on the client. This is because, as the compressed file is being sent over HTTP, the server is already decompressing it and saving it on the disk.

However, we still haven’t addressed why this paradigm, with its continuous data flow, is more efficient than using a buffered API. *Figure 6.3* should make this concept easier to grasp:

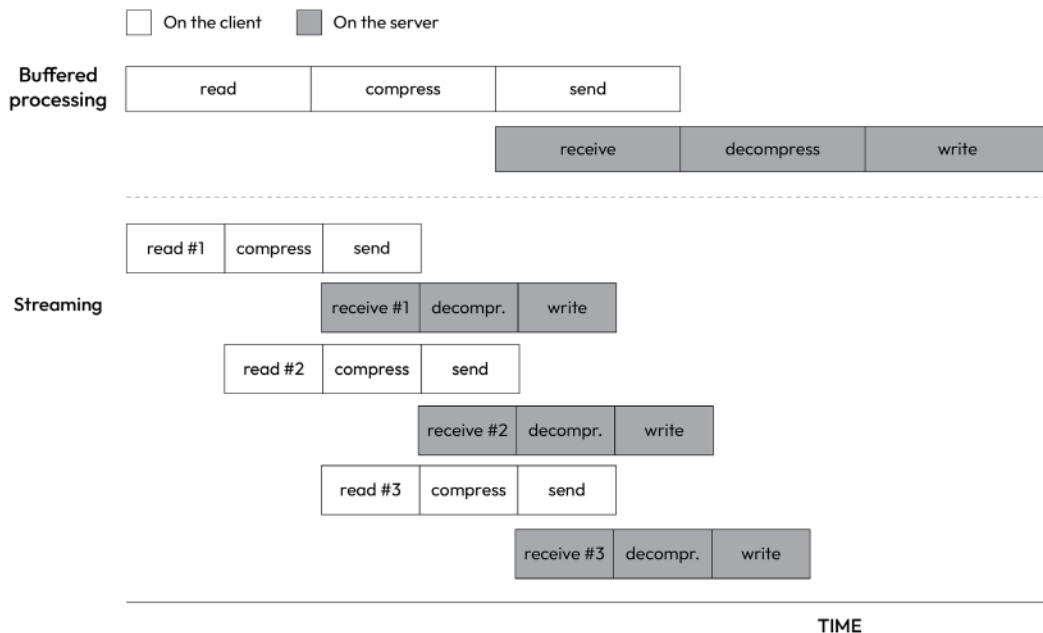


Figure 6.3: Buffering and streaming compared

When a file is processed, it goes through a number of sequential steps:

1. [Client] Read from the filesystem
2. [Client] Compress the data
3. [Client] Send it to the server
4. [Server] Receive from the client
5. [Server] Decompress the data
6. [Server] Write the data to disk

To complete the processing, we have to go through each stage like in an assembly line, in sequence, until the end. In *Figure 6.3*, we can see that, using a buffered API, the process is entirely sequential. To compress the data, we first must wait for the entire file to be read, then, to send the data, we have to wait for the entire file to be both read and compressed, and so on.

Using streams, the assembly line is kicked off as soon as we receive the first chunk of data, without waiting for the entire file to be read. But more amazingly, when the next chunk of data is available, there is no need to wait for the previous set of tasks to be completed; instead, another assembly line is launched concurrently. This works perfectly because each task that we execute is asynchronous, so it can be executed concurrently by Node.js. The only constraint is that the order in which the chunks arrive at each stage must be preserved. The internal implementation of Node.js streams takes care of maintaining the order for us.

As we can see from *Figure 6.3*, the result of using streams is that the entire process takes less time, because we waste no time waiting for all the data to be read and processed all at once.



This section might make it seem like streams are *always* faster than using a buffered approach. While that's often true (as in the example we just covered), it's not guaranteed. Streams are designed for memory efficiency, not necessarily speed. The abstraction they provide can add overhead, which might slow things down. If all the data you need fits in memory, is already loaded, and doesn't need to be transferred between processes or systems, processing it directly without streams is likely to give you faster results.

Composability

The code we've seen so far demonstrates how streams can be composed using the `pipe()` method. This method allows us to connect different processing units, each responsible for a single functionality, in true Node.js style. Streams can do this because they share a consistent interface, making them compatible with one another at the API level. The only requirement is that the next stream in the pipeline must support the data type produced by the previous stream (binary data or objects, as we'll explore later in this chapter).

To further demonstrate the composability of Node.js streams, let's try to add an encryption layer to the `gzip-send/gzip-receive` application we built earlier. This will require just a few small changes to both the client and the server.

Adding client-side encryption

Let's start with the client:

```
// crypto-gzip-send.js
// ...
import { createCipheriv, randomBytes } from 'node:crypto' // 1
// ...
const secret = Buffer.from(process.argv[4], 'hex') // 2
const iv = randomBytes(16) // 3
// ...
```

Let's review what we changed here:

1. First of all, we import the `createCipheriv()` Transform stream and the `randomBytes()` function from the `node:crypto` module.
2. We get the server's encryption secret from the command line. We expect the string to be passed as a hexadecimal string, so we read this value and load it in memory using a buffer set to hex mode.
3. Finally, we generate a random sequence of bytes that we will be using as an initialization vector (`nodejsdp.link/iv`) for the file encryption.



An Initialization Vector (IV) is a bit like giving a deck of cards a different shuffle before dealing them, even if you're always using the same deck. By starting each round with a different shuffle, it becomes much harder for someone watching your hands closely to predict the cards you're holding. In cryptography, the IV sets the initial state for encryption. It's usually random or unique, ensuring that encrypting the same message twice with the same key produces different results. This helps prevent attackers from identifying patterns. Note that the IV is required for later decryption. The message recipient must know both the key and the IV to decrypt the message, and only the key must remain secret (generally, the IV is transferred together with the encrypted message, while the key is exchanged in some other secure way). The card-shuffling analogy isn't perfect, but it helps illustrate how starting with a different configuration each time can significantly increase security.

Now, we can update the piece of code responsible for creating the HTTP request:

```
const http RequestOptions = {
  hostname: serverHost,
  headers: {
    'content-type': 'application/octet-stream',
    'content-encoding': 'gzip',
    'x-filename': basename(filename),
    'x-initialization-vector': iv.toString('hex') // 1
  }
}
```

```
// ...

const req = request(httpRequestOptions, (res) => {
  console.log(`Server response: ${res.statusCode}`)
})

createReadStream(filename)
  .pipe(createGzip())
  .pipe(createCipheriv('aes192', secret, iv)) // 2
  .pipe(req)

// ...
```

The main changes here are:

1. We pass the initialization vector to the server as an HTTP header.
2. We encrypt the data, just after the Gzip phase.

That's all for the client side.

Adding server-side decryption

Let's now refactor the server. The first thing that we need to do is import some utility functions from the core `node:crypto` module, which we can use to generate a random encryption key (the secret):

```
// crypto-gzip-receive.js
// ...
import { createDecipheriv, randomBytes } from 'node:crypto'

const secret = randomBytes(24)
console.log(`Generated secret: ${secret.toString('hex')}`)
```

The generated secret is printed to the console as a hex string so that we can share that with our clients.

Now, we need to update the file reception logic:

```
const server = createServer((req, res) => {
  const filename = basename(req.headers['x-filename'])
  const iv = Buffer.from(req.headers['x-initialization-vector'], 'hex') // 1
  const destFilename = join('received_files', filename)
  console.log(`File request received: ${filename}`)
```

```
req
  .pipe(createDecipheriv('aes192', secret, iv)) // 2
  .pipe(createGunzip())
  .pipe(createWriteStream(destFilename))
  // ...
```

Here, we are applying two changes:

1. We have to read the encryption **initialization vector** sent by the client.
2. The first step of our streaming pipeline is now responsible for decrypting the incoming data using the `createDecipheriv` Transform stream from the `crypto` module.

With very little effort (just a few lines of code), we added an encryption layer to our application; we simply had to use some already available Transform streams (`createCipheriv` and `createDecipheriv`) and included them in the stream processing pipelines for the client and the server. In a similar way, we can add and combine other streams, as if we were playing with LEGO bricks.

The main advantage of this approach is reusability, but as we can see from the code so far, streams also enable cleaner and more modular code. For these reasons, streams are often used not just to deal with pure I/O, but also to simplify and modularize code.

Now that you have had an appetizer of what using streams tastes like, we are ready to explore, in a more structured way, the different types of streams available in Node.js.



In this implementation, we used encryption as an example to demonstrate the composability of streams. Since our client-server communication relies on the HTTP protocol, a more standard and possibly simpler approach would have been to use HTTPS by simply switching from the `node:http` module to the `node:https` module. Regardless of which implementation you decide to use, make sure that, if you are transferring data over a network, you use some strong form of encryption. Never transfer unencrypted data over a network!

Getting started with streams

In the previous section, we learned why streams are so powerful, but also that they are everywhere in Node.js, starting from its core modules. For example, we have seen that the `fs` module has `createReadStream()` for reading from a file and `createWriteStream()` for writing to a file, the HTTP request and response objects are essentially streams, the `zlib` module allows us to compress and decompress data using a streaming interface, and, finally, even the `crypto` module exposes some useful streaming primitives like `createCipheriv` and `createDecipheriv`.

Now that we know why streams are so important, let's take a step back and start to explore them in more detail.

Anatomy of streams

Every stream in Node.js is an implementation of one of the four base abstract classes available in the stream core module:

- Readable
- Writable
- Duplex
- Transform

Each stream class is also an instance of `EventEmitter`. Streams, in fact, can produce several types of events, such as `end` when a `Readable` stream has finished reading, `finish` when a `Writable` stream has completed writing (we have already seen this one in some of the examples before), or `error` when something goes wrong.

One reason why streams are so flexible is the fact that they can handle not just binary data, but almost any JavaScript value. In fact, they support two operating modes:

- **Binary mode:** To stream data in the form of chunks, such as buffers or strings
- **Object mode:** To stream data as a sequence of discrete objects (allowing us to use almost any JavaScript value)

These two operating modes allow us to use streams not just for I/O, but also as a tool to elegantly compose processing units in a functional fashion, as we will see later in this chapter.

Let's start our deep dive into Node.js streams by introducing the class of `Readable` streams.

Readable streams

A `Readable` stream represents a source of data. In Node.js, it's implemented using the `Readable` abstract class, which is available in the `stream` module.

Reading from a stream

There are two approaches to receive the data from a `Readable` stream: **non-flowing** (or **paused**) and **flowing**. Let's analyze these modes in more detail.

The non-flowing mode

The non-flowing or paused mode is the default pattern for reading from a `Readable` stream. It involves attaching a listener to the stream for the `readable` event, which signals the availability of new data to read. Then, in a loop, we read the data continuously until the internal buffer is emptied. This can be done using the `read()` method, which synchronously reads from the internal buffer and returns a `Buffer` object representing the chunk of data. The `read()` method has the following signature:

```
readable.read([size])
```

Using this approach, the data is pulled from the stream on demand.

To show how this works, let's create a new module named `read-stdin.js`, which implements a simple program that reads from the standard input (which is also a Readable stream) and echoes everything back to the standard output:

```
process.stdin
  .on('readable', () => {
    let chunk
    console.log('New data available')
    while ((chunk = process.stdin.read()) !== null) {
      console.log(
        `Chunk read (${chunk.length} bytes): "${chunk.toString()}"`
      )
    }
  })
  .on('end', () => console.log('End of stream'))
```

The `read()` method is a synchronous operation that pulls a data chunk from the internal buffers of the Readable stream. The returned chunk is, by default, a `Buffer` object if the stream is working in binary mode.



In a Readable stream working in binary mode, we can read strings instead of buffers by calling `setEncoding(encoding)` on the stream, and providing a valid encoding format (for example, `utf8`). This approach is recommended when streaming UTF-8 text data, as the stream will properly handle multibyte characters, doing the necessary buffering to make sure that no character ends up being split into separate chunks. In other words, every chunk produced by the stream will be a valid UTF-8 sequence of bytes.

Note that you can call `setEncoding()` as many times as you want on a Readable stream, even after you've started consuming the data from the stream. The encoding will be switched dynamically on the next available chunk. Streams are inherently binary; encoding is just a view over the binary data that is emitted by the stream.

The data is read solely from within the Readable listener, which is invoked as soon as new data is available. The `read()` method returns `null` when there is no more data available in the internal buffers; in such a case, we have to wait for another readable event to be fired, telling us that we can read again, or wait for the end event that signals the end of the stream. When a stream is working in binary mode, we can also specify that we are interested in reading a specific amount of data by passing a size value to the `read()` method. This is particularly useful when implementing network protocols or when parsing specific data formats.

Now, we are ready to run the `read-stdin.js` module and experiment with it. Let's type some characters into the console and then press *Enter* to see the data echoed back into the standard output. To terminate the stream and hence generate a graceful end event, we need to insert an EOF (end-of-file) character (using *Ctrl* + *Z* on Windows or *Ctrl* + *D* on Linux and macOS).



We can also try to connect our program with other processes. This is possible using the pipe operator (`|`), which redirects the standard output of a program to the standard input of another. For example, we can run a command such as the following:

```
cat <path to a file> | node read-stdin.js
```

This is an amazing demonstration of how the streaming paradigm is a universal interface that enables our programs to communicate, regardless of the language they are written in.

Flowing mode

Another way to read from a stream is by attaching a listener to the data event. This will switch the stream into using **flowing mode**, where the data is not pulled using `read()`, but instead is pushed to the data listener as soon as it arrives. For example, the `read-stdin.js` application that we created earlier will look like this using flowing mode:

```
process.stdin
  .on('data', (chunk) => {
    console.log('New data available')
    console.log(
      `Chunk read (${chunk.length} bytes): "${chunk.toString()}"`
    )
  })
  .on('end', () => console.log('End of stream'))
```

Flowing mode offers less flexibility to control the flow of data compared to non-flowing mode. The default operating mode for streams is non-flowing, so to enable flowing mode, it's necessary to attach a listener to the data event or explicitly invoke the `resume()` method. To temporarily stop the stream from emitting data events, we can invoke the `pause()` method, causing any incoming data to be cached in the internal buffer. Calling `pause()` will switch the stream back to non-flowing mode.

Async iterators

Readable streams are also async iterators; therefore, we could rewrite our `read-stdin.js` example as follows:

```
for await (const chunk of process.stdin) {
  console.log('New data available')
  console.log(`Chunk read (${chunk.length} bytes): "${chunk.toString()}"`)
}

console.log('End of stream')
```

We will discuss async iterators in greater detail in *Chapter 9, Behavioral Design Patterns*, so don't worry too much about the syntax in the preceding example for now. What's important to know is that you can also consume data from a Readable stream using this convenient `for await ... of` syntax.

Implementing Readable streams

Now that we know how to read from a stream, the next step is to learn how to implement a new custom Readable stream. To do this, it's necessary to create a new class by inheriting the prototype Readable from the stream module. The concrete stream must provide an implementation of the `_read()` method, which has the following signature:

```
readable._read(size)
```

The internals of the Readable class will call the `_read()` method, which, in turn, will start to fill the internal buffer using `push()`:

```
readable.push(chunk)
```



Please note that `read()` is a method called by the stream consumers, while `_read()` is a method to be implemented by a stream subclass and should never be called directly. The underscore prefix is used to indicate that the method is not to be considered public and should not be called directly.

To demonstrate how to implement new Readable streams, we can try to implement a stream that generates random strings. Let's create a new module that contains the code of our random string generator:

```
// random-stream.js
import { Readable } from 'node:stream'
import Chance from 'chance' // v1.1.12

const chance = new Chance()

export class RandomStream extends Readable {
  constructor(options) {
    super(options)
    this.emittedBytes = 0
  }

  _read(size) {
    const chunk = chance.string({ length: size }) // 1
    this.push(chunk, 'utf8') // 2
    this.emittedBytes += chunk.length
    if (chance.bool({ likelihood: 5 })) { // 3
      this.push(null)
    }
  }
}
```

For this example, we are using a third-party module from npm called `chance` (nodejsdp.link/chance), which is a library for generating all sorts of random values, from numbers to strings to entire sentences.



Note that `chance` is not cryptographically secure, which means it can be used for tests or simulations but not to generate tokens, passwords, or other security-related purposes.

We start by defining a new class called `RandomStream`, which specifies `Readable` as its parent. In the constructor of this class, we have to invoke `super(options)`, which will call the constructor of the parent class (`Readable`), initializing the stream's internal state.



If you have a constructor that only invokes `super(options)`, you can remove it, since the class will inherit the parent constructor by default. Just be careful to remember to call `super(options)` every time you need to write a custom constructor.

The possible parameters that can be passed through the `options` object include the following:

- The encoding argument, which is used to convert buffers into strings (defaults to `null`)
- A flag to enable object mode (`objectMode`, defaults to `false`)
- The upper limit of the data stored in the internal buffer, after which no more reading from the source should be done (`highWaterMark`, defaults to 16KB)

Inside the constructor, we initialized an instance variable: `emittedBytes`. We will be using this variable to keep track of how many bytes have been emitted so far from the stream. This is going to be useful for debugging, but it's not a requirement when creating `Readable` streams.

Okay, now let's discuss the implementation of the `_read()` method:

1. The method generates a random string of length equal to `size` using `chance`.
2. It pushes the string into the internal buffer. Note that since we are pushing strings, we also need to specify the encoding, `utf8` (this is not necessary if the chunk is simply a binary `Buffer`).
3. It terminates the stream randomly, with a likelihood of 5 percent, by pushing `null` into the internal buffer to indicate an EOF situation or, in other words, the end of the stream. This is just an implementation detail that we are adopting to force the stream to eventually terminate. Without this condition, our stream would be producing random data indefinitely.

Finite vs. Infinite Readable Streams



It's up to us to determine whether a Readable stream should terminate. You can signal the end of a stream by invoking `this.push(null)` in the `_read()` method. Some streams are naturally finite. For example, when reading from a file, the stream will end once all the bytes have been read because the file has a defined size. In other cases, we might create streams that provide data indefinitely. For instance, a readable stream could deliver continuous temperature readings from a sensor or a live video feed from a security camera. These streams will keep producing data for as long as the source remains active, and no communication errors occur.

Note that the `size` argument in the `_read()` function is an advisory parameter. It's good to honor it and push only the amount of data that was requested by the caller, even though it is not mandatory to do so.



When we invoke `push()`, we should check whether it returns `false`. When that happens, it means that the internal buffer of the receiving stream has reached the `highWaterMark` limit and we should stop adding more data to it. This is called **backpressure**, and we will be discussing it in more detail in the next section of this chapter. For now, just be aware that this implementation is not perfect because it does not handle backpressure.

That's it for `RandomStream`, we are now ready to use it. Let's see how to instantiate a `RandomStream` object and pull some data from it (using flowing mode):

```
// index.js
import { RandomStream } from './random-stream.js'

const randomStream = new RandomStream()
randomStream
  .on('data', chunk => {
    console.log(`Chunk received (${chunk.length} bytes): ${chunk.toString()}`)
  })
  .on('end', () => {
    console.log(`Produced ${randomStream.emittedBytes} bytes of random data`)
  })
```

Now, everything is ready for us to try our new custom stream. Simply execute the `index.js` module as usual and watch a random set of strings flow on the screen.

Simplified construction

For simple custom streams, we can avoid creating a custom class by using the *Readable* stream's *simplified construction* approach. With this approach, we only need to invoke `new Readable(options)` and pass a method named `read()` in the set of options. The `read()` method here has exactly the same semantic as the `_read()` method that we saw in the class extension approach. Let's rewrite `RandomStream` using the simplified constructor approach:

```
// simplified-construction.js
import { Readable } from 'node:stream'
import Chance from 'chance' // v1.1.12

const chance = new Chance()
let emittedBytes = 0

const randomStream = new Readable({
  read(size) {
    const chunk = chance.string({ length: size })
    this.push(chunk, 'utf8')
    emittedBytes += chunk.length
    if (chance.bool({ likelihood: 5 })) {
      this.push(null)
    }
  },
})

// now you can read data from the randomStream instance directly ...
```

This approach can be particularly useful when you don't need to manage a complicated state, and allows you to take advantage of a more succinct syntax. In the previous example, we created a single instance of our custom stream. If we want to adopt the simplified constructor approach, but we need to create multiple instances of the custom stream, we can wrap the initialization logic in a factory function that we can invoke multiple times to create those instances.



We will discuss the Factory pattern and other creational design patterns in *Chapter 7, Creational Design Patterns*.

Readable streams from iterables

You can easily create *Readable* stream instances from arrays or other *iterable* objects (that is, **generators**, **iterators**, and **async iterators**) using the `Readable.from()` helper.

In order to get accustomed to this helper, let's look at a simple example where we convert data from an array into a Readable stream:

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

const mountains = [
  { name: 'Everest', height: 8848 },
  { name: 'K2', height: 8611 },
  { name: 'Kangchenjunga', height: 8586 },
  { name: 'Lhotse', height: 8516 },
  { name: 'Makalu', height: 8481 }
]

const mountainsStream = Readable.from(mountains)
mountainsStream.on('data', (mountain) => {
  console.log(`${mountain.name.padStart(14)}\t${mountain.height}m`)
})
```

As we can see from this code, the `Readable.from()` method is quite simple to use: the first argument is an iterable instance (in our case, the `mountains` array). `Readable.from()` accepts an additional optional argument that can be used to specify stream options like `objectMode`.



Note that we didn't have to explicitly set `objectMode` to `true`. By default, `Readable.from()` will set `objectMode` to `true`, unless this is explicitly opted out by setting it to `false`. Stream options can be passed as a second argument to the function.

Running the previous code will produce the following output:

```
Everest      8848m
      K2      8611m
Kangchenjunga 8586m
      Lhotse   8516m
      Makalu   8481m
```



You should avoid instantiating large arrays in memory. Imagine if, in the previous example, we wanted to list all the mountains in the world. There are about 1 million mountains, so if we were to load all of them in an array upfront, we would allocate a quite significant amount of memory. Even if we then consume the data in the array through a `Readable` stream, all the data has already been preloaded, so we are effectively voiding the memory efficiency of streams. It's always preferable to load and consume the data in chunks, and you could do so by using native streams such as `fs.createReadStream`, by building a custom stream, or by using `Readable.from` with lazy iterables such as generators, iterators, or async iterators. We will see some examples of the latter approach in *Chapter 9, Behavioral Design Patterns*.

Writable streams

A Writable stream represents a data destination. Imagine, for instance, a file on the filesystem, a database table, a socket, the standard output, or the standard error interface. In Node.js, these kinds of abstractions can be implemented using the `Writable` abstract class, which is available in the `stream` module.

Writing to a stream

Pushing some data down a Writable stream is a straightforward business; all we have to do is use the `write()` method, which has the following signature:

```
writable.write(chunk, [encoding], [callback])
```

The `encoding` argument is optional and can be specified if `chunk` is a string (it defaults to `utf8`, and it's ignored if `chunk` is a buffer). The `callback` function, on the other hand, is called when the chunk is flushed into the underlying resource and is optional as well.

To signal that no more data will be written to the stream, we have to use the `end()` method:

```
writable.end([chunk], [encoding], [callback])
```

We can provide a final chunk of data through the `end()` method; in this case, the `callback` function is equivalent to registering a listener to the `finish` event, which is fired when all the data written in the stream has been flushed into the underlying resource.

Now, let's show how this works by creating a small HTTP server that outputs a random sequence of strings:

```
// entropy-server.js
import { createServer } from 'node:http'
import Chance from 'chance' // v1.1.12

const chance = new Chance()

const server = createServer((_req, res) => {
```



```
res.writeHead(200, { 'Content-Type': 'text/plain' }) // 1
do { // 2
  res.write(`${chance.string()}\n`)
} while (chance.bool({ likelihood: 95 }))
res.end('\n\n') // 3
res.on('finish', () => console.log('All data sent')) // 4
})

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

The HTTP server that we created writes into the `res` object, which is an instance of `http.ServerResponse` and also a `Writable` stream. What happens is explained as follows:

1. We first write the head of the HTTP response. Note that `writeHead()` is not a part of the `Writable` interface; in fact, it's an auxiliary method exposed by the `http.ServerResponse` class and is specific to the HTTP protocol. This method writes into the stream a properly formatted HTTP header, which will contain the status code 200 and a header specifying the content type of the response body that we are about to stream.
2. We start a loop that terminates with a likelihood of 5% (we instruct `chance.bool()` to return `true` 95% of the time). Inside the loop, we write a random string into the stream. Note that we use a `do ... while` loop here because we want to make sure to produce at least one random string.
3. Once we are out of the loop, we call `end()` on the stream, indicating that no more data will be written. Also, we provide a final string containing two new line characters to be written into the stream before ending it.
4. Finally, we register a listener for the `finish` event, which will be fired when all the data has been flushed into the underlying socket.

To test the server, we can open a browser at the address `http://localhost:3000` or use `curl` from the terminal as follows:

```
curl -i --raw localhost:3000
```

At this point, the server should start sending random strings to the HTTP client that you chose. If you are using a web browser, bear in mind that modern hardware can process things very quickly and that some browsers might buffer the data, so the streaming behavior might not be apparent.



By using the `-i --raw` flags in the `curl` command, we can have a peek at many details of the HTTP protocol. Specifically, `-i` includes response headers in the output. This allows us to see the exact data transferred in the header part of the response when we invoke `res.writeHead()`. The `node:http` module simplifies working with the HTTP protocol by automatically formatting response headers and applies sensible defaults such as adding standard headers like `Connection: keep-alive` and `Transfer-Encoding: chunked`. This last header is particularly interesting. It informs the client how to interpret the body of the incoming response. Chunked encoding is especially useful when sending large amounts of data whose total size isn't known until the request has been fully processed. This makes it a perfect fit for writable Node.js streams. With chunked encoding, the `Content-Length` header is omitted. Instead, each chunk begins with its length in hexadecimal format, followed by `\r\n`, the chunk's data, and another `\r\n`. The stream ends with a terminating chunk, which is identical to a regular chunk except that its length is zero. In our code, we don't need to handle these details manually. The `ServerResponse` writable stream provided by the `node:http` module takes care of encoding chunks correctly for us. We simply provide chunks by calling `write()` or `end()` on the response stream, and the stream handles the rest. This is one of the strengths of Node.js streams: they abstract away complex implementation details, making them easy to work with. If you want to learn more about chunked encoding, check out: nodejsdp.link/transfer-encoding. By using the `--raw` option, which disables all internal HTTP decoding of content or transfer encodings, we can see that these chunk terminators (`\r\n`) are present in the data received from the server.

Backpressure

Node.js data streams, like liquids in a piping system, can suffer from bottlenecks when data is written faster than the stream can handle. To manage this, incoming data is buffered in memory. However, without feedback from the stream to the writer, the buffer could keep growing, potentially leading to excessive memory usage.

Node.js streams are designed to maintain steady and predictable memory usage, even during large data transfers. Writable streams include a built-in signaling mechanism to alert the application when the internal buffer has accumulated too much data. This signal indicates that it's better to pause and wait for the buffered data to be flushed to the stream's destination before sending more data. The `writable.write()` method returns `false` once the buffer size exceeds the `highWaterMark` limit.

In Writable streams, the `highWaterMark` value sets the maximum buffer size in bytes. When this limit is exceeded, `write()` returning `false` signals the application to stop writing. Once the buffer is cleared, a `drain` event is emitted, indicating it's safe to resume writing. This process is known as **backpressure**.

Backpressure is an advisory mechanism. Even if `write()` returns `false`, we could ignore this signal and continue writing, making the buffer grow indefinitely. The stream won't be blocked automatically when the `highWaterMark` threshold is reached; therefore, it is recommended to always be mindful and respect the backpressure.



The mechanism described in this section is similarly applicable to Readable streams. In fact, backpressure exists in Readable streams too, and it's triggered when the `push()` method, which is invoked inside `_read()`, returns `false`. However, that's a problem specific to stream implementers, so we usually have to deal with it less frequently.

We can quickly demonstrate how to take into account the backpressure of a Writable stream by modifying the `entropy-server.js` module that we created previously:

```
// ...

const CHUNK_SIZE = 16 * 1024 - 1
const chance = new Chance()

const server = createServer((_req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' })

  let backPressureCount = 0
  let bytesSent = 0
  function generateMore() { // 1
    do {
      const randomChunk = chance.string({ length: CHUNK_SIZE }) // 2
      const shouldContinue = res.write(`${randomChunk}\n`) // 3
      bytesSent += CHUNK_SIZE
      if (!shouldContinue) { // 4
        console.warn(`back-pressure x${++backPressureCount}`)
        return res.once('drain', generateMore)
      }
    } while (chance.bool({ likelihood: 95 }))
    res.end('\n\n')
  }

  generateMore()

  res.on('finish', () => console.log(`Sent ${bytesSent} bytes`))
})

// ...
```

The most important steps of the previous code can be summarized as follows:

1. We wrapped the main data generation logic in a function called `generateMore()`.
2. To increase the chances of receiving some backpressure, we increased the size of the data chunk to 16 KB minus 1 byte, which is very close to the default `highWaterMark` limit.
3. After writing a chunk of data, we check the return value of `res.write()`. If we receive `false`, it means that the internal buffer is full, and we should stop sending more data.
4. When this happens, we exit the function and register another cycle of writes using `generateMore()` for when the `drain` event is emitted.

If we now try to run the server again, and then generate a client request with `curl` (or with a browser), there is a high probability that there will be some backpressure, as the server produces data at a very high rate, faster than the underlying socket can handle. This example also prints how many backpressure events happen and how much data is being transferred for every request. You are encouraged to try different requests, check the logs, and try to make sense of what's happening under the hood.

Implementing Writable streams

We can implement a new `Writable` stream by inheriting the class `Writable` and providing an implementation for the `_write()` method. Let's try to do it immediately while discussing the details along the way.

Let's build a `Writable` stream that receives objects in the following format:

```
{
  path: <path to a file>
  content: <string or buffer>
}
```

For each one of these objects, our stream will save the `content` property into a file created at the given `path`. We can immediately see that the inputs of our stream are objects, and not strings or buffers. This means that our stream must work in object mode, which gives us a great excuse to also explore object mode with this example:

```
// to-file-stream.js
import { Writable } from 'node:stream'
import { promises as fs } from 'node:fs'
import { dirname } from 'node:path'
import { mkdirp } from 'mkdirp' // v3.0.1

export class ToFileStream extends Writable {
  constructor(options) {
    super({ ...options, objectMode: true })
  }
}
```

```
  _write(chunk, _encoding, cb) {
    mkdirp(dirname(chunk.path))
      .then(() => fs.writeFile(chunk.path, chunk.content))
      .then(() => cb())
      .catch(cb)
  }
}
```

We created a new class for our new stream, which extends `Writable` from the `stream` module.

We had to invoke the parent constructor to initialize its internal state; we also needed to make sure that the `options` object specifies that the stream works in object mode (`objectMode: true`). Other options accepted by `Writable` are as follows:

- `highWaterMark` (the default is 16 KB): This controls the backpressure limit.
- `decodeStrings` (defaults to `true`): This enables the automatic decoding of strings into binary buffers before passing them to the `_write()` method. This option is ignored in object mode.

Finally, we provided an implementation for the `_write()` method. As you can see, the method accepts a data chunk and an encoding (which makes sense only if we are in binary mode and the stream option `decodeStrings` is set to `false`). Also, the method accepts a callback function (`cb`), which needs to be invoked when the operation completes; it's not necessary to pass the result of the operation but, if needed, we can still pass an error that will cause the stream to emit an error event.

Now, to try the stream that we just built, we can create a new module and perform some write operations against the stream:

```
// index.js
import { join } from 'node:path'
import { ToFileStream } from './to-file-stream.js'
const tfs = new ToFileStream()

const outDir = join(import.meta.dirname, 'files')

tfs.write({ path: join(outDir, 'file1.txt'), content: 'Hello' })
tfs.write({ path: join(outDir, 'file2.txt'), content: 'Node.js' })
tfs.write({ path: join(outDir, 'file3.txt'), content: 'streams' })
tfs.end(() => console.log('All files created'))
```

Here, we created and used our first custom `Writable` stream. Run the new module as usual and check its output. You will see that after the execution, three new files will be created within a new folder called `files`.

Simplified construction

As we saw for Readable streams, Writable streams also offer a simplified construction mechanism. If we were to rewrite `ToFileStream` using the simplified construction for Writable streams, it would look like this:

```
// ...
const tfs = new Writable({
  objectMode: true,
  write(chunk, _encoding, cb) {
    mkdirp(dirname(chunk.path))
    .then(() => fs.writeFile(chunk.path, chunk.content))
    .then(() => cb())
    .catch(cb)
  },
})
// ...
```

With this approach, we are simply using the `Writable` constructor and passing a `write()` function that implements the custom logic of our `Writable` instance. Note that with this approach, the `write()` function doesn't have an underscore in the name. We can also pass other construction options like `objectMode`.

Duplex streams

A Duplex stream is a stream that is both Readable and Writable. It is useful when we want to describe an entity that is both a data source and a data destination, such as network sockets, for example. Duplex streams inherit the methods of both `stream.Readable` and `stream.Writable`, so this is nothing new to us. This means that we can `read()` or `write()` data, or listen for both readable and drain events.

To create a custom Duplex stream, we have to provide an implementation for both `_read()` and `_write()`. The options object passed to the `Duplex()` constructor is internally forwarded to both the Readable and Writable constructors. The options are the same as those we already discussed in the previous sections, with the addition of a new one called `allowHalfOpen` (defaults to `true`) that, if set to `false`, will cause both parts (Readable and Writable) of the stream to end if only one of them does.



If we need to have a Duplex stream working in object mode on one side and binary mode on the other, we can use the options `readableObjectMode` and `writableObjectMode` independently.

Transform streams

Transform streams are a special kind of Duplex stream that are specifically designed to handle data transformations. Just to give you a few concrete examples, the functions `zlib.createGzip()` and `crypto.createCipheriv()` that we discussed at the beginning of this chapter create Transform streams for compression and encryption, respectively.

In a simple Duplex stream, there is no immediate relationship between the data read from the stream and the data written into it (at least, the stream is agnostic to such a relationship). Think about a TCP socket, which just sends and receives data to and from the remote peer; the socket is not aware of any relationship between the input and output. *Figure 6.4* illustrates the data flow in a Duplex stream:

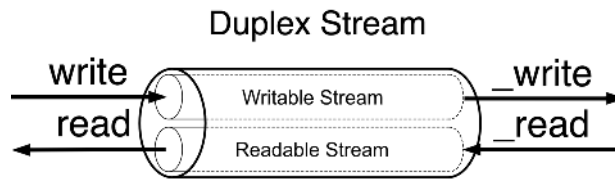


Figure 6.4: Duplex stream schematic representation

On the other hand, Transform streams apply some kind of transformation to each chunk of data that they receive from their Writable side, and then make the transformed data available on their Readable side. *Figure 6.5* shows how the data flows in a Transform stream:



Figure 6.5: Transform stream schematic representation

Returning to our compression example, a transform stream can be visualized as follows: when uncompressed data is written to the stream, its internal implementation compresses the data and stores it in an internal buffer. When the data is read from the other end, the compressed version of the data is retrieved. This is how transformation happens on the fly: data comes in, gets transformed, and then goes out.

From a user perspective, the programmatic interface of a Transform stream is exactly like that of a Duplex stream. However, when we want to implement a new Duplex stream, we have to provide both the `_read()` and `_write()` methods, while for implementing a new Transform stream, we have to fill in another pair of methods: `_transform()` and `_flush()`.

Let's see how to create a new Transform stream with an example.

Implementing Transform streams

Let's implement a Transform stream that replaces all the occurrences of a given string:

```
// replaceStream.js
import { Transform } from 'node:stream'

export class ReplaceStream extends Transform {
  constructor(searchStr, replaceStr, options) {
    super({ ...options })
    this.searchStr = searchStr
    this.replaceStr = replaceStr
    this.tail = ''
  }

  _transform(chunk, _encoding, cb) {
    const pieces = (this.tail + chunk).split(this.searchStr) // 1
    const lastPiece = pieces[pieces.length - 1] // 2
    const tailLen = this.searchStr.length - 1
    this.tail = lastPiece.slice(-tailLen)
    pieces[pieces.length - 1] = lastPiece.slice(0, -tailLen)

    this.push(pieces.join(this.replaceStr)) // 3
    cb()
  }

  _flush(cb) {
    this.push(this.tail)
    cb()
  }
}
```

In this example, we created a new class extending the Transform base class. The constructor of the class accepts three arguments: `searchStr`, `replaceStr`, and `options`. As you can imagine, they allow us to define the text to match and the string to use as a replacement, plus an `options` object for advanced configuration of the underlying Transform stream. We also initialize an internal `tail` variable, which will be used later by the `_transform()` method.

Now, let's analyze the `_transform()` method, which is the core of our new class. The `_transform()` method has practically the same signature as the `_write()` method of the Writable stream, but instead of writing data into an underlying resource, it pushes it into the internal read buffer using `this.push()`, exactly as we would do in the `_read()` method of a Readable stream. This shows how the two sides of a Transform stream are connected.

The `_transform()` method of `ReplaceStream` implements the core of our algorithm. To search for and replace a string in a buffer is an easy task; however, it's a totally different story when the data is streaming, and possible matches might be distributed across multiple chunks. The procedure followed by the code can be explained as follows:

1. Our algorithm splits the data in memory (`tail` data and the current chunk) using `searchStr` as a separator.
2. Then, it takes the last item of the array generated by the operation and extracts the last `searchString.length - 1` characters. The result is saved in the `tail` variable and will be prepended to the next chunk of data.
3. Finally, all the pieces resulting from `split()` are joined together using `replaceStr` as a separator and pushed into the internal buffer.

When the stream ends, we might still have some content in the `tail` variable not pushed into the internal buffer. That's exactly what the `_flush()` method is for; it is invoked just before the stream is ended, and this is where we have one final chance to finalize the stream or push any remaining data before completely ending the stream.

The `_flush()` method only takes in a callback, which we have to make sure to invoke when all the operations are complete, causing the stream to be terminated. With this, we have completed our `ReplaceStream` class.

Why is the `tail` variable necessary?

Streams process data in chunks, and these chunks don't always align with the boundaries of the target search string. For example, if the string we are trying to match is split across two chunks, the `split()` operation on a chunk alone won't detect it, potentially leaving part of the match unnoticed. The `tail` variable ensures that the last portion of a chunk—potentially part of a match—is preserved and concatenated with the next chunk. This way, the stream can properly handle matches that span chunk boundaries, avoiding incorrect replacements or missing matches entirely.



In `Transform` streams, it's not uncommon for the logic to involve buffering data from multiple chunks before there's enough information to perform the transformation. For example, cryptography often works on fixed-size blocks of data. If a chunk doesn't provide enough data to form a complete block, the `Transform` stream accumulates multiple chunks until it has enough to process the transformation. This buffering behavior ensures transformations are accurate and consistent, even when input data arrives in unpredictable sizes.

This should also clarify why the `_flush()` method exists. It's provided to handle any remaining data still buffered in the `Transform` stream when the writer has finished writing. This ensures that leftover data—such as the `tail` in this example—is processed and emitted, preventing incomplete or lost output.

Now, it's time to try the new stream. Let's create a script that writes some data into the stream and then reads the transformed result:

```
// index.js
import { ReplaceStream } from './replace-stream.js'

const replaceStream = new ReplaceStream('World', 'Node.js')
replaceStream.on('data', chunk => process.stdout.write(chunk.toString()))

replaceStream.write('Hello W')
replaceStream.write('orld!')
replaceStream.end('\n')
```

To make life a little bit harder for our stream, we spread the search term (which is `World`) across two different chunks, then, using flowing mode, we read from the same stream, logging each transformed chunk. Running the preceding program should produce the following output:

```
Hello Node.js!
```

Simplified construction

Unsurprisingly, even Transform streams support simplified construction. At this point, we should have developed an instinct for how this API might look, so let's get our hands dirty straight away and rewrite the previous example using this approach:

```
// simplified-construction.js
// ...
const searchStr = 'World'
const replaceStr = 'Node.js'
let tail = ''

const replaceStream = new Transform({
  defaultEncoding: 'utf8',

  transform(chunk, _encoding, cb) {
    const pieces = (tail + chunk).split(searchStr)
    const lastPiece = pieces[pieces.length - 1]
    const tailLen = searchStr.length - 1
    tail = lastPiece.slice(-tailLen)
    pieces[pieces.length - 1] = lastPiece.slice(0, -tailLen)
    this.push(pieces.join(replaceStr))
    cb()
  },
})
```

```
flush(cb) {  
  this.push(tail)  
  cb()  
},  
})  
  
// now write to replaceStream ...
```

As expected, simplified construction works by directly instantiating a new Transform object and passing our specific transformation logic through the transform() and flush() functions directly through the options object. Note that transform() and flush() don't have a prepended underscore here.

Filtering and aggregating data with Transform streams

As we discussed earlier, Transform streams are a great tool for building data transformation pipelines. In a previous example, we showed how to use a Transform stream to replace words in a text stream. We also mentioned other use cases, like compression and encryption. But Transform streams aren't limited to those examples. They're often used for tasks like filtering and aggregating data.

To make this more concrete, imagine a Fortune 500 company asks us to analyze a large file containing all their sales data for 2024. The file, data.csv, is a sales report in CSV format, and they want us to calculate the total profit for sales made in Italy. Sure, we could use a spreadsheet application to do this, but where's the fun in that?

Instead, let's use Node.js streams. Streams are well-suited for this kind of task because they can process large datasets incrementally, without loading everything into memory. This makes them efficient and scalable. Plus, building a solution with streams sets the stage for automation; perfect if you need to generate similar reports regularly or process other large datasets in the future.

Let's imagine the sales data that is stored in the CSV file contains three fields per line: item type, country of sale, and profit. So, such a file could look like this:

```
type, country, profit  
Household, Namibia, 597290.92  
Baby Food, Iceland, 808579.10  
Meat, Russia, 277305.60  
Meat, Italy, 413270.00  
Cereal, Malta, 174965.25  
Meat, Indonesia, 145402.40  
Household, Italy, 728880.54  
[... many more lines]
```

Now, it's clear that we must find all the records that have "Italy" in the country field and, in the process, sum up the profit value of the matching lines into a single number.

In order to process a CSV file in a streaming fashion, we can use the excellent third-party module `csv-parse` (nodejsdp.link/csv-parse).

If we assume for a moment that we have already implemented our custom streams to filter and aggregate the data, a possible solution to this task might look like this:

```
// index.js
import { createReadStream } from 'node:fs'
import { Parser } from 'csv-parse' // v5.6.0
import { FilterByCountry } from './filter-by-country.js'
import { SumProfit } from './sum-profit.js'

const csvParser = new Parser({ columns: true })

createReadStream('data.csv.gz') // 1
  .pipe(csvParser) // 2
  .pipe(new FilterByCountry('Italy')) // 3
  .pipe(new SumProfit()) // 4
  .pipe(process.stdout) // 5
```

The streaming pipeline proposed here consists of five steps:

1. We read the source CSV file as a stream.
2. We use the `csv-parse` library to parse every line of the document as a CSV record. For every line, this stream will emit an object containing the properties `type`, `country`, and `profit`. With the option `columns: true`, the library will read the names of the available columns from the first row of the CSV file.
3. We filter all the records by country, retaining only the records that match the country “Italy.” All the records that don’t match “Italy” are dropped, which means that they will not be passed to the other steps in the pipeline. Note that this is one of the custom `Transform` streams that we have to implement.
4. For every record, we accumulate the profit. This stream will eventually emit a single string, which represents the value of the total profit for products sold in Italy. This value will be emitted by the stream only when all the data from the original file has been completely processed. Note that this is the second custom `Transform` stream that we have to implement to complete this project.
5. Finally, the data emitted from the previous step is displayed in the standard output.

Now, let’s implement the `FilterByCountry` stream:

```
// filter-by-country.js
import { Transform } from 'node:stream'

export class FilterByCountry extends Transform {
  constructor(country, options = {}) {
    options.objectMode = true
    super(options)
```

```
    this.country = country
  }

  _transform(record, _enc, cb) {
    if (record.country === this.country) {
      this.push(record)
    }
    cb()
  }
}
```

`FilterByCountry` is a custom Transform stream. We can see that the constructor accepts an argument called `country`, which allows us to specify the country name we want to match on. In the constructor, we also set the stream to run in `objectMode` because we know it will be used to process objects (records coming from the CSV file).

In the `_transform` method, we check if the country of the current record matches the country specified at construction time. If it's a match, then we pass the record on to the next stage of the pipeline by calling `this.push()`. Regardless of whether the record matches or not, we need to invoke `cb()` to indicate that the current record has been successfully processed and that the stream is ready to receive another record.



Pattern: Transform filter

Invoke `this.push()` in a conditional way to allow only some data to reach the next stage of the pipeline.

Finally, let's implement the `SumProfit` filter:

```
// sum-profit.js
import { Transform } from 'node:stream'

export class SumProfit extends Transform {
  constructor(options = {}) {
    options.objectMode = true
    super(options)
    this.total = 0
  }

  _transform(record, _enc, cb) {
    this.total += Number.parseFloat(record.profit)
    cb()
  }
}
```

```
  _flush(cb) {
    this.push(this.total.toString())
    cb()
  }
}
```

This stream needs to run in `objectMode` as well, because it will receive objects representing records from the CSV file. Note that, in the constructor, we also initialize an instance variable called `total` and we set its value to `0`.

In the `_transform()` method, we process every record and use the current `profit` value to increase the `total`. It's important to note that this time, we are not calling `this.push()`. This means that no value is emitted while the data is flowing through the stream. We still need to call `cb()`, though, to indicate that the current record has been processed and the stream is ready to receive another one.

In order to emit the final result when all the data has been processed, we have to define a custom flush behavior using the `_flush()` method. Here, we finally call `this.push()` to emit a string representation of the resulting `total` value. Remember that `_flush()` is automatically invoked before the stream is closed.



Pattern: Streaming aggregation

Use `_transform()` to process the data and accumulate the partial result, then call `this.push()` only in the `_flush()` method to emit the result when all the data has been processed.

This completes our example. Now, you can grab the CSV file from the code repository and execute this program to see what the total profit for Italy is. No surprise it's going to be a lot of money since we are talking about the profit of a Fortune 500 company!



You could combine filtering and aggregation into a single `Transform` stream. While this approach might be less reusable, it can offer a slight performance boost since less data gets passed between steps in the stream pipeline. If you're up for the challenge, try implementing this as an exercise!



The Node.js streams library includes a set of `Readable` stream helper methods (experimental at the time of writing). Among these are `Readable.map()` and `Readable.reduce()`, which could solve the problem we just explored in a more concise and streamlined way. We'll dive into `Readable` stream helpers later in this chapter.

PassThrough streams

There is a fifth type of stream that is worth mentioning: `PassThrough`. This type of stream is a special type of `Transform` stream that outputs every data chunk without applying any transformation.

`PassThrough` is possibly the most underrated type of stream, but there are several circumstances in which it can be a very valuable tool in our toolbox. For instance, `PassThrough` streams can be useful for observability or to implement late piping and lazy stream patterns.

Observability

If we want to observe how much data is flowing through one or more streams, we could do so by attaching a data event listener to a `PassThrough` instance and then piping this instance at a given point in a stream pipeline. Let's see a simplified example to be able to appreciate this concept:

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

let bytesWritten = 0
const monitor = new PassThrough()
monitor.on('data', chunk => {
  bytesWritten += chunk.length
})
monitor.on('finish', () => {
  console.log(`${bytesWritten} bytes written`)
})

monitor.write('Hello!')
monitor.end()
```

In this example, we are creating a new instance of `PassThrough` and using the `data` event to count how many bytes are flowing through the stream. We also use the `finish` event to dump the total amount to the console. Finally, we write some data directly into the stream using `write()` and `end()`. This is just an illustrative example; in a more realistic scenario, we would be piping our `monitor` instance at a given point in a stream pipeline. For instance, if we wanted to monitor how many bytes are written to disk in our first file compression example of this chapter, we could easily achieve that by doing something like this:

```
createReadStream(filename)
  .pipe(createGzip())
  .pipe(monitor)
  .pipe(createWriteStream(`${filename}.gz`))
```

The beauty of this approach is that we didn't have to touch any of the other existing streams in the pipeline, so if we need to observe other parts of the pipeline (for instance, imagine we want to know the number of bytes of the uncompressed data), we can move `monitor` around with very little effort. We could even have multiple `PassThrough` streams to monitor different parts of a pipeline at the same time.



Note that you could implement an alternative version of the `monitor` stream by using a custom transform stream instead. In such a case, you would have to make sure that the received chunks are pushed without any modification or delay, which is something that a `PassThrough` stream would do automatically for you. Both approaches are equally valid, so pick the approach that feels more natural to you.

Late piping

In some circumstances, we might have to use APIs that accept a stream as an input parameter. This is generally not a big deal because we already know how to create and use streams. However, it may get a little bit more complicated if the data we want to read or write through the stream is only available after we've called the given API.

To visualize this scenario in more concrete terms, let's imagine that we have to use an API that gives us the following function to upload a file to a data storage service:

```
function upload (filename, contentStream) {  
  // ...  
}
```



This function is effectively a simplified version of what is commonly available in the SDK of file storage services like Amazon Simple Storage Service (S3) or Azure Blob Storage service. Often, those libraries will provide the user with a more flexible function that can receive the content data in different formats (for example, a string, a buffer, or a `Readable` stream).

Now, if we want to upload a file from the filesystem, this is a trivial operation, and we can do something like this:

```
import { createReadStream } from 'fs'  
upload('a-picture.jpg', createReadStream('/path/to/a-picture.jpg'))
```

But what if we want to do some processing on the file stream before the upload? For instance, let's say we want to compress or encrypt the data. Also, what if we need to perform this transformation asynchronously after the upload function has been called?

In such cases, we can provide a `PassThrough` stream to the `upload()` function, which will effectively act as a placeholder. The internal implementation of `upload()` will immediately try to consume data from it, but there will be no data available in the stream until we actually write to it. Also, the stream won't be considered complete until we close it, so the `upload()` function will have to wait for data to flow through the `PassThrough` instance to initiate the upload.

Let's see a possible command-line script that uses this approach to upload a file from the filesystem and compresses it using **Brotli** compression. We are going to assume that the third-party `upload()` function is provided in a file called `upload.js`.

```
// upload-cli.js
import { createReadStream } from 'node:fs'
import { createBrotliCompress } from 'node:zlib'
import { PassThrough } from 'node:stream'
import { basename } from 'node:path'
import { upload } from './upload.js'

const filepath = process.argv[2] // 1
const filename = basename(filepath)
const contentStream = new PassThrough() // 2

upload(`${filename}.br`, contentStream) // 3
  .then(response => {
    console.log(`Server response: ${response.data}`)
  })
  .catch(err => {
    console.error(err)
    process.exit(1)
  })

createReadStream(filepath) // 4
  .pipe(createBrotliCompress())
  .pipe(contentStream)
```



In this book's repository, you will find a complete implementation of this example that allows you to upload files to an HTTP server that you can run locally.

Let's review what's happening in the previous example:

1. We get the path to the file we want to upload from the first command-line argument and use `basename` to extrapolate the filename from the given path.
2. We create a placeholder for our content stream as a `PassThrough` instance.
3. Now, we invoke the `upload` function by passing our filename (with the added `.br` suffix, indicating that it is using Brotli compression) and the placeholder content stream.
4. Finally, we create a pipeline by chaining a filesystem `Readable` stream, a Brotli compression `Transform` stream, and finally our content stream as the destination.

When this code is executed, the upload will start as soon as we invoke the `upload()` function (possibly establishing a connection to the remote server), but the data will start to flow only later, when our pipeline is initialized. Note that our pipeline will also close the `contentStream` when the processing completes, which will indicate to the `upload()` function that all the content has been fully consumed.



Pattern

Use a `PassThrough` stream when you need to provide a placeholder for data that will be read or written in the future.

We can also use this pattern to transform the interface of the `upload()` function. Instead of accepting a `Readable` stream as input, we can make it return a `Writable` stream, which can then be used to provide the data we want to upload:

```
function createUploadStream (filename) {  
  // ...  
  // returns a writable stream that can be used to upload data  
}
```

If we were tasked to implement this function, we could achieve that in a very elegant way by using a `PassThrough` instance, as in the following example implementation:

```
function createUploadStream (filename) {  
  const connector = new PassThrough()  
  upload(filename, connector)  
  return connector  
}
```

In the preceding code, we are using a `PassThrough` stream as a connector. This stream becomes a perfect abstraction for a case where the consumer of the library can write data at any later stage.

The `createUploadStream()` function can then be used as follows:

```
const upload = createUploadStream('a-file.txt')  
upload.write('Hello World')  
upload.end()
```



This book's repository also contains an HTTP upload example that adopts this alternative pattern.

Lazy streams

Sometimes, we need to create a large number of streams at the same time, for example, to pass them to a function for further processing. A typical example is when using archiver (nodejsdp.link/archiver), a package for creating archives such as TAR and ZIP. The archiver package allows you to create an archive from a set of streams, representing the files to add. The problem is that if we want to pass a large number of streams, such as from files on the filesystem, we would likely get an `EMFILE`, too many open files error. This is because functions like `createReadStream()` from the `fs` module will actually open a file descriptor every time a new stream is created, even before you start to read from those streams.

In more generic terms, creating a stream instance might initialize expensive operations straight away (for example, open a file or a socket, initialize a connection to a database, and so on), even before we start to use such a stream. This might not be desirable if you are creating a large number of stream instances for later consumption.

In these cases, you might want to delay the expensive initialization until you need to consume data from the stream.

It is possible to achieve this by using a library like `lazystream` (nodejsdp.link/lazystream). This library allows you to create proxies for actual stream instances, where the creation of the stream instance is deferred until some piece of code starts to consume data from the proxy.

In the following example, `lazystream` allows us to create a lazy `Readable` stream for the special Unix file `/dev/urandom`:

```
import lazystream from 'lazystream'
const lazyURandom = new lazystream.Readable(function (options) {
  return fs.createReadStream('/dev/urandom')
})
```

The function we pass as a parameter to `new lazystream.Readable()` is effectively a factory function that generates the proxied stream when necessary.

Behind the scenes, `lazystream` is implemented using a `PassThrough` stream that, only when its `_read()` method is invoked for the first time, creates the proxied instance by invoking the factory function, and pipes the generated stream into the `PassThrough` itself. The code that consumes the stream is totally agnostic of the proxying that is happening here, and it will consume the data as if it was flowing directly from the `PassThrough` stream. `lazystream` implements a similar utility to create a lazy `Writable` stream as well.

Creating lazy `Readable` and `Writable` streams from scratch could be an interesting exercise that is left to you. If you get stuck, have a look at the source code of `lazystream` for inspiration on how to implement this pattern.

In the next section, we will discuss the `.pipe()` method in greater detail and also see other ways to connect different streams to form a processing pipeline.

Connecting streams using pipes

The concept of Unix pipes was invented by Douglas McIlroy. This enabled the output of a program to be connected to the input of the next. Take a look at the following command:

```
echo Hello World! | sed s/World/Node.js/g
```

In the preceding command, `echo` will write `Hello World!` to its standard output, which is then redirected to the standard input of the `sed` command (thanks to the pipe `|` operator). Then, `sed` replaces any occurrence of `World` with `Node.js` and prints the result to its standard output (which, this time, is the console).

In a similar way, Node.js streams can be connected using the `pipe()` method of the `Readable` stream object, which has the following interface:

```
readable.pipe(writable, [options])
```

We have already used the `pipe()` method in a few examples, but let's finally dive into what it does for us under the hood.

Very intuitively, the `pipe()` method takes the data that is emitted from the readable stream and pumps it into the provided writable stream. Also, the writable stream is ended automatically when the readable stream emits an end event (unless we specify `{end: false}` as options). The `pipe()` method returns the writable stream passed in the first argument, allowing us to create chained invocations if such a stream is also `Readable` (such as a `Duplex` or `Transform` stream).

Piping two streams together will create *suction*, which allows the data to flow automatically to the writable stream, so there is no need to call `read()` or `write()`, but most importantly, there is no need to control the backpressure anymore, because it's automatically taken care of.

To provide a quick example, we can create a new module that takes a text stream from the standard input, applies the *replace* transformation discussed earlier when we built our custom `ReplaceStream`, and then pushes the data back to the standard output:

```
// replace.js
import { ReplaceStream } from './replace-stream.js'

process.stdin
  .pipe(new ReplaceStream(process.argv[2], process.argv[3]))
  .pipe(process.stdout)
```

The preceding program pipes the data that comes from the standard input into an instance of `ReplaceStream` and then back to the standard output. Now, to try this small application, we can leverage a Unix pipe to redirect some data into its standard input, as shown in the following example:

```
echo Hello World! | node replace.js World Node.js
```

This should produce the following output:

```
Hello Node.js!
```

This simple example demonstrates that streams (and in particular, text streams) are a universal interface and that pipes are the way to compose and interconnect all these interfaces almost magically.

Pipes and error handling

The `pipe()` method is very powerful, but there's one important problem: error events are not propagated automatically through the pipeline when using `pipe()`. Take, for example, this code fragment:

```
stream1
  .pipe(stream2)
  .on('error', () => {})
```

In the preceding pipeline, we will catch only the errors coming from `stream2`, which is the stream that we attached the listener to. This means that, if we want to catch any error generated from `stream1`, we have to attach another error listener directly to it, which will make our example look like this:

```
stream1
  .on('error', () => {})
  .pipe(stream2)
  .on('error', () => {})
```

This is clearly not ideal, especially when dealing with pipelines with a significant number of steps. To make this matter worse, in the event of an error, the failing stream is only *unpiped* from the pipeline. The failing stream is not properly destroyed, which might leave dangling resources (for example, file descriptors, connections, and so on) and leak memory. A more robust (yet inelegant) implementation of the preceding snippet might look like this:

```
function handleError (err) {
  console.error(err)
  stream1.destroy()
  stream2.destroy()
}

stream1
  .on('error', handleError)
  .pipe(stream2)
  .on('error', handleError)
```

In this example, we registered a handler for the error event for both `stream1` and `stream2`. When an error happens, our `handleError()` function is invoked, and we can log the error and destroy every stream in the pipeline. This allows us to ensure that all the allocated resources are properly released, and the error is handled gracefully.

Better error handling with pipeline()

Handling errors manually in a pipeline is not just cumbersome, but also error-prone—something we should avoid if we can!

Luckily, the core `node:stream` package offers us an excellent utility function that can make building pipelines a much safer and more enjoyable practice, which is the `pipeline()` helper function.

In a nutshell, you can use the `pipeline()` function as follows:

```
pipeline(stream1, stream2, stream3, ... , cb)
```

The last argument is an optional callback that will be called when the stream finishes. If it finishes because of an error, the callback will be invoked with the given error as the first argument.

If you prefer to avoid callbacks and rather use a Promise, there's a Promise-based alternative in the `node:stream/promises` package:

```
pipeline(stream1, stream2, stream3, ...) // returns a promise
```

This alternative returns a Promise that will resolve when the pipeline completes or rejects in case of an error.

Both of these helpers pipe every stream passed in the arguments list to the next one. For each stream, they will also register a proper error and `close` listeners. This way, all the streams are properly destroyed when the pipeline completes successfully or when it's interrupted by an error.

To get some practice with these helpers, let's write a simple command-line script that implements the following pipeline:

- Reads a Gzip data stream from the standard input
- Decompresses the data
- Makes all the text uppercase
- Gzips the resulting data
- Sends the data back to the standard output

```
// uppercasify-gzipped.js
import { createGzip, createGunzip } from 'node:zlib' // 1
import { Transform } from 'node:stream'
import { pipeline } from 'node:stream/promises'

const uppercasify = new Transform({ // 2
  transform(chunk, _enc, cb) {
    this.push(chunk.toString().toUpperCase())
    cb()
  },
})
```

```
await pipeline( // 3
  process.stdin,
  createGunzip(),
  uppercasify,
  createGzip(),
  process.stdout
)
```

In this example:

1. We are importing the necessary dependencies from `zlib`, `stream`, and the `stream/promises` modules.
2. We create a simple `Transform` stream that makes every chunk uppercase.
3. We define our pipeline, where we list all the stream instances in order. Note that we use `await` to wait for the pipeline to complete. In this example, this is not mandatory because we don't do anything after the pipeline is completed, but it's a good practice to have this since we might decide to evolve our script in the future, or we might want to add a `try catch` around this expression to handle potential errors.

The pipeline will start automatically by consuming data from the standard input and producing data for the standard output.

We could test our script with the following command:

```
echo 'Hello World!' | gzip | node uppercasify-gzipped.js | gunzip
```

This should produce the following output:

```
HELLO WORLD!
```

If we try to remove the `gzip` step from the preceding sequence of commands, our script will fail with an uncaught error. This error is raised by the stream created with the `createGunzip()` function, which is responsible for decompressing the data. If the data is not actually gzipped, the decompression algorithm won't be able to process the data and it will fail. In such a case, `pipeline()` will take care of cleaning up after the error and destroy all the streams in the pipeline.

Now that we have built a solid understanding of Node.js streams, we are ready to move into some more involved stream patterns like control flow and advanced piping patterns.

Asynchronous control flow patterns with streams

Going through the examples that we have presented so far, it should be clear that streams can be useful not only to handle I/O, but also as an elegant programming pattern that can be used to process any kind of data. But the advantages do not end at its simple appearance; streams can also be leveraged to turn “asynchronous control flow” into “**flow control**,” as we will see in this section.

Sequential execution

By default, streams will handle data in sequence. For example, the `_transform()` function of a Transform stream will never be invoked with the next chunk of data until the previous invocation completes by calling `callback()`. This is an important property of streams, crucial for processing each chunk in the right order, but it can also be exploited to turn streams into an elegant alternative to the traditional control flow patterns.

Let's look at some code to clarify what we mean. We will be working on an example to demonstrate how we can use streams to execute asynchronous tasks in sequence. Let's create a function that concatenates a set of files received as input, making sure to honor the order in which they are provided. Let's create a new module called `concat-files.js` and define its contents as follows:

```
import { createReadStream, createWriteStream } from 'node:fs'
import { Readable, Transform } from 'node:stream'

export function concatFiles(dest, files) {
  return new Promise((resolve, reject) => {
    const destStream = createWriteStream(dest)
    Readable.from(files) // 1
      .pipe(
        new Transform({ // 2
          objectMode: true,
          transform(filename, _enc, done) {
            const src = createReadStream(filename)
            src.pipe(destStream, { end: false })
            // same as ((err) => done(err))
            // propagates the error
            src.on('error', done)
            // same as (() => done())
            // propagates correct completion
            src.on('end', done) // 3
          },
        })
      )
      .on('error', err => {
        destStream.end()
        reject(err)
      })
      .on('finish', () => { // 4
        destStream.end()
        resolve()
      })
  })
}
```



```
  })  
}
```

The preceding function implements a sequential iteration over the `files` array by transforming it into a stream. The algorithm can be explained as follows:

1. First, we use `Readable.from()` to create a `Readable` stream from the `files` array. This stream operates in object mode (the default setting for streams created with `Readable.from()`) and it will emit filenames: every chunk is a string indicating the path to a file. The order of the chunks respects the order of the files in the `files` array.
2. Next, we create a custom `Transform` stream to handle each file in the sequence. Since we are receiving strings, we set the option `objectMode` to `true`. In our transformation logic, for each file we create a `Readable` stream to read the file content and pipe it into `destStream` (a `Writable` stream for the destination file). We make sure not to close `destStream` after the source file finishes reading by specifying `{ end: false }` in the `pipe()` options.
3. When all the contents of the source file have been piped into `destStream`, we invoke the `done` function to communicate the completion of the current processing, which is necessary to trigger the processing of the next file.
4. When all the files have been processed, the `finish` event is fired; we can finally end `destStream` and invoke the `cb()` function of `concatFiles()`, which signals the completion of the whole operation.

We can now try to use the little module we just created:

```
// concat.js  
import { concatFiles } from './concat-files.js'  
  
try {  
  await concatFiles(process.argv[2], process.argv.slice(3))  
} catch (err) {  
  console.error(err)  
  process.exit(1)  
}  
  
console.log('All files concatenated successfully')
```

We can now run the preceding program by passing the destination file as the first command-line argument, followed by a list of files to concatenate; for example:

```
node concat.js all-together.txt file1.txt file2.txt
```

This should create a new file called `all-together.txt` containing, in order, the contents of `file1.txt` and `file2.txt`.

With the `concatFiles()` function, we were able to obtain an asynchronous sequential iteration using only streams. This is an elegant and compact solution that enriches our toolbelt, along with the techniques we already explored in *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*, and *Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await*.



Pattern

Use a stream, or combination of streams, to easily iterate over a set of asynchronous tasks in sequence.

In the next section, we will discover how to use Node.js streams to implement unordered concurrent task execution.

Unordered concurrent execution

We just saw that streams process data chunks in sequence, but sometimes, this can be a bottleneck as we would not make the most of the concurrency of Node.js. If we have to execute a slow asynchronous operation for every data chunk, it can be advantageous to make the execution concurrent and speed up the overall process. Of course, this pattern can only be applied if there is no relationship between each chunk of data, which might happen frequently for object streams, but very rarely for binary streams.



Caution

Unordered concurrent streams cannot be used when the order in which the data is processed is important.

To make the execution of a `Transform` stream concurrent, we can apply the same patterns that we learned about in *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*, but with some adaptations to get them working with streams. Let's see how this works.

Implementing an unordered concurrent stream

Let's immediately demonstrate how to implement an unordered concurrent stream with an example. Let's create a module called `concurrent-stream.js` and define a generic `Transform` stream that executes a given transform function concurrently:

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

export class ConcurrentStream extends Transform {
  constructor(userTransform, opts) { // 1
    super({ objectMode: true, ...opts })
    this.userTransform = userTransform
    this.running = 0
    this.terminateCb = null
  }
}
```

```
_transform(chunk, enc, done) { // 2
  this.running++
  this.userTransform(
    chunk,
    enc,
    this.push.bind(this),
    this._onComplete.bind(this)
  )
  done()
}

_flush(done) { // 3
  if (this.running > 0) {
    this.terminateCb = done
  } else {
    done()
  }
}

_onComplete(err) { // 4
  this.running--
  if (err) {
    return this.emit('error', err)
  }
  if (this.running === 0) {
    this.terminateCb?.()
  }
}
}
```

Let's analyze this new class step by step:

1. As you can see, the constructor accepts a `userTransform()` function, which is then saved as an instance variable. This function will implement the transformation logic that should be executed for every object flowing through the stream. In this constructor, we invoke the parent constructor to initialize the internal state of the stream, and we enable the object mode by default.

2. Next, it is the `_transform()` method. In this method, we execute the `userTransform()` function and then increment the count of running tasks. Finally, we notify the Transform stream that the current transformation step is complete by invoking `done()`. The trick for triggering the processing of another item concurrently is exactly this. We are not waiting for the `userTransform()` function to complete before invoking `done()`; instead, we do it immediately. On the other hand, we provide a special callback to `userTransform()`, which is the `this._onComplete()` method. This allows us to get notified when the execution of `userTransform()` completes.
3. The `_flush()` method is invoked just before the stream terminates, so if there are still tasks running, we can put the release of the finish event on hold by not invoking the `done()` callback immediately. Instead, we assign it to the `this.terminateCallback` variable.
4. To understand how the stream is then properly terminated, we have to look into the `_onComplete()` method. This last method is invoked every time an asynchronous task completes. It checks whether there are any more tasks running and, if there are none, it invokes the `this.terminateCallback()` function, which will cause the stream to end, releasing the finish event that was put on hold in the `_flush()` method. Note that `_onComplete()` is a method that we introduced for convenience as part of the implementation of our `ConcurrentStream`; it is not a method we are overriding from the base Transform stream class.

The `ConcurrentStream` class we just built allows us to easily create a Transform stream that executes its tasks concurrently, but there is a caveat: it does not preserve the order of the items as they are received. In fact, while it starts every task in order, asynchronous operations can complete and push data at any time, regardless of when they are started. This property does not play well with binary streams where the order of data usually matters, but it can surely be useful with some types of object streams.

Implementing a URL status monitoring application

Now, let's apply our `ConcurrentStream` to a concrete example. Let's imagine that we want to build a simple service to monitor the status of a big list of URLs. Let's imagine all these URLs are contained in a single file and are newline-separated.

Streams can offer a very efficient and elegant solution to this problem, especially if we use our `ConcurrentStream` class to check the URLs in a concurrent fashion.

```
// check-urls.js
import { createInterface } from 'node:readline'
import { createReadStream, createWriteStream } from 'node:fs'
import { pipeline } from 'node:stream/promises'
import { ConcurrentStream } from './concurrent-stream.js'
const inputFile = createReadStream(process.argv[2]) // 1
const fileLines = createInterface({ // 2
  input: inputFile,
})
const checkUrls = new ConcurrentStream( // 3
  async (url, _enc, push, done) => {
    if (!url) {
```

```

    return done()
  }
  try {
    await fetch(url, {
      method: 'HEAD',
      timeout: 5000,
      signal: AbortSignal.timeout(5000),
    })
    push(`${url} is up\n`)
  } catch (err) {
    push(`${url} is down: ${err}\n`)
  }
  done()
}
)
const outputFile = createWriteStream('results.txt') // 4

await pipeline(fileLines, checkUrls, outputFile) // 5

console.log('All urls have been checked')
```

As we can see, with streams, our code looks very elegant and straightforward: we initialize the various components of our streaming pipeline and then we combine them together. But let's discuss some important details:

1. First, we create a Readable stream from the file given as input.
2. We leverage the `createInterface()` function from the `node:readline` module to create a stream that wraps the input stream and provides the content of the original file line by line. This is a convenient helper that is very flexible and allows us to read lines from various sources.
3. At this point, we create our `ConcurrentStream` instance. In our custom transformation logic, we expect to receive one URL at a time. If the URL is empty (e.g., if there's an empty line in the source file), we just ignore the current entry. Otherwise, we make a HEAD request to the given URL with a timeout of 5 seconds. If the request is successful, the stream emits a string that describes the positive outcome; otherwise, it emits a string that describes an error. Either way, we call the `done()` callback, which tells the `ConcurrentStream` that we have completed processing the current task. Note that, since we are handling failure gracefully, the stream can continue processing tasks even if one of them fails. Also, note that we are using both `timeout` and an `AbortSignal` because `AbortSignal` ensures that the request will fail if it takes longer than 5 seconds, regardless of whether data is actively being transferred. Some bot prevention tools deliberately keep connections open by sending responses at very slow rates, effectively causing bots to hang indefinitely. By implementing this mechanism, we ensure that requests are treated as failed if they exceed 5 seconds for any reason.

4. The last stream that we need to create is our output stream: a file called `results.txt`.
5. Finally, we have all the pieces together! We just need to combine the streams into a pipeline to let the data flow between them. And, once the pipeline completes, we print a success message.

Now, we can run the `check-urls.js` module with a command such as this:

```
node check-urls.js urls.txt
```

Here, the file `urls.txt` contains a list of URLs (one per line); for example:

```
https://mario.fyi  
https://loige.co  
http://thiswillbedownforsure.com
```

When the command finishes running, we will see that a file, `results.txt`, was created. This contains the results of the operation; for example:

```
http://thiswillbedownforsure.com is down  
https://mario.fyi is up  
https://loige.co is up
```

There is a good probability that the order in which the results are written is different from the order in which the URLs were specified in the input file. This is clear evidence that our stream executes its tasks concurrently, and it does not enforce any order between the various data chunks in the stream.



For the sake of curiosity, we might want to try replacing `ConcurrentStream` with a normal `Transform` stream and compare the behavior and performance of the two (you might want to do this as an exercise). Using `Transform` directly is way slower, because each URL is checked in sequence, but on the other hand, the order of the results in the file `results.txt` is preserved.

In the next section, we will see how to extend this pattern to limit the number of concurrent tasks running at a given time.

Unordered limited concurrent execution

If we try to run the `check-urls.js` application against a file that contains thousands or millions of URLs, we will surely run into issues. Our application will create an uncontrolled number of connections all at once, sending a considerable amount of data concurrently, and potentially undermining the stability of the application and the availability of the entire system. As we already know, the solution to keep the load and resource usage under control is to limit the number of concurrent tasks running at any given time.

Let's see how this works with streams by creating a `limited-concurrent-stream.js` module, which is an adaptation of `concurrent-stream.js`, which we created in the previous section.

Let's see what it looks like, starting from its constructor (we will highlight the changed parts):

```
export class LimitedConcurrentStream extends Transform {
  constructor (concurrency, userTransform, opts) {
    super({ ...opts, objectMode: true })
    this.concurrency = concurrency
    this.userTransform = userTransform
    this.running = 0
    this.continueCb = null
    this.terminateCb = null
  }
  // ...
}
```

We need a concurrency limit to be taken as input, and this time, we are going to save two callbacks, one for any pending `_transform` method (`continueCb`—more on this next) and another one for the callback of the `_flush` method (`terminateCb`).

Next is the `_transform()` method:

```
_transform (chunk, enc, done) {
  this.running++
  this.userTransform(
    chunk,
    enc,
    this.push.bind(this),
    this._onComplete.bind(this)
  )
  if (this.running < this.concurrency) {
    done()
  } else {
    this.continueCb = done
  }
}
```

This time, in the `_transform()` method, we must check whether we have any free execution slots before we can invoke `done()` and trigger the processing of the next item. If we have already reached the maximum number of concurrently running streams, we save the `done()` callback in the `continueCb` variable so that it can be invoked as soon as a task finishes.

The `_flush()` method remains exactly the same as in the `ConcurrentStream` class, so let's move directly to implementing the `_onComplete()` method:

```
_onComplete (err) {
  this.running--
  if (err) {
    return this.emit('error', err)
  }
  const tmpCb = this.continueCb
  this.continueCb = null
  tmpCb?.()
  if (this.running === 0) {
    this.terminateCb && this.terminateCb()
  }
}
```

Every time a task completes, we invoke any saved `continueCb()` that will cause the stream to unblock, triggering the processing of the next item.

That's it for the `LimitedConcurrentStream` class. We can now use it in the `check-urls.js` module in place of `ConcurrentStream` and have the concurrency of our tasks limited to the value that we set (check the code in the book's repository for a complete example).

Ordered concurrent execution

The concurrent streams that we created previously may shuffle the order of the emitted data, but there are situations where this is not acceptable. Sometimes, in fact, it is necessary to have each chunk emitted in the same order in which it was received. However, not all hope is lost. We can still run the transform function concurrently; all we must do is sort the data emitted by each task so that it follows the same order in which the data was received. It's important here to clearly distinguish between the internal processing logic applied to each received chunk, which can safely occur concurrently and therefore in any arbitrary order, and how the processed data is ultimately emitted by the transform stream, which might need to preserve the original order of chunks.

The technique we are going to use involves the use of a buffer to reorder the chunks while they are emitted by each running task. For brevity, we are not going to provide an implementation of such a stream, as it's quite verbose for the scope of this book. What we are going to do instead is reuse one of the available packages on npm built for this specific purpose, that is, `parallel-transform` (nodejsdp.link/parallel-transform).

We can quickly check the behavior of an ordered concurrent execution by modifying our existing `check-urls` module. Let's say that we want our results to be written in the same order as the URLs in the input file, while executing our checks concurrently. We can do this using `parallel-transform`:

```
//...
import parallelTransform from 'parallel-transform' // v1.2.0

const inputFile = createReadStream(process.argv[2])
const fileLines = createInterface({
  input: inputFile,
})
const checkUrls = parallelTransform(8, async function (url, done) {
  if (!url) {
    return done()
  }
  try {
    await fetch(url, { method: 'HEAD', timeout: 5 * 1000 })
    this.push(`${url} is up\n`)
  } catch (err) {
    this.push(`${url} is down: ${err}\n`)
  }
  done()
})
const outputFile = createWriteStream('results.txt')

await pipeline(fileLines, checkUrls, outputFile)

console.log('All urls have been checked')
```

In the example here, `parallelTransform()` creates a Transform stream in object mode that executes our transformation logic with a maximum concurrency of 8. If we try to run this new version of `check-urls.js`, we will now see that the `results.txt` file lists the results in the same order as the URLs appear in the input file. It is important to see that, even though the order of the output is the same as the input, the asynchronous tasks still run concurrently and can possibly complete in any order.



When using the ordered concurrent execution pattern, we need to be aware of slow items blocking the pipeline or growing the memory indefinitely. In fact, if there is an item that requires a very long time to complete, depending on the implementation of the pattern, it will either cause the buffer containing the pending ordered results to grow indefinitely or the entire processing to block until the slow item completes. With the first strategy, we are optimizing for performance, while with the second, we get predictable memory usage. `parallel-transform` implementation opts for predictable memory utilization and maintains an internal buffer that will not grow more than the specified maximum concurrency.

With this, we conclude our analysis of the asynchronous control flow patterns with streams. Next, we are going to focus on some piping patterns.

Piping patterns

As in real-life plumbing, Node.js streams can also be piped together by following different patterns. We can, in fact, merge the flow of two different streams into one, split the flow of one stream into two or more pipes, or redirect the flow based on a condition. In this section, we are going to explore the most important plumbing patterns that can be applied to Node.js streams.

Combining streams

In this chapter, we have stressed the fact that streams provide a simple infrastructure to modularize and reuse our code, but there is one last piece missing from this puzzle: what if we want to modularize and reuse an entire pipeline? What if we want to combine multiple streams so that they look like one from the outside? The following figure shows what this means:

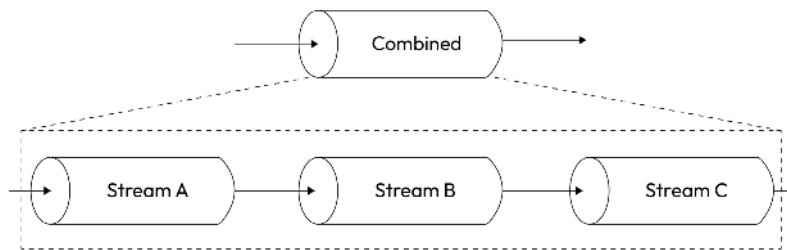


Figure 6.6: Combining streams

From Figure 6.6, we should already get a hint of how this works:

- When we write into the combined stream, we are writing into the first stream of the pipeline.
- When we read from the combined stream, we are reading from the last stream of the pipeline.

A combined stream is usually a Duplex stream, which is built by connecting the first stream to its Writable side and the last stream to its Readable side.



To create a Duplex stream out of two different streams, one Writable and one Readable, we can use an npm module such as `duplexer3` (nodejsdp.link/duplexer3) or `duplexify` (nodejsdp.link/duplexify).

But that's not enough. In fact, another important characteristic of a combined stream is that it must capture and propagate all the errors that are emitted from any stream inside the pipeline. As we already mentioned, any error event is not automatically propagated down the pipeline when we use `pipe()`, and we should explicitly attach an error listener to each stream. We saw that we could use the `pipeline()` helper function to overcome the limitations of `pipe()` with error management, but the issue with both `pipe()` and the `pipeline()` helper is that the two functions return only the last stream of the pipeline, so we only get the (last) Readable component and not the (first) Writable component.

We can verify this very easily with the following snippet of code:

```
import { createReadStream, createWriteStream } from 'node:fs'
import { Transform, pipeline } from 'node:stream'
import assert from 'node:assert/strict'

const streamA = createReadStream('package.json')
const streamB = new Transform({
  transform(chunk, _enc, done) {
    this.push(chunk.toString().toUpperCase())
    done()
  },
})
const streamC = createWriteStream('package-uppercase.json')

const pipelineReturn = pipeline(streamA, streamB, streamC, () => {
  // handle errors here
})
assert.equal(streamC, pipelineReturn) // valid

const pipeReturn = streamA.pipe(streamB).pipe(streamC)
assert.equal(streamC, pipeReturn) // valid
```

From the preceding code, it should be clear that with just `pipe()` or `pipeline()`, it would not be trivial to construct a combined stream.

To recap, a combined stream has two major advantages:

- We can redistribute it as a black box by hiding its internal pipeline.
- We have simplified error management, as we don't have to attach an error listener to each stream in the pipeline, but just to the combined stream itself.

Combining streams is common in Node.js, and `node:stream` exposes `compose()` to make it clean. It merges two or more streams into a single Duplex: writes you perform on the composite enter the first stream in the chain, reads come from the last. Backpressure is preserved end to end, and if any inner stream errors, the composite emits error and the whole chain is destroyed.

```
import { compose } from 'node:stream'
// ... define streamA, streamB, streamC
const combinedStream = compose(streamA, streamB, streamC)
```

When we do something like this, `compose` will create a pipeline out of our streams, return a new combined stream that abstracts away the complexity of our pipeline, and provide the advantages discussed previously.



Unlike `.pipe()` or `pipeline()`, `compose()` is lazy: it just builds the chain and does not start any data flow, so you still need to pipe the returned `Duplex` to a source and/or destination to move data. Use it when you want to package a reusable processing pipeline as one stream; use `pipeline()` when you want to wire a source to a destination and wait for completion.

Implementing a combined stream

To illustrate a simple example of combining streams, let's consider the case of the following two Transform streams:

- One that both compresses and encrypts the data
- One that both decrypts and decompresses the data

Using `compose`, we can easily build these streams (in a file called `combined-streams.js`) by combining some of the streams that we already have available from the core libraries:

```
import { createGzip, createGunzip } from 'node:zlib'
import { createCipheriv, createDecipheriv, scryptSync } from 'node:crypto'
import { compose } from 'node:stream'

function createKey(password) {
  return scryptSync(password, 'salt', 24)
}

export function createCompressAndEncrypt(password, iv) {
  const key = createKey(password)
  const combinedStream = compose(
    createGzip(),
    createCipheriv('aes192', key, iv)
  )
  combinedStream.iv = iv

  return combinedStream
}

export function createDecryptAndDecompress(password, iv) {
  const key = createKey(password)
  return compose(createDecipheriv('aes192', key, iv), createGunzip())
}
```

We can now use these combined streams as if they were black boxes, for example, to create a small application that archives a file by compressing and encrypting it. Let's do that in a new module named `archive.js`:

```
import { createReadStream, createWriteStream } from 'node:fs'
import { pipeline } from 'node:stream'
import { randomBytes } from 'node:crypto'
import { createCompressAndEncrypt } from './combined-streams.js'

const [, , password, source] = process.argv
const iv = randomBytes(16)
const destination = `${source}.gz.enc`

pipeline(
  createReadStream(source),
  createCompressAndEncrypt(password, iv),
  createWriteStream(destination),
  err => {
    if (err) {
      console.error(err)
      process.exit(1)
    }
    console.log(`${destination} created with iv: ${iv.toString('hex')}`)
  }
)
```

Note how we don't have to worry about how many steps there are within `archive.js`. In fact, we just treat it as a single stream within our current pipeline. This makes our combined stream easily reusable in other contexts.

Now, to run the archive module, simply specify a password and a file in the command-line arguments:

```
node archive.js mypassword /path/to/a/file.txt
```

This command will create a file called `/path/to/a/file.txt.gz.enc`, and it will print the generated initialization vector to the console.

Now, as an exercise, you could use the `createDecryptAndDecompress()` function to create a similar script that takes a password, an initialization vector, and an archived file and unarchives it. Don't worry, if you get stuck, we will have a solution implemented in this book's code repository under the file `unarchive.js`.



In real-life applications, it is generally preferable to include the initialization vector as part of the encrypted data, rather than requiring the user to pass it around. One way to implement this is by having the first 16 bytes emitted by the archive stream represent the initialization vector. The unarchive utility would need to be updated accordingly to consume the first 16 bytes before starting to process the data in a streaming fashion. This approach would add some additional complexity, which is outside the scope of this example; therefore, we opted for a simpler solution. Once you feel comfortable with streams, we encourage you to try to implement, as an exercise, a solution where the initialization vector doesn't have to be passed around by the user.

With this example, we have clearly demonstrated how important it is to combine streams. On one side, it allows us to create reusable compositions of streams, and on the other, it simplifies the error management of a pipeline.

Forking streams

We can perform a *fork* of a stream by piping a single `Readable` stream into multiple `Writable` streams. This is useful when we want to send the same data to different destinations; for example, two different sockets or two different files. It can also be used when we want to perform different transformations on the same data, or when we want to split the data based on some criteria. If you are familiar with the Unix command `tee` (nodejsdp.link/tee), this is exactly the same concept applied to Node.js streams!

Figure 6.7 gives us a graphical representation of this pattern:

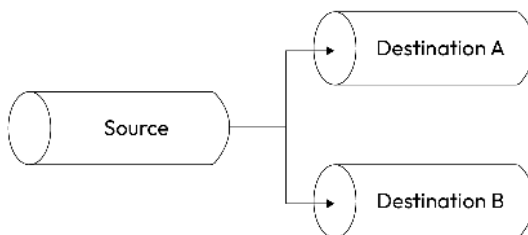


Figure 6.7: Forking a stream

Forking a stream in Node.js is quite easy, but there are a few caveats to keep in mind. Let's start by discussing this pattern with an example. It will be easier to appreciate the caveats of this pattern once we have an example at hand.

Implementing a multiple checksum generator

Let's create a small utility that outputs both the sha1 and md5 hashes of a given file. Let's call this new module `generate-hashes.js`:

```
import { createReadStream, createWriteStream } from 'node:fs'
import { createHash } from 'node:crypto'

const filename = process.argv[2]
const sha1Stream = createHash('sha1').setEncoding('hex')
const md5Stream = createHash('md5').setEncoding('hex')

const inputStream = createReadStream(filename)

inputStream.pipe(sha1Stream).pipe(createWriteStream(`${filename}.sha1`))

inputStream.pipe(md5Stream).pipe(createWriteStream(`${filename}.md5`))
```

Very simple, right? The `inputStream` variable is piped into `sha1Stream` on one side and `md5Stream` on the other. There are a few things to note that happen behind the scenes:

- Both `md5Stream` and `sha1Stream` will be ended automatically when `inputStream` ends, unless we specify `{ end: false }` as an option when invoking `pipe()`.
- The two forks of the stream will receive a reference to the same data chunks, so we must be very careful when performing side-effect operations on the data, as that would affect every stream that we are sending data to.
- Backpressure will work out of the box; the flow coming from `inputStream` will go as fast as the slowest branch of the fork. In other words, if one destination pauses the source stream to handle backpressure for a long time, all the other destinations will be waiting as well. Also, one destination blocking indefinitely will block the entire pipeline!
- If we pipe to an additional stream after we've started consuming the data at source (async piping), the new stream will only receive new chunks of data. In those cases, we can use a `PassThrough` instance as a placeholder to collect all the data from the moment we start consuming the stream. Then, the `PassThrough` stream can be read at any future time without the risk of losing any data. Just be aware that this approach might generate backpressure and block the entire pipeline, as discussed in the previous point.

Merging streams

Merging is the opposite operation to forking and involves piping a set of Readable streams into a single Writable stream, as shown in *Figure 6.8*:

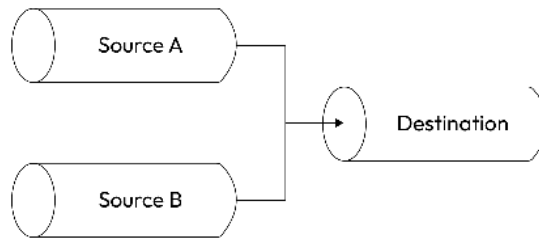


Figure 6.8: Merging streams

Merging multiple streams into one is, in general, a simple operation; however, we have to pay attention to the way we handle the end event, as piping using the default options (whereby `{ end: true }`) causes the destination stream to end as soon as one of the sources ends. This can often lead to an error, as the other active sources continue to write to an already terminated stream.

The solution to this problem is to use the option `{ end: false }` when piping multiple sources to a single destination and then invoke `end()` on the destination only when all the sources have completed reading.

Merging text files

To make a simple example, let's implement a small program that takes an output path and an arbitrary number of text files, and then merges the lines of every file into the destination file. Our new module is going to be called `merge-lines.js`. Let's define its contents, starting from some initialization steps:

```
import { createReadStream, createWriteStream } from 'node:fs'
import { Readable, Transform } from 'node:stream'
import { createInterface } from 'node:readline'

const [, , dest, ...sources] = process.argv
```

In the preceding code, we are just loading all the dependencies and initializing the variables that contain the name of the destination (`dest`) file and all the source files (`sources`).

Next, we will create the destination stream:

```
const destStream = createWriteStream(dest)
```


Now, it's time to initialize the source streams:

```
let endCount = 0
for (const source of sources) {
  const sourceStream = createReadStream(source, { highWaterMark: 16 })
  const linesStream = Readable.from(createInterface({ input: sourceStream }))
  const addLineEnd = new Transform({
    transform(chunk, _encoding, cb) {
      cb(null, `${chunk}\n`)
    },
  })

  sourceStream.on('end', () => {
    if (++endCount === sources.length) {
      destStream.end()
      console.log(`${dest} created`)
    }
  })
  linesStream
    .pipe(addLineEnd)
    .pipe(destStream, { end: false })
}
```

In this code, we initialize a source stream for each file in the sources array. Each source is read using `createReadStream()`.

The `createInterface()` function from the `node:readline` module is used to process each source file line by line, producing a `linesStream` that emits individual lines of the source file.

To ensure each emitted line ends with a newline character, we use a simple `Transform` stream (`addLineEnd`). This transform appends `\n` to each chunk of data.

We also attach an end event listener to each source stream. This listener increments a counter (`endCount`) each time a source stream finishes. When all source streams have been read, it ensures the destination stream (`destStream`) is closed, signaling the completion of the streaming pipeline.

Finally, each `linesStream` is piped through the `addLineEnd` transform and into the destination stream. During this last step, we use the `{ end: false }` option to keep the destination stream open even when one of the sources ends. The destination stream is only closed when all source streams have finished, ensuring no data is lost during the merge. This last step is where the merge happens, because we are effectively piping multiple independent streams into the same destination stream.

We can now execute this code with the following command:

```
node merge-lines.js <destination> <source1> <source2> <source3> ...
```

If you run this code with large enough files, you will notice that the destination file will contain lines that are randomly intermingled from all the source files (a low `highWaterMark` of 16 bytes makes this property even more apparent). This kind of behavior can be acceptable in some types of object streams and some text streams split by line (as in our current example), but it is often undesirable when dealing with most binary streams.

There is one variation of the pattern that allows us to merge streams in order; it consists of consuming the source streams one after the other. When the previous one ends, the next one starts emitting chunks (it is like *concatenating* the output of all the sources). As always, on npm, we can find some packages that also deal with this situation. One of them is `multistream` (<https://npmjs.org/package/multistream>).

Multiplexing and demultiplexing

There is a particular variation of the merge stream pattern in which we don't really want to just join multiple streams together, but instead, use a shared channel to deliver the data of a set of streams. This is a conceptually different operation because the source streams remain logically separated inside the shared channel, which allows us to split the stream again once the data reaches the other end of the shared channel. *Figure 6.9* clarifies this situation:

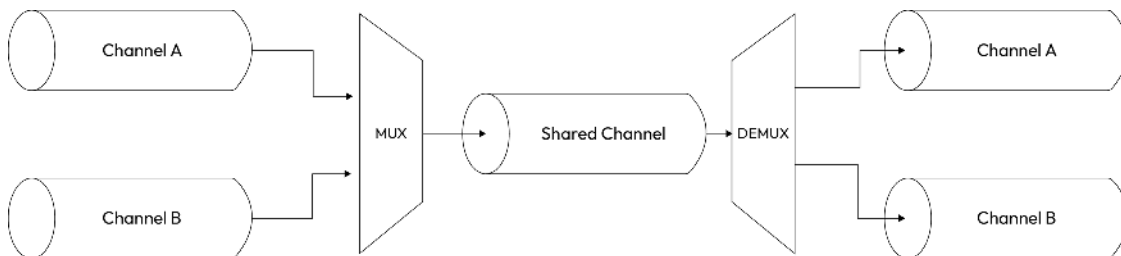


Figure 6.9: Multiplexing and demultiplexing streams

The operation of combining multiple streams (in this case, also known as channels) to allow transmission over a single stream is called **multiplexing**, while the opposite operation, namely reconstructing the original streams from the data received from a shared stream, is called **demultiplexing**. The *devices* that perform these operations are called **multiplexer** (or **mux**) and **demultiplexer** (or **demux**), respectively. This is a widely studied area in computer science and telecommunications in general, as it is one of the foundations of almost any type of communication media, such as telephony, radio, TV, and, of course, the Internet itself. For the scope of this book, we will not go too far with the explanations, as this is a vast topic.

What we want to demonstrate in this section is how it's possible to use a shared Node.js stream to transmit multiple logically separated streams that are then separated again at the other end of the shared stream.

Building a remote logger

Let's use an example to drive our discussion. We want a small program that starts a child process and redirects both its standard output and standard error to a remote server, which, in turn, saves the two streams in two separate files. So, in this case, the shared medium is a TCP connection, while the two channels to be multiplexed are the `stdout` and `stderr` of a child process. We will leverage a technique called **packet switching**, the same technique that is used by protocols such as IP, TCP, and UDP. Packet switching involves wrapping the data into *packets*, allowing us to specify various meta information that's useful for multiplexing, routing, controlling the flow, checking for corrupted data, and so on. The protocol that we are implementing in our example is very minimalist. We wrap our data into simple packets, as illustrated in *Figure 6.10*:

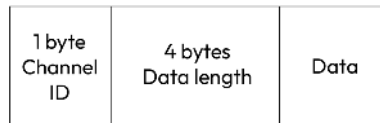


Figure 6.10: Byte structure of the data packet for our remote logger

As shown in *Figure 6.10*, the packet contains the actual data, but also a header (*Channel ID + Data length*), which will make it possible to differentiate the data of each stream and enable the demultiplexer to route the packet to the right channel.

Client side – multiplexing

Let's start to build our application from the client side. With a lot of creativity, we will call the module `client.js`. This represents the part of the application that is responsible for starting a child process and multiplexing its streams.

So, let's start by defining the module. First, we need some dependencies:

```
import { fork } from 'node:child_process'
import { connect } from 'node:net'
```

Now, let's implement a function that performs the multiplexing of a list of sources:

```
function multiplexChannels(sources, destination) {
  let openChannels = sources.length
  for (let i = 0; i < sources.length; i++) {
    sources[i]
      .on('readable', function () { // 1
        let chunk
        while ((chunk = this.read()) !== null) {
          const outBuff = Buffer.alloc(1 + 4 + chunk.length) // 2
          outBuff.writeUInt8(i, 0)
          outBuff.writeUInt32BE(chunk.length, 1)
          chunk.copy(outBuff, 5)
          console.log(`Sending packet to channel: ${i}`)
```

```

        destination.write(outBuff) // 3
    }
})
.on('end', () => { // 4
    if (--openChannels === 0) {
        destination.end()
    }
})
}
}

```

The `multiplexChannels()` function accepts the source streams to be multiplexed and the destination channel as input, and then it performs the following steps:

1. For each source stream, it registers a listener for the `readable` event, where we read the data from the stream using the non-flowing mode (the use of the non-flowing mode will give us more flexibility on reading a specific number of bytes, as we get to write the demultiplexing code).
2. When a chunk is read, we wrap it into a packet called `outBuff` that contains, in order, 1 byte (`UInt8`) for the channel ID (offset 0), 4 bytes (`UInt32BE`) for the packet size (offset 1), and then the actual data (offset 5).
3. When the packet is ready, we write it into the destination stream.
4. Finally, we register a listener for the `end` event so that we can terminate the destination stream when all the source streams have ended.



Our protocol is capable of multiplexing up to 256 different source streams because we have 1 byte to identify the channel. This is probably enough for most use cases, but if you need more, you can use more bytes to identify the channel.

Now, the last part of our client becomes very easy:

```

const socket = connect(3000, () => { // 1
    const child = fork( // 2
        process.argv[2],
        process.argv.slice(3),
        { silent: true }
    )
    multiplexChannels([child.stdout, child.stderr], socket) // 3
})

```

In this last code fragment, we perform the following operations:

1. We create a new TCP client connection to the address `localhost:3000`.
2. We start the child process by using the first command-line argument as the path, while we provide the rest of the `process.argv` array as arguments for the child process. We specify the option `{silent: true}` so that the child process does not inherit `stdout` and `stderr` of the parent.
3. Finally, we take `stdout` and `stderr` of the child process and we multiplex them into the socket's Writable stream using the `mutiplexChannels()` function.

Server side – demultiplexing

Now, we can take care of creating the server side of the application (`server.js`), where we demultiplex the streams from the remote connection and pipe them into two different files.

Let's start by creating a function called `demultiplexChannel()`:

```
import { createWriteStream } from 'node:fs'
import { createServer } from 'node:net'

function demultiplexChannel(source, destinations) {
  let currentChannel = null
  let currentLength = null

  source
    .on('readable', () => { // 1
      let chunk
      if (currentChannel === null) { // 2
        chunk = source.read(1)
        currentChannel = chunk?.readUInt8(0)
      }

      if (currentLength === null) { // 3
        chunk = source.read(4)
        currentLength = chunk?.readUInt32BE(0)
        if (currentLength === null) {
          return null
        }
      }

      chunk = source.read(currentLength) // 4
      if (chunk === null) {
        return null
      }
    })
}
```

```

    console.log(`Received packet from: ${currentChannel}`)
    destinations[currentChannel].write(chunk) // 5
    currentChannel = null
    currentLength = null
  })
  .on('end', () => { // 6
    for (const destination of destinations) {
      destination.end()
    }
    console.log('Source channel closed')
  })
}

```

The preceding code might look complicated, but it is not. Thanks to the features of Node.js Readable streams, we can easily implement the demultiplexing of our little protocol as follows:

1. We start reading from the stream using the non-flowing mode (as you can see, now we can easily read as many bytes as we need for every part of the received message).
2. First, if we have not read the channel ID yet, we try to read 1 byte from the stream and then transform it into a number.
3. The next step is to read the length of the data. We need 4 bytes for that, so it's possible (even if unlikely) that we don't have enough data in the internal buffer, which will cause the `this.read()` invocation to return `null`. In such a case, we simply interrupt the parsing and retry at the next readable event.
4. When we can finally also read the data size, we know how much data to pull from the internal buffer, so we try to read it all. Again, if this operation returns `null`, we don't yet have all the data in the buffer, so we return `null` and retry on the next readable event.
5. When we read all the data, we can write it to the right destination channel, making sure that we reset the `currentChannel` and `currentLength` variables (these will be used to parse the next packet).
6. Lastly, we make sure to end all the destination channels when the source channel ends.

Now that we can demultiplex the source stream, let's put our new function to work:

```

const server = createServer(socket => {
  const stdoutStream = createWriteStream('stdout.log')
  const stderrStream = createWriteStream('stderr.log')
  demultiplexChannel(socket, [stdoutStream, stderrStream])
})
server.listen(3000, () => console.log('Server started'))

```

In the preceding code, we first start a TCP server on port 3000; then, for each connection that we receive, we create two Writable streams pointing to two different files: one for the standard output and the other for the standard error. These are our destination channels. Finally, we use `demultiplexChannel()` to demultiplex the socket stream into `stdoutStream` and `stderrStream`.

Running the mux/demux application

Now, we are ready to try our new mux/demux application, but first, let's create a small Node.js program to produce some sample output:

```
// generate-data.js
console.log('out1')
console.log('out2')
console.error('err1')
console.log('out3')
console.error('err2')
```

Okay, now we are ready to try our remote logging application. First, let's start the server:

```
node server.js
```

Then, we'll start the client by providing the file that we want to start as a child process:

```
node client.js generateData.js
```

The client will run almost immediately, but at the end of the process, the standard input and standard output of the `generate-data.js` application will have traveled through one single TCP connection and been demultiplexed on the server into two separate files.



Please make a note that, as we are using `child_process.fork()` ([nodejsdp.link/fork](https://nodejs.org/api/child_process.html#child_process_fork)), our client will only be able to launch other Node.js modules.

Multiplexing and demultiplexing object streams

The example that we have just shown demonstrates how to multiplex and demultiplex a binary/text stream, but it's worth mentioning that the same rules apply to object streams. The biggest difference is that when using objects, we already have a way to transmit the data using atomic messages (the objects), so multiplexing would be as easy as setting a `channelID` property in each object. Demultiplexing would simply involve reading the `channelID` property and routing each object toward the right destination stream.

Another pattern involving only demultiplexing is routing the data coming from a source depending on some condition. With this pattern, we can implement complex flows, such as the one shown in *Figure 6.11*:

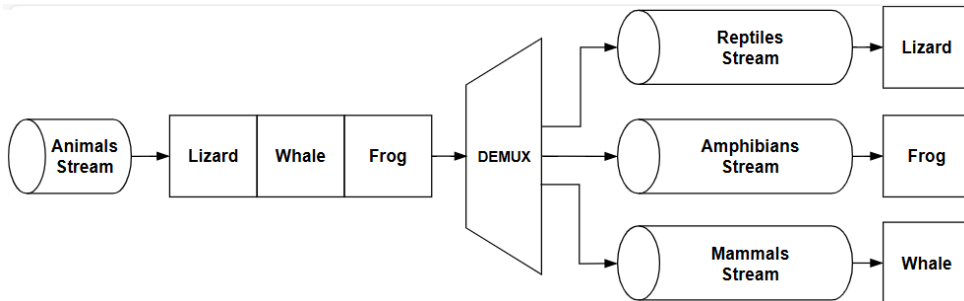


Figure 6.11: Demultiplexing an object stream

The demultiplexer used in the system in *Figure 6.11* takes a stream of objects representing *animals* and distributes each of them to the right destination stream based on the class of the animal: *reptiles*, *amphibians*, or *mammals*.

Using the same principle, we can also implement an `if...else` statement for streams. For some inspiration, take a look at the `ternary-stream` package (nodejsdp.link/ternary-stream), which allows us to do exactly that.

Readable stream utilities

In this chapter, we’ve explored how Node.js streams work, how to create custom streams, and how to compose them into efficient, elegant data processing pipelines. To complete the picture, let’s look at some utilities provided by the `node:stream` module that simplify working with Readable streams. These utilities are designed to streamline data processing in a streaming fashion and bring a functional programming flavor to stream operations.

All these utilities are methods available for any Readable stream, including `Duplex`, `PassThrough`, and `Transform` streams. Since most of these methods return a new Readable stream, they can be chained together to create expressive, pipeline-like code. Unsurprisingly, many of these methods mirror common operations available in the `Array` prototype, but they are optimized for handling streaming data.

Here’s a summary of the key methods:

Mapping and transformation

- `readable.map(fn)`: Applies a transformation function (`fn`) to each chunk in the stream, returning a new stream with the transformed data. If `fn` returns a `Promise`, the result is awaited before being passed to the output stream.
- `readable.flatMap(fn)`: Similar to `map`, but allows `fn` to return streams, iterables, or async iterables, which are then flattened and merged into the output stream.

Filtering and iteration

- `readable.filter(fn)`: Filters the stream by applying `fn` to each chunk. Only chunks for which `fn` returns a *truthy* value are included in the output stream. Supports async `fn` functions.
- `readable.forEach(fn)`: Invokes `fn` for each chunk in the stream. This is typically used for side effects rather than producing a new stream. If `fn` returns a `Promise`, it will be awaited before processing the next chunk.

Searching and evaluation

- `readable.some(fn)`: Checks if at least one chunk satisfies the condition in `fn`. Once a *truthy* value is found, the stream is destroyed, and the returned `Promise` resolves to `true`. If no chunk satisfies the condition, it resolves to `false`.
- `readable.every(fn)`: Verifies if all chunks satisfy the condition in `fn`. If any chunk fails the condition, the stream is destroyed, and the returned `Promise` resolves to `false`. Otherwise, it resolves to `true` when the stream ends.
- `readable.find(fn)`: It returns a `Promise` that will resolve to the value of the first chunk that satisfies the condition in `fn`. If no chunk meets the condition, the returned `Promise` will resolve to `undefined` once the stream ends.

Limiting and reducing

- `readable.drop(n)`: Skips the first `n` chunks in the stream, returning a new stream that starts from the $(n+1)$ th chunk.
- `readable.take(n)`: Returns a new stream that includes, at most, the first `n` chunks. Once `n` chunks are reached, the stream is terminated.
- `readable.reduce(fn, initialValue)`: Reduces the stream by applying `fn` to each chunk, accumulating a result that is returned as a `Promise`. If no `initialValue` is provided, the first chunk is used as the initial value.

The official documentation has lots of examples for all these methods and there are other less common methods we haven't explored for brevity. We recommend you check out the docs (nodejsdp.link/stream-iterators) if any of these still feel confusing and you are unsure about when to use them.

Just to give you a more practical overview, let's re-implement the processing pipeline we illustrated before to explain filtering and reducing with a custom `Transform` stream, but this time we are going to use only `Readable` stream utilities. As a reminder, in this example, we are parsing a CSV file that contains sales data. We want to calculate the total amount of profit made from sales in Italy. Every line of the CSV file has 3 fields: *type*, *country*, and *profit*. The first line contains the CSV headers.

```
import { createReadStream } from 'node:fs'
import { createInterface } from 'node:readline'
import { Readable, compose } from 'node:stream'
import { createGunzip } from 'node:zlib'

const uncompressedData = compose( // 1
```

```
    createReadStream('data.csv.gz'),
    createGunzip()
  )
  const byLine = Readable.from( // 2
    createInterface({ input: uncompressedData })
  )

  const totalProfit = await byline // 3
    .drop(1) // 4
    .map(chunk => { // 5
      const [type, country, profit] = chunk.toString().split(',')
      return { type, country, profit: Number.parseFloat(profit) }
    })
    .filter(record => record.country === 'Italy') // 6
    .reduce((acc, record) => acc + record.profit, 0) // 7

  console.log(totalProfit)
```

Here's a step-by-step breakdown of what the preceding code does:

1. The data comes from a *gzipped* CSV file, so we initially compose a file read stream and a decompression stream to create a source stream that gives uncompressed CSV data.
2. We want to read the data line by line, so we use the `createInterface()` utility from the `node:readline` module to wrap our source stream and give us a new `Readable` stream (`byLine`) that produces lines from the original stream.
3. Here's where we start to use some of the helpers we discussed in this section. Since the last helper is `.reduce()`, which returns a `Promise`, we use `await` here to wait for the returned `Promise` to resolve and to capture the final result in the `total` variable.
4. The first helper we use is `.drop(1)`, which allows us to skip the first line of the uncompressed source data. This line will contain the CSV header ("*type,country,profit*") and no useful data, so it makes sense to skip it. This operation returns a new `Readable` stream, so we can chain other helper methods.
5. The next helper we use in the chain is `.map()`. In the mapping function, we provide all the necessary logic to parse a line from the original CSV file and convert it into a record object containing the fields `type`, `country`, and `profit`. This operation returns another `Readable` stream, so we can keep chaining more helper functions to continue building our processing logic.
6. The next step is `.filter()`, which we use to retain only records that represent profit associated with the country Italy. Once again, this operation gives us a new `Readable` stream.
7. The last step of the processing pipeline is `.reduce()`. We use this helper to aggregate all the filtered records by summing their profit. As we mentioned before, this operation will give us a `Promise` that will resolve to the total profit once the stream completes.

This example shows how to create stream processing pipelines using a more direct approach. In this approach, we chain helper methods, and we have all the transformation logic clearly visible in the same context (assuming we define all the transformation functions in line). This approach can be particularly convenient in situations where the transformation logic is very simple, and you don't need to build highly specialized and reusable custom `Transform` streams.



Note that, in this example, we created our own basic way of parsing records out of CSV lines rather than using a dedicated library for it. We did this just to have an excuse to showcase how to use the `.drop()` and `.map()` methods. Our implementation is very rudimentary, and it doesn't handle all the possible edge cases. This is fine because we know there aren't edge cases (e.g., quoted fields) in our input data, but in real-world projects, we would recommend using a reliable CSV parsing library instead.

Web Streams

The **WHATWG Streams Standard** (nodejsdp.link/web-streams) provides a standardized API for working with streaming data, known as “**Web Streams**.” While inspired by Node.js streams, it has its own distinct implementation and is designed to be a universal standard for the broader JavaScript ecosystem, including browsers.

About a decade after the initial development of Node.js streams, Web Streams emerged to address the lack of a native streaming API in browser environments, something that made it difficult to efficiently work with large datasets on the frontend.

Today, most modern browsers support the Web Streams standard natively, making it an ideal choice for building streaming pipelines within the browser. In contrast, Node.js streams are not natively available in browsers. You could bring Node.js streams to the browser by installing them as a library in your project, but their utility is limited since native APIs like `fetch` use Web Streams to send requests or read responses incrementally. Given this, using Web Streams in the browser is the recommended choice.

Web Streams have also been implemented in Node.js, effectively giving us two competing APIs to deal with streaming data. However, at the time of writing, Web Streams is still relatively new and hasn't yet reached the same level of adoption as native Node.js streams within the large Node.js ecosystem. That's why this chapter focused mainly on Node.js streams, but understanding Web Streams is still an important piece of knowledge, and we expect it to become more relevant in the coming years.

Fortunately, getting started with Web Streams should be easy if you have been following this chapter. Most of the concepts are aligned, and the primary differences lie in function names and arguments, which is something that can be easily learned by checking the Web Streams API documentation.

One aspect that is worth exploring here is the interoperability between Node.js and Web Streams. Fortunately, it's possible to convert Node.js stream objects to equivalent Web Stream objects and vice versa. This makes it easy to transition or work with third-party libraries that use Web Streams in the context of Node.js.

Let's briefly discuss how this interoperability works.

In the Web Streams standard, we have 3 primary types of objects:

- **ReadableStream**: Source of streaming data and pretty much equivalent to a Readable Node.js stream.
- **WritableStream**: Destination for streaming data; equivalent to a Node.js Writable stream.
- **TransformStream**: Allows you to transform streaming data in a streaming pipeline. Equivalent to a Node.js Transform stream.

Note how these concepts match almost perfectly. Also note how, thanks to the *Stream* suffix of the Web Streams classes, we don't have naming conflicts between equivalent streaming abstractions.

Converting Node.js streams to Web Streams

You can easily convert Node.js streams to equivalent Web Streams objects by using the `.toWeb(sourceNodejsStream)` method available respectively in the `Readable`, `Writable`, and `Transform` classes.

Let's see what the syntax looks like:

```
import { Readable, Writable, Transform } from 'node:stream'

const nodeReadable = new Readable({/*...*/}) // Readable
const webReadable = Readable.toWeb(nodeReadable) // ReadableStream

const nodeWritable = new Writable({/*...*/}) // Writable
const webWritable = Writable.toWeb(nodeWritable) // WritableStream

const nodeTransform = new Transform({/*...*/}) // Transform
const webTransform = Transform.toWeb(nodeTransform) // TransformStream
```

Converting Web Streams to Node.js streams

The `Readable`, `Writable`, and `Transform` classes also expose methods to convert a Web Stream to an equivalent Node.js stream. These methods, unsurprisingly, have the following signature: `.fromWeb(sourceWebStream)`.

Let's see a quick example to clarify the syntax:

```
import { Readable, Writable, Transform } from 'node:stream'
import {
  ReadableStream,
  WritableStream,
  TransformStream,
} from 'node:stream/web'
```

```
const webReadable = new ReadableStream({/*...*/}) // ReadableStream
const nodeReadable = Readable.fromWeb(webReadable) // Readable

const webWritable = new WritableStream({/*...*/}) // WritableStream
const nodeWritable = Writable.fromWeb(webWritable) // Writable

const webTransform = new TransformStream({/*...*/}) // TransformStream
const nodeTransform = Transform.fromWeb(webTransform) // Transform
```

The last two snippets illustrate how easy it is to convert stream types between Node.js streams and Web Streams.

One important detail to keep in mind is that these conversions don't destroy the source stream but rather wrap it in a new object that is compliant with the target API. For example, when we convert a Node.js Readable stream to a web ReadableStream, we can still read from the source stream while also reading from the new Web Stream. The following example should help to clarify this idea:

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

const nodeReadable = new Readable({
  read() {
    this.push('Hello, ')
    this.push('world!')
    this.push(null)
  },
})

const webReadable = Readable.toWeb(nodeReadable)

nodeReadable.pipe(process.stdout)
webReadable.pipeTo(Writable.toWeb(process.stdout))
```

In the preceding example, we are defining a Node.js stream that emits the string “Hello, world!” in 2 chunks before completing. We convert this stream into an equivalent Web Stream, then we pipe both the source Node.js stream and the newly created Web Stream to standard output.

This code will produce the following output:

```
Hello, Hello, world!world!
```

This is because, every time that the source Node.js stream emits a chunk, the same chunk is also emitted by the associated Web Stream.



The `.fromWeb()` and `.toWeb()` methods are implementations of the **Adapter pattern** that we will discuss in more detail in *Chapter 8, Structural Design Patterns*.

Stream consumer utilities

As we've repeated countless times throughout this chapter, streams are designed to transfer and process large amounts of data in small chunks. However, there are situations where you need to consume the entire content of a stream and accumulate it in memory. This is more common than it might seem, largely because many abstractions in the Node.js ecosystem use streams as the fundamental building block for data transfer. This design provides a great deal of flexibility, but it also means that sometimes you need to handle chunk-by-chunk data manually. In such cases, it's important to understand how to convert a stream of discrete chunks into a single, buffered piece of data that can be processed as a whole.

A good example of this is the low-level `node:http` module, which allows you to make HTTP requests. When handling an HTTP response, Node.js represents the response body as a `Readable` stream. This means you're expected to process the response data incrementally, as chunks arrive.

But what if you know in advance that the response body contains a JSON-serialized object? In that case, you can't process the chunks independently; you need to wait until the entire response has been received so you can parse it as a complete string using `JSON.parse()`.

A simple implementation of this pattern might look like the following code:

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

const req = request('http://example.com/somefile.json', res => { // 1
  let buffer = '' // 2
  res.on('data', chunk => {
    buffer += chunk
  })
  res.on('end', () => { // 3
    console.log(JSON.parse(buffer))
  })
})
req.end() // 4
```

To better understand this example, let's discuss its main points:

1. Here, a request is being made to `http://example.com/somefile.json`. The second argument is a callback that receives the response (`res`) object, which is a `Readable` stream. This stream emits chunks of data as they arrive over the network.
2. Inside the response callback, we initialize an empty string called `buffer`. As each chunk of data arrives (via the `'data'` event), we concatenate it to the `buffer` string. This effectively buffers the entire response body in memory. This approach is necessary when you need to handle the whole response as a complete unit – for example, when parsing JSON, since `JSON.parse()` only works on complete strings.

3. Once the entire response has been received and no more data will arrive ('end' event), we use `JSON.parse()` to deserialize the accumulated string into a JavaScript object. The resulting object is then logged to the console.
4. Finally, `req.end()` is called to signal that no request body will be sent (our request is complete and can be forwarded). Since this is a GET request with no body, it's necessary to explicitly finalize the request.

A final point worth noting is that this code doesn't require `async/await` because it relies entirely on event-based callbacks, which is the traditional way of handling asynchronous operations in Node.js streams.

This solution works, but it's a bit *boilerplate-heavy*. Thankfully, there's a better solution, thanks to the `node:stream/consumers` module.

This built-in library was introduced in Node.js version 16 to expose various utilities that make it easy to consume the entire content from a Node.js Readable instance or a Web Streams ReadableStream instance.

This module exposes the `consumers` object, which implements the following static methods:

- `consumers.arrayBuffer(stream)`
- `consumers.blob(stream)`
- `consumers.buffer(stream)`
- `consumers.text(stream)`
- `consumers.json(stream)`

Each one of these methods *consumes* the given stream and returns a Promise that resolves only when the stream has been fully consumed.

It's easy to guess that each method accumulates the data into a different kind of object. `arrayBuffer()`, `blob()`, and `buffer()` will accumulate chunks as binary data in an `ArrayBuffer`, a `Blob`, or a `Buffer` instance, respectively. `text()` accumulates data in a string object, while `json()` accumulates data in a string object and will also try to deserialize the data using `JSON.parse()` before resolving the corresponding Promise.

This means that we can rewrite the previous example as follows:

```
import { request } from 'node:https'
import consumers from 'node:stream/consumers'

const req = request(
  'http://example.com/somefile.json',
  async res => {
    const buffer = await consumers.json(res)
    console.log(buffer)
  }
)
req.end()
```

Much more concise and elegant, isn't it?



If you use `fetch` to make HTTP(s) requests, the response object provided by the `fetch` API has various consumers built in. You could rewrite the previous example as follows:

```
const res = await fetch('http://example.com/somefile.json')
const buffer = await res.json()
console.log(buffer)
```

The response object (`res`) also exposes `.blob()`, `.arrayBuffer()`, and `.text()` if you want to accumulate the response data as a binary buffer or as text. Note that the `.buffer()` method is missing, though. This is because the `Buffer` class is not part of the Web standard, but it exists only in Node.js.

Summary

In this chapter, we shed some light on Node.js streams and some of their most common use cases. We learned why streams are so acclaimed by the Node.js community and we mastered their basic functionality, enabling us to discover more and navigate comfortably in this new world. We analyzed some advanced patterns and started to understand how to connect streams in different configurations, grasping the importance of interoperability, which is what makes streams so versatile and powerful.

If we can't do something with one stream, we can probably do it by connecting other streams together, and this works great with the *one thing per module* philosophy. At this point, it should be clear that streams are not just a *good-to-know* feature of Node.js; they are an essential part – a crucial pattern to handle binary data, strings, and objects. It's not by chance that we dedicated an entire chapter to them.

In the next few chapters, we will focus on the traditional object-oriented design patterns. But don't be fooled; even though JavaScript is, to some extent, an object-oriented language, in Node.js, the functional or hybrid approach is often preferred. Get rid of every prejudice before reading the next chapters.

Exercises

- **6.1 Data compression efficiency:** Write a command-line script that takes a file as input and compresses it using the different algorithms available in the `zlib` module (Brotli, Deflate, Gzip). You want to produce a summary table that compares the algorithm's compression time and compression efficiency on the given file. Hint: This could be a good use case for the `fork` pattern, but remember that we made some important performance considerations when we discussed it earlier in this chapter.

- **6.2 Stream data processing:** On Kaggle, you can find a lot of interesting datasets, such as London Crime Data (nodejsdp.link/london-crime). You can download the data in CSV format and build a stream processing script that analyzes the data and tries to answer the following questions:
 - Did the number of crimes go up or down over the years?
 - What are the most dangerous areas of London?
 - What is the most common crime per area?
 - What is the least common crime?

Hint: You can use a combination of Transform streams and PassThrough streams to parse and observe the data as it is flowing. Then, you can build in-memory aggregations for the data, which can help you answer the preceding questions. Also, you don't need to do everything in one pipeline; you could build very specialized pipelines (for example, one per question) and use the fork pattern to distribute the parsed data across them.

- **6.3 File share over TCP:** Build a client and a server to transfer files over TCP. Extra points if you add a layer of encryption on top of that and if you can transfer multiple files at once. Once you have your implementation ready, give the client code and your IP address to a friend or a colleague, then ask them to send you some files! Hint: You could use mux/demux to receive multiple files at once.
- **6.4 Animations with Readable streams:** Did you know you can create amazing terminal animations with just Readable streams? Well, to understand what we are talking about here, try to run `curl parrot.live` in your terminal and see what happens! If you think that this is cool, why don't you try to create something similar? Hint: If you need some help with figuring out how to implement this, you can check out the actual source code of `parrot.live` by simply accessing its URL through your browser.

7

Creational Design Patterns

This might seem a little unexpected for a book with “Design Patterns” in its title, but it took six chapters to get to a point where we can start talking a little bit more in depth about design patterns, at least in the conventional sense. So, what is a design pattern? In simple terms, a design pattern is a reusable solution to a recurring problem. The term is really broad in its definition and it can span multiple domains of an application. However, the term is often associated with a well-known set of object-oriented patterns that were popularized in the 90s by the book *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education, by the legendary **Gang of Four (GoF)**: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. We will often refer to these specific sets of patterns as *traditional* design patterns or GoF design patterns.

Applying this set of object-oriented design patterns in JavaScript is not as linear and formal as it would be in a classical object-oriented language. As we know, JavaScript is object-oriented, prototype-based, and has dynamic typing. It also treats functions as first-class citizens and allows functional programming styles. These characteristics make JavaScript a very versatile language, which gives tremendous power to the developer, but at the same time, it causes fragmentation of programming styles, conventions, techniques, and, ultimately, the patterns of its ecosystem. With JavaScript, there are so many ways to achieve the same result that each developer has their own opinion on what’s the best way to approach a problem. A clear demonstration of this phenomenon is the abundance of frameworks and opinionated libraries in the JavaScript ecosystem; probably no other language has ever seen so many, especially since Node.js has given new, astonishing possibilities to JavaScript and has created so many new scenarios.

In this context, the nature of JavaScript affects traditional design patterns too. There are so many ways in which traditional design patterns can be implemented in JavaScript that the traditional, strongly object-oriented implementation stops being relevant.

In some cases, the traditional implementation of these design patterns is not even possible because JavaScript, as we know, doesn’t have *real* abstract classes or interfaces. What doesn’t change, though, is the original idea at the base of each pattern, the problem it solves, and the concepts at the heart of the solution.



If you're familiar with TypeScript, you probably know that it adds language features such as abstract classes and interfaces. These can make it easier to express some design patterns in a way that feels closer to the original GoF implementations. In this book, however, we focus on plain JavaScript to show how these patterns work using the core features of the language itself. TypeScript is a compile-time abstraction. Its types, interfaces, and abstract classes—while helpful—vanish once the code is transpiled, leaving only JavaScript running in the environment. That's why the authors believe it's more valuable to understand how these patterns behave at runtime rather than relying too heavily on tools that don't exist during execution. Once you've mastered the fundamentals, you'll be in a great position to layer on any abstraction you prefer, whether that's TypeScript, JSDoc annotations, or something else entirely.

In this chapter and the two that follow, we will see how some of the most important GoF design patterns apply to Node.js and its philosophy, thus rediscovering their importance from another perspective. Among these traditional patterns, we will also have a look at some “less traditional” design patterns born from within the JavaScript ecosystem itself.

In this chapter, in particular, we'll take a look at a class of design patterns called **creational** patterns. As the name suggests, these patterns address problems related to the creation of objects. For example, the *Factory* pattern allows us to encapsulate the creation of an object within a function. The *Revealing Constructor* pattern allows us to expose private object properties and methods only during the object's creation, while the *Builder* pattern simplifies the creation of complex objects. Finally, the *Singleton* pattern and the *Dependency Injection* pattern help us with wiring modules within our applications.



In this chapter, as well as in the following two, we assume that you have some notion of how inheritance works in JavaScript. Please also be advised that we will often use generic and more intuitive diagrams to describe a pattern in place of standard UML. This is because many patterns can have an implementation based not only on classes but also on objects and even functions.

Factory

We'll begin our journey from one of the most common design patterns in Node.js: **Factory**. As you will see, the Factory pattern is very versatile and serves more than one purpose. Its main advantage is its ability to decouple the creation of an object from a specific implementation. This allows us, for example, to create an object whose class or shape is determined at runtime.

A factory also lets us expose a much smaller surface area than a class. While a class can be extended, instantiated directly, or even monkey-patched, a factory—being just a function—offers fewer ways for consumers to misuse or interfere with the internals of the implementation. This makes the resulting API less error-prone and easier to maintain.

Finally, factories can also help enforce encapsulation by leveraging closures, keeping internal details truly private and hidden from the outside world.

Decoupling object creation and implementation

We already stressed the fact that, in JavaScript, the functional paradigm is often preferred to a purely object-oriented design for its simplicity, usability, and *small surface area*. This is especially true when creating new object instances. In fact, invoking a factory instead of directly creating a new object from a class using the `new` operator or `Object.create()` is so much more convenient and flexible in several respects.

First and foremost, a factory allows us to *separate the creation of an object from its implementation*. Essentially, a factory wraps the creation of a new instance, giving us more flexibility and control in the way we do it. Inside the factory, we can choose to create a new instance of a class using the `new` operator, or leverage closures to dynamically build a stateful object literal, or even return a different object type based on a particular condition. The consumer of the factory is totally agnostic about how the creation of the instance is carried out. The truth is that by using `new`, we are binding our code to one specific way of creating an object, while with a factory, we can have much more flexibility, almost for free. As a quick example, let's consider a simple factory that creates an `Image` object:

```
function createImage(name) {  
  return new Image(name)  
}  
const image = createImage('photo.jpeg')
```

The `createImage()` factory might look totally unnecessary; why not instantiate the `Image` class by using the `new` operator directly? Why not write something like the following:

```
const image = new Image(name)
```

As we already mentioned, using `new` binds our code to one particular type of object, which, in the preceding case, is the `Image` object type. A factory, on the other hand, gives us much more flexibility. Imagine that we want to refactor the `Image` class, splitting it into smaller classes, one for each image format that we support.

If we exposed a factory as the only means to create new images, we could simply rewrite it as follows, without breaking any of the existing code:

```
const jpgRgx = /\.jpe?g$/  
const gifRgx = /\.gif$/  
const pngRgx = /\.png$/  
  
function createImage(name) {  
  if (name.match(jpgRgx)) {  
    return new ImageJpeg(name)  
  }  
  if (name.match(gifRgx)) {  
    return new ImageGif(name)  
  }  
}
```

```
    }  
    if (name.match(pngRgx)) {  
      return new ImagePng(name)  
    }  
    throw new Error('Unsupported format')  
  }  
}
```

Our factory also allows us to keep the classes hidden and prevents them from being extended or modified (remember the principle of small surface area?). In JavaScript, this can be achieved by exporting only the factory while keeping the classes private.

A mechanism to enforce encapsulation

A factory can also be used as an **encapsulation** mechanism, thanks to closures.

Encapsulation refers to controlling the access to some internal details of a component by preventing external code from manipulating them directly. The interaction with the component happens only through its public interface, isolating the external code from the changes in the implementation details of the component. Encapsulation is a fundamental principle of object-oriented design, together with inheritance, polymorphism, and abstraction.

In JavaScript, one of the main ways to enforce encapsulation is through function scopes and closures. A factory makes it straightforward to enforce private variables. Consider the following, for example:

```
function createPerson(name) {  
  const privateProperties = {}  
  
  const person = {  
    setName(name) {  
      if (!name) {  
        throw new Error('A person must have a name')  
      }  
      privateProperties.name = name  
    },  
    getName() {  
      return privateProperties.name  
    }  
  }  
  
  person.setName(name)  
  return person  
}
```

In the preceding code, we leverage a closure to create two objects: a person object, which represents the public interface returned by the factory, and a group of `privateProperties` that are inaccessible from the outside and that can be manipulated only through the interface provided by the person object. For example, in the preceding code, we make sure that a person's name is never empty; this would be impossible to enforce if name were just a normal property of the person object.



Using closures is not the only technique that we have for enforcing encapsulation. In fact, other possible approaches are:

- Using private class fields (the hashtag or pound symbol, "#", prefix syntax). More on this at nodejsdp.link/tc39-private-fields.
- Using WeakMaps. More on this at nodejsdp.link/weakmaps-private.
- Using symbols, as explained in the following article: nodejsdp.link/symbol-private.
- Defining private variables in a constructor (as recommended by Douglas Crockford: nodejsdp.link/crockford-private). This is the legacy but also best-known approach.
- Using conventions, for example, prefixing the name of a property with an underscore, "_". However, this does not technically prevent a member from being read or modified from the outside.

Building a simple code profiler

Now, let's work on a complete example using a factory. Let's build a simple *code profiler*: an object that helps measure how long a piece of code takes to run. Profilers are often used to identify performance bottlenecks by tracking the duration of specific operations or functions. Our profiler will expose the following methods:

- A `start()` method that triggers the start of a profiling session
- An `end()` method to terminate the session and log its execution time to the console

Let's start by creating a file named `profiler.js`, which will have the following content:

```
class Profiler {
  constructor (label) {
    this.label = label
    this.lastTime = null
  }

  start() {
    this.lastTime = process.hrtime()
  }

  end() {
```

```
    const diff = process.hrtime(this.lastTime)
    console.log(`Timer "${this.label}" took ${diff[0]} seconds ` +
      `and ${diff[1]} nanoseconds.`)
  }
}
```

The `Profiler` class we just defined uses the default high-resolution timer of Node.js to save the current time when `start()` is invoked, and then calculate the elapsed time when `end()` is executed, printing the result to the console.

Now, if we are going to use such a profiler in a real-world application to calculate the execution time of different routines, we can easily imagine the huge amount of profiling information printed to the console, especially in a production environment. What we may want to do instead is redirect the profiling information to another source, for example, a dedicated log file, or disable the profiler altogether if the application is running in production mode. It's clear that if we were to instantiate a `Profiler` object directly by using the `new` operator, we would need some extra logic in the client code or in the `Profiler` object itself in order to switch between the different logics.

Alternatively, we can use a factory to abstract the creation of the `Profiler` object so that, depending on whether the application runs in production or development mode, we can return a fully working `Profiler` instance or a mock object with the same interface but with empty methods. This is exactly what we are going to do in our `profiler.js` module. Instead of exporting the `Profiler` class, we will export only our factory. The following is its code:

```
const noopProfiler = {
  start () {},
  end () {}
}

export function createProfiler (label) {
  if (process.env.NODE_ENV === 'production') {
    return noopProfiler
  }

  return new Profiler(label)
}
```

The `createProfiler()` function is our factory, and its role is to abstract the creation of a `Profiler` object from its implementation. If the application is running in production mode, we return `noopProfiler`, which essentially doesn't do anything, effectively disabling any profiling. If the application is not running in production mode, then we create and return a new, fully functional `Profiler` instance.

Thanks to JavaScript's dynamic typing, we were able to return an object instantiated with the new operator in one circumstance and a simple object literal in the other (this is also known as **duck typing**, and you can read more about it at nodejsdp.link/duck-typing). This confirms how we can create objects in any way we like inside the factory function. We could also execute additional initialization steps or return a different type of object based on particular conditions, all of this while isolating the consumer of the object from all these details. We can easily understand the power of this simple pattern.



In the previous example, we used `NODE_ENV` to build a smart factory that switches between production and development versions of our profiler. This can be a useful technique, but use it with care. Relying too much on `NODE_ENV` to change code behavior can make your code harder to test, debug, and maintain, especially if certain branches only run in production. Use it for configuration, not control flow.

Now, let's play with our profiler factory a bit. Let's create an algorithm to calculate all the factors of a given number and use our profiler to record its running time:

```
// index.js
import { createProfiler } from './profiler.js'

function getAllFactors(n) {
  let intNumber = n
  const profiler = createProfiler(`Finding all factors of ${intNumber}`)

  profiler.start()
  const factors = []
  for (let factor = 2; factor <= intNumber; factor++) {
    while (intNumber % factor === 0) {
      factors.push(factor)
      intNumber /= factor
    }
  }
  profiler.end()

  return factors
}

const myNumber = process.argv[2]
const myFactors = getAllFactors(myNumber)
console.log(`Factors of ${myNumber} are: `, myFactors)
```

The profiler variable contains our Profiler object, whose implementation will be decided by the `createProfiler()` factory at runtime, based on the `NODE_ENV` environment variable.

For example, if we run the module in production mode, we will get no profiling information:

```
NODE_ENV=production node index.js 2201307499
```

However, if we run the module in development mode, we will see the profiling information printed to the console:

```
node index.js 2201307499
Timer "Finding all factors of 2201307499" took 0 seconds and 6010750
nanoseconds.
Factors of 2201307499 are: [ 38737, 56827 ]
```

The example that we just presented is just a simple application of the Factory function pattern, but it clearly shows the advantages of separating an object's creation from its implementation.

In the wild

As we said, factories are very common in Node.js. We can find one example in the popular *Knex* (nodejsdp.link/knex) package. Knex is a SQL query builder that supports multiple databases. Its package exports just a function, which is a factory. The factory performs various checks, selects the right dialect object to use based on the database engine, and finally, creates and returns the Knex object. Take a look at the code at nodejsdp.link/knex-factory.

Builder

Builder is a creational design pattern that simplifies the creation of complex objects by providing a fluent interface, which allows us to build the object step by step. This greatly improves the readability and the general developer experience when creating such complex objects.

The most apparent situation in which we could benefit from the Builder pattern is a class with a constructor that has a long list of arguments or takes many complex parameters as input. Usually, these kinds of classes require so many parameters in advance because all of them are necessary to build an instance that is complete and in a consistent state, so it's necessary to take this into account when considering potential solutions.

So, let's see the general structure of the pattern. Imagine you are building the backend for a video game about racing boats. You might have a *Boat* class with a constructor such as the following:

```
class Boat {
  constructor (hasMotor, motorCount, motorBrand, motorModel,
              hasSails, sailsCount, sailsMaterial, sailsColor,
              hullColor, hasCabin) {

    // ...
  }
}
```

Invoking such a constructor would create some hard-to-read code, which is easily prone to errors. Looking at your code after a few days, you might be asking yourself, “Which argument is what?”. Even worse if you need to refactor the order of the arguments or add new arguments.

Take the following code, for example:

```
const myBoat = new Boat(true, 2, 'Best Motor Co. ', 'OM123', true, 1,
                        'fabric', 'white', 'blue', false)
```

A first step to improve the design of this constructor is to aggregate all arguments in a single object literal (generally called a *configuration object*), such as the following:

```
class Boat {
  constructor(allParameters) {
    // ...
  }
}

const myBoat = new Boat({
  hasMotor: true,
  motorCount: 2,
  motorBrand: 'Best Motor Co. ',
  motorModel: 'OM123',
  hasSails: true,
  sailsCount: 1,
  sailsMaterial: 'fabric',
  sailsColor: 'white',
  hullColor: 'blue',
  hasCabin: false
})
```

As we can note from the previous code, this new constructor is indeed much better than the original one, as it allows us to clearly see which parameter receives each value. Also, it's easy to add new parameters or to change their order with limited impact on existing code. However, we can do even better than this. One drawback of using a single object literal to pass all inputs at once is that the only way to know what the actual inputs are is to look at the class documentation or, even worse, into the code of the class. In addition to that, there is no enforced protocol that guides the developers toward the creation of a coherent class. For example, if we specify `hasMotor: true`, then we are required to also specify a `motorCount`, a `motorBrand`, and a `motorModel`, but there is nothing in this interface that conveys this information to us.



Using TypeScript could help with making it easy to understand what parameters are available and what their type is, but you might have to come up with a more sophisticated type definition for the configuration object if you want to express dependencies between parameters. For example, you could create nested properties for `motor` and `sails` and mark them as optional, as in the following snippet:

```
export type BoatConfig = {
  motor?: {
    count: number
    brand: string
    model: string
  }
  sails?: {
    count: number
    material: string
    color: string
  }
  hullColor: string
  hasCabin: boolean
}

class Boat {
  constructor(config: BoatConfig) {
    // ...
  }
}

const myBoat = new Boat({
  motor: {
    count: 2,
    brand: 'Best Motor Co. ',
    model: 'OM123',
  },
  sails: {
    count: 1,
    material: 'fabric',
    color: 'white',
  },
})
```

```
hullColor: 'blue',  
hasCabin: false,  
})
```



With this approach, the type system will help you get the configuration right. If you want to create a boat that has a motor or sails, you'll need to provide all the necessary configuration for those properties as nested objects. While this approach works in simple cases, we will see how the Builder pattern offers a more flexible alternative that works well even with just plain JavaScript, and how it allows us to keep the list of parameters flat.

The Builder pattern can be an excellent solution to these problems, offering a fluent interface that is simple to read, self-documenting, and that guides the creation of a coherent object. Let's look at the `BoatBuilder` class, which implements the Builder pattern for the `Boat` class:

```
// boat.js  
export class BoatBuilder {  
  withMotors (count, brand, model) {  
    this.hasMotor = true  
    this.motorCount = count  
    this.motorBrand = brand  
    this.motorModel = model  
    return this  
  }  
  
  withSails (count, material, color) {  
    this.hasSails = true  
    this.sailsCount = count  
    this.sailsMaterial = material  
    this.sailsColor = color  
    return this  
  }  
  
  hullColor (color) {  
    this.hullColor = color  
    return this  
  }  
  
  withCabin () {  
    this.hasCabin = true  
    return this  
  }  
}
```

```
build() {  
  return new Boat({  
    hasMotor: this.hasMotor,  
    motorCount: this.motorCount,  
    motorBrand: this.motorBrand,  
    motorModel: this.motorModel,  
    hasSails: this.hasSails,  
    sailsCount: this.sailsCount,  
    sailsMaterial: this.sailsMaterial,  
    sailsColor: this.sailsColor,  
    hullColor: this.hullColor,  
    hasCabin: this.hasCabin  
  })  
}  
}
```

To fully appreciate the positive impact that the Builder pattern has on the way we create our Boat objects, let's see an example of that:

```
// index.js  
const myBoat = new BoatBuilder()  
  .withMotors(2, 'Best Motor Co. ', 'OM123')  
  .withSails(1, 'fabric', 'white')  
  .withCabin()  
  .hullColor('blue')  
  .build()
```

As we can see, the role of our BoatBuilder class is to collect all the parameters needed to create a Boat using some helper methods. We usually have a method for each parameter or set of related parameters, but there is no exact rule for that. It is down to the designer of the Builder class to decide the name and behavior of each method responsible for collecting the input parameters.

We can instead summarize some general rules for implementing the Builder pattern, as follows:

- The main objective is to break down a complex constructor into multiple, more readable, and more manageable steps.
- Try to create builder methods that can set multiple related parameters at once.
- Deduce and implicitly set parameters based on the values received as input by a setter method, and in general, try to encapsulate as much parameter-setting-related logic into the setter methods so that the consumer of the builder interface is free from doing so.
- If necessary, it's possible to further manipulate the parameters (for example, type casting, normalization, supporting default values, or extra validation) before passing them to the constructor of the class being built to simplify the work left to do by the builder class consumer even more.

- Consider providing sensible default values for parameters that are not explicitly set. This makes the builder easier to use, especially when some configuration options are optional or rarely customized.



In JavaScript, the Builder pattern can also be applied to invoke functions, not just to build objects using their constructor. In fact, from a technical point of view, the two scenarios are almost identical. The major difference when dealing with functions is that instead of having a `build()` method, we would have an `invoke()` method that invokes the complex function with the parameters collected by the builder object and returns any eventual result to the caller.

Next, we will work on a more concrete example that makes use of the Builder pattern we've just learned about.

Implementing a URL object builder

We want to implement a `Url` class that can hold all the components of a standard URL, validate them, and format them back into a string. This class is going to be intentionally simple and minimal, so for standard production use, we recommend the built-in URL class ([nodejsdp.link/docs-url](https://nodejs.org/docs/latest/api/url)).

Now, let's implement our custom `Url` class in a file called `url.js`:

```
export class Url {
  constructor(protocol, username, password, hostname,
    port, pathname, search, hash) {
    this.protocol = protocol
    this.username = username
    this.password = password
    this.hostname = hostname
    this.port = port
    this.pathname = pathname
    this.search = search
    this.hash = hash

    this.validate()
  }

  validate() {
    if (!(this.protocol && this.hostname)) {
      throw new Error('Must specify at least a protocol and a hostname')
    }
  }

  toString() {
```

```

    let url = ''
    url += `${this.protocol}://`
    if (this.username && this.password) {
        url += `${this.username}:${this.password}@`
    }
    url += this.hostname
    if (this.port) {
        url += this.port
    }
    if (this.pathname) {
        url += this.pathname
    }
    if (this.search) {
        url += `?${this.search}`
    }
    if (this.hash) {
        url += `#${this.hash}`
    }
    return url
}
}

```

A standard URL is made of several components, so to take them all in, the `Url` class constructor is inevitably big. Invoking such a constructor can be a challenge, as we must keep track of the argument position to know which component of the URL we are passing. Look at the following example to get an idea of this:

```
return new Url('https', null, null, 'example.com', null, null, null, null)
```

This is the perfect situation for applying the Builder pattern we just learned about. Let's do that now. The plan is to create a `UrlBuilder` class, which has a setter method for each parameter (or set of related parameters) needed to instantiate the `Url` class. Finally, the builder is going to have a `build()` method to retrieve a new `Url` instance that's been created using all the parameters that have been set in the builder. So, let's implement the builder in a file called `urlBuilder.js`:

```

export class UrlBuilder {
    setProtocol(protocol) {
        this.protocol = protocol
        return this
    }

    setAuthentication(username, password) {
        this.username = username
        this.password = password
    }
}

```

```
        return this
    }

    setHostname(hostname) {
        this.hostname = hostname
        return this
    }

    setPort(port) {
        this.port = port
        return this
    }

    setPathname(pathname) {
        this.pathname = pathname
        return this
    }

    setSearch(search) {
        this.search = search
        return this
    }

    setHash(hash) {
        this.hash = hash
        return this
    }

    build() {
        return new Url(this.protocol, this.username, this.password,
            this.hostname, this.port, this.pathname, this.search,
            this.hash)
    }
}
```

This should be straightforward. Just note the way we coupled together the username and password parameters into a single `setAuthentication()` method. This clearly conveys the fact that if we want to specify any authentication information in the `Url`, we have to provide both username and password.

Now, we are ready to try our `UrlBuilder` and witness its benefits over using the `Url` class directly. We can do that in a file called `index.js`:

```
import { UrlBuilder } from './urlBuilder.js'

const url = new UrlBuilder()
  .setProtocol('https')
  .setAuthentication('user', 'pass')
  .setHostname('example.com')
  .build()

console.log(url.toString())
```

As we can see, the readability of the code has improved dramatically. Each setter method clearly gives us a hint of which parameter we are setting, and on top of that, they provide some guidance on how those parameters must be set (for example, username and password must be set together).



Note that this implementation is very basic and it's missing some important features. For instance, the search functionality could be more user-friendly by allowing an array of key-value pairs (or a map or object) instead of just a plain string. This would make it easier to escape special characters automatically, so the user wouldn't need to worry about it.



The Builder pattern can also be implemented directly into the target class. For example, we could have refactored the `Url` class by adding an empty constructor (and, therefore, no validation at the object's creation time) and the setter methods for the various components, rather than creating a separate `UrlBuilder` class. However, this approach has a major flaw. Using a builder that is separate from the target class has the advantage of always producing instances that are guaranteed to be in a consistent state. For example, we could easily add some validation in the `UrlBuilder.build()` method to make sure that the protocol and hostname have been set. At this point, every `Url` object returned by `UrlBuilder.build()` is guaranteed to be valid and in a consistent state; calling `toString()` on such objects will always return a valid URL. The same cannot be said if we implemented the Builder pattern on the `Url` class directly. In fact, in this case, if we invoke `toString()` while we are still setting the various URL components, its return value may not be valid.

In the wild

The Builder pattern is quite common in Node.js and JavaScript as it provides a very elegant solution to the problem of creating complex objects or invoking complex functions. One perfect example is creating new HTTP(S) client requests with the `request()` API from the `http` and `https` built-in modules.

If we look at its documentation (available at nodejsdp.link/docs-http-request), we can immediately see it accepts a large number of options, which is the usual sign that the Builder pattern can potentially provide a better interface. In fact, a popular HTTP(S) request wrapper, *superagent* (nodejsdp.link/superagent), aims to simplify the creation of new requests by implementing the Builder pattern, thus providing a fluent interface to create new requests step by step. See the following code fragment for an example:

```
import superagent from 'superagent' // v10.1.1

superagent
  .post('https://example.com/api/person')
  .send({ name: 'John Doe', role: 'user' })
  .set('accept', 'json')
  .then((response) => {
    // deal with the response
  })
```

From the previous code, we can note that this is an unusual builder; in fact, we don't have a `build()` or `invoke()` method (or any other method with a similar purpose). What triggers the request instead is an invocation to the `then()` method. It's interesting to note that the *superagent* request object is not a promise but rather a custom *thenable*, where the `then()` method triggers the execution of the request we have built with the builder object. Because JavaScript treats any *thenable* like a promise, the same behavior applies when using `await` instead of explicitly calling `.then()`. For example, `await superagent.post(...)` will still trigger the request; in fact, behind the scenes, the `await` keyword simply calls the object's `.then()` method. This clever design allows the library to provide a fluent, builder-style API that integrates seamlessly with both promise chaining and modern `async/await` syntax.



We already discussed *thenables* in Chapter 5, *Asynchronous Control Flow Patterns with Promises and Async/Await*.

If you take a look at the library's code, you will see the Builder pattern in action in the `Request` class (nodejsdp.link/superagent-src-builder).

Another common use case for the Builder pattern is when you want to provide a JavaScript interface to a complex configuration file or a domain-specific language. A great example of this is an Object-Relational Mapper (ORM) library. ORMs let you interact with databases using JavaScript objects instead of writing raw SQL queries. Many of them include a SQL query builder, often implemented using the Builder pattern, to help you construct queries through a fluent and composable API. We have already mentioned *Knex* as a good example of the Factory pattern. *Knex* also offers a query builder that leverages the Builder pattern ([docs: nodejsdp.link/knex-query-builder](https://docs.nodejsdp.link/knex-query-builder)).

To prove that this is a very common use case, let's also look at another famous SQL ORM library: Drizzle (nodejsdp.link/drizzle). With Drizzle (v0.38.2), given a database connection (db) and two schemas (posts and comments) representing two database tables, we can construct and execute a SQL query as follows:

```
await db
  .select()
  .from(posts)
  .leftJoin(comments, eq(posts.id, comments.post_id))
  .where(eq(posts.id, 10))
```

This will effectively build and execute the following SQL statement:

```
SELECT *
FROM posts
LEFT JOIN comments
  ON posts.id = comments.post_id
WHERE posts.id = 10
```

Note that also in this case, the Builder pattern leverages a custom *thenable*, so the construction only happens when we call `await` (which implicitly calls `.then()` on the builder object).

Perhaps a slightly more traditional example of using the Builder pattern in the wild is parsing CLI arguments. The popular library `commander.js` (nodejsdp.link/commander) uses another variation of the Builder pattern to let you define how to parse CLI arguments and various other configuration options that are typical of CLI applications. Let's see an example, straight from the docs:

```
// cli.js
import { Command } from 'commander' v14.0.0

const program = new Command()
program
  .name('string-util')
  .description('CLI to some JavaScript string utilities')
  .version('0.8.0')
  .command('split')
  .description('Split a string into substrings and display as an array')
  .argument('<string>', 'string to split')
  .option('--first', 'display just the first substring')
  .option('-s, --separator <char>', 'separator character', ',')
  .action((str, options) => {
    const limit = options.first ? 1 : undefined
    console.log(str.split(options.separator, limit))
  })
  .parse()
```

This example implements a complete CLI utility that allows for string splitting in the terminal. For example, you can call it with:

```
node cli.js --separator "," "Hello, World"
```

And you will see the following result in the terminal:

```
[ 'Hello', ' World' ]
```

You should be able to appreciate how the Builder pattern is used here to specify all the details of this program: which options are supported and which positional arguments, but also the version, the description, and the specific function that needs to be executed for the `split` subcommand. The construction is finalized when the `parse()` method is called. The peculiarity of this implementation is that `parse()` does not return a new object, but instead, it starts the parsing of the arguments, which are then stored internally in the program instance and made accessible through the `program` object. The construction also evaluates which command needs to be executed, triggering the appropriate action.

This concludes our exploration of the Builder pattern. Next, we'll look at the Revealing Constructor pattern.

Revealing Constructor

The Revealing Constructor pattern is one of those patterns that you won't find in the GoF book, since it originated directly from JavaScript and the Node.js community. It solves a very tricky problem, which is: How can we “reveal” some private functionality of an object only at the moment of the object's creation? This is particularly useful when we want to allow an object's internals to be manipulated only during its creation phase. This allows for a few interesting scenarios, such as:

- Creating objects that can be modified only at creation time (i.e., it becomes immutable afterward)
- Creating objects whose custom behavior can be defined only at creation time
- Creating objects that can be initialized only once at creation time

These are just a few possibilities enabled by the Revealing Constructor pattern. But to better understand all the possible use cases, let's see what the pattern is about by looking at the following code fragment:

```
//           (1)           (2)           (3)
const object = new SomeClass(function executor(revealedMembers) {
  // manipulation code ...
})
```

As we can see from the previous code, the Revealing Constructor pattern is made of three fundamental elements: a **constructor** (1) that takes a function as input (the **executor** (2)), which is invoked at creation time and receives a subset of the object's internals as input (**revealedMembers** (3)).

For the pattern to work, the revealed functionality must otherwise not be accessible to the users of the object once it is created. This can be achieved with one of the encapsulation techniques we've mentioned in the previous section regarding the Factory pattern.



Domenic Denicola was the first to identify and name the pattern in one of his blog posts, which can be found at nodejsdp.link/domenic-revealing-constructor.

Now, let's look at a couple of examples to better understand how the Revealing Constructor pattern works.

Building an immutable buffer

Immutable objects and data structures have many excellent properties that make them ideal to use in countless situations in place of their mutable (or changeable) counterparts. Immutable refers to the property of an object by which its data or state becomes unmodifiable once it's been created.

With immutable objects, we don't need to create **defensive copies** before passing them around to other libraries or functions. We simply have a strong guarantee, by definition, that they won't be modified, even when they are passed to code that we don't know or control.

Modifying an immutable object can only be done by creating a new copy, which can make the code more maintainable and easier to reason about. We do this to make it easier to keep track of state changes.

Another common use case for immutable objects is efficient change detection. Since every change requires a copy, and if we assume that every copy corresponds to a modification, then detecting a change is as simple as using the strict equality operator (or triple equal, `===`). This technique is used extensively in frontend programming to efficiently detect if the UI needs refreshing.

In this context, let's now create a simple immutable version of the Node.js Buffer component (nodejsdp.link/docs-buffer) using the Revealing Constructor pattern. The pattern allows us to manipulate an immutable buffer only at creation time.

Let's implement our immutable buffer in a new file called `immutableBuffer.js`, as follows:

```
const MODIFIER_NAMES = ['swap', 'write', 'fill']

export class ImmutableBuffer {
  constructor(size, executor) {
    const buffer = Buffer.alloc(size) // 1
    const modifiers = {} // 2
    for (const prop in buffer) { // 3
      if (typeof buffer[prop] !== 'function') {
        continue
      }

      if (MODIFIER_NAMES.some(m => prop.startsWith(m))) { // 4
        modifiers[prop] = buffer[prop].bind(buffer)
      } else {
```

```

        this[prop] = buffer[prop].bind(buffer) // 5
    }
}

    executor(modifiers) // 6
}
}

```

Let's now see how our new `ImmutableBuffer` class works:

1. First, we allocate a new Node.js buffer (`buffer`) of the size specified in the `size` constructor argument.
2. Then, we create an object literal (`modifiers`) to hold all the methods that can mutate the buffer.
3. After that, we iterate over all the properties (own and inherited) of our internal buffer, making sure to skip all those that are not functions.
4. Next, we try to identify whether the current `prop` is a method that allows us to modify the buffer. We do that by trying to match its name with one of the strings in the `MODIFIER_NAMES` array. If we have such a method, we bind it to the buffer instance, and then we add it to the `modifiers` object.
5. If our method is not a modifier method, then we add it directly to the current instance (`this`).
6. Finally, we invoke the `executor` function received as input in the constructor and pass the `modifiers` object as an argument, which will allow `executor` to mutate our internal buffer.

In practice, our `ImmutableBuffer` is acting as a **proxy** between its consumers and the internal buffer object. Some of the methods of the buffer instance are exposed directly through the `ImmutableBuffer` interface (the read-only methods), while others are provided to the `executor` function (the modifier methods).

We will analyze the Proxy pattern in more detail in *Chapter 8, Structural Design Patterns*.



Please keep in mind that this is just a demonstration of the Revealing Constructor pattern, so the implementation of the immutable buffer is intentionally kept simple. For example, we are not exposing the size of the buffer or providing other means to initialize the buffer. We'll leave this to you as an exercise.

Now, let's write some code to demonstrate how to use our new `ImmutableBuffer` class. Let's create a new file, `index.js`, containing the following code:

```

import { ImmutableBuffer } from './immutableBuffer.js'

const hello = 'Hello!'
const immutable = new ImmutableBuffer(hello.length,
  ({ write }) => { // 1
    write(hello)
  }
)

```

```

    }
  )

  console.log(String.fromCharCode(immutable.readInt8(0))) // 2

  // the following line will throw
  // "TypeError: immutable.write is not a function"

  // immutable.write('Hello?') // 3

```

The first thing we can note from the previous code is how the executor function uses the `write()` function (which is part of the modifier methods) to write a string into the buffer (1). In a similar way, the executor function could've used `fill()`, `writeInt8()`, `swap16()`, or any other method exposed in the `modifiers` object.

The code we've just seen also demonstrates how the new `ImmutableBuffer` instance exposes only the methods that don't mutate the buffer, such as `readInt8()` (2), while it doesn't provide any method to change the content of the buffer (3).

In the wild

The Revealing Constructor pattern offers very strong guarantees, and for this reason, it's mainly used in contexts where we need to provide foolproof encapsulation. A perfect application of the pattern would be in components used by hundreds of thousands of developers that have to provide unopinionated interfaces and strict encapsulation. However, we can also use the pattern in our projects to improve reliability and simplify code sharing with other people and teams (since it can make an object safer to use by third parties).

A popular application of the Revealing Constructor pattern is in the JavaScript `Promise` class. Some of you may have already noticed it. When we create a new `Promise` from scratch, its constructor accepts as input an executor function that will receive the `resolve()` and `reject()` functions used to mutate the internal state of the `Promise`. Let's provide a reminder of what this looks like:

```

return new Promise((resolve, reject) => {
  // ...
})

```

Once created, the `Promise` state cannot be altered by any other means. All we can do is receive its fulfillment value or rejection reason using the methods we already learned about in *Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await*.

Another example, yet again from Domenic Denicola, is an event emitter implementation where the behavior of the event publisher is defined at construction time through a revealing constructor. The idea is that once the emitter is constructed, it's not possible to change the logic and introduce new events. If you are curious, you can check out the implementation: nodejsdp.link/revealing-ee.

Singleton

Now, we are going to spend a few words on a pattern that is among the most used in object-oriented programming, which is the **Singleton** pattern. As we will see, Singleton is one of those patterns that has a trivial implementation in Node.js that's almost not worth discussing. However, there are a few caveats and limitations that every good Node.js developer must know.

The purpose of the Singleton pattern is to enforce the presence of only one instance of a class and centralize its access. There are a few reasons for using a single instance across all the components of an application:

- For sharing stateful information
- For optimizing resource usage
- To synchronize access to a resource

As you can imagine, those are quite common scenarios. Take, for example, a typical Database class, which provides access to a database:

```
// Database.js
export class Database {
  constructor (dbName, connectionDetails) {
    // ...
  }
  // ...
}
```

Typical implementations of such a class usually keep a pool of database connections, so it doesn't make sense to create a new Database instance for each request. Plus, a Database instance may store some stateful information, such as the list of pending transactions. So, our Database class meets two criteria for justifying the Singleton pattern. Therefore, what we usually want is to configure and instantiate a single Database instance at the start of our application and let every component use that single shared Database instance.

A lot of people new to Node.js get confused about how to implement the Singleton pattern correctly; however, the answer is easier than what we might think. Simply exporting an instance from a module is already enough to obtain something very similar to the Singleton pattern. Consider, for example, the following lines of code:

```
// dbInstance.js
import { Database } from './Database.js'

export const dbInstance = new Database('my-app-db', {
  url: 'localhost:5432',
  username: 'user',
  password: 'password'
})
```


By simply exporting a new instance of our Database class, we can already assume that within the current package (which can easily be the entire code of our application), we are going to have only one shared reference to the `dbInstance` value. This is possible because, as we know from *Chapter 2, The Module System*, Node.js will cache the module, making sure not to execute its code at every import.

For example, we can easily obtain a shared instance of the `dbInstance` module, which we defined earlier, with the following line of code:

```
import { dbInstance } from './dbInstance.js'
```

But there is a caveat. The module is cached using its full path as the lookup key, so it is only guaranteed to be a singleton within the current package. In fact, each package may have its own set of private dependencies inside its `node_modules` directory, which might result in multiple instances of the same package and, therefore, of the same module, resulting in our singleton not really being unique anymore! This is, of course, a rare scenario, but it's important to understand what its consequences are.

Consider, for example, the case in which the `Database.js` and `dbInstance.js` files that we saw earlier are wrapped into a package named `mydb`. The following lines of code would be in its `package.json` file:

```
{
  "name": "mydb",
  "version": "2.0.0",
  "type": "module",
  "main": "dbInstance.js"
}
```

Next, consider two packages (`package-a` and `package-b`), both of which have a single file called `index.js` containing the following code:

```
import { dbInstance } from 'mydb'

export function getDbInstance () {
  return dbInstance
}
```

Both `package-a` and `package-b` have a dependency on the `mydb` package. However, `package-a` depends on version `1.0.0` of the `mydb` package, while `package-b` depends on version `2.0.0` of the same package (which, for our example, has a mostly compatible implementation, but just a different version specified in its `package.json` file).

Given the structure we just described, we would end up with the following package dependency tree:

```
app/
|-- node_modules
|   |-- package-a
|   |   |-- node_modules
|   |       |-- mydb
```

```
`-- package-b
  |-- node_modules
    |-- mydb
```

We end up with a directory structure like the one here because `package-a` and `package-b` require two different versions of the `mydb` module (for example, `1.0.0` versus `2.0.0`). In this case, a typical package manager such as `npm` or `yarn` would not “hoist” the dependency to the top `node_modules` directory, but it will instead install a private copy of each package.

With the directory structure we just saw, both `package-a` and `package-b` have a dependency on the `mydb` package; in turn, the `app` package, which is our root package, depends on both `package-a` and `package-b`.

The scenario we just described will break the assumption about the uniqueness of the database instance. In fact, consider the following file (`index.js`) located in the root folder of the `app` package:

```
import { getDbInstance as getDbFromA } from 'package-a'
import { getDbInstance as getDbFromB } from 'package-b'

const isSame = getDbFromA() === getDbFromB()
console.log('Is the db instance in package-a the same ' +
  `as package-b? ${isSame ? 'YES' : 'NO'}`)
```

If you run the previous file, you will notice that the answer to *Is the db instance in package-a the same as package-b?* is NO. In fact, `package-a` and `package-b` will actually load two different instances of the `dbInstance` object because the `mydb` module will resolve to a different directory, depending on the package it is required from. This clearly breaks the assumptions of the Singleton pattern.



If, instead, both `package-a` and `package-b` required two versions of the `mydb` package compatible with each other, for example, `^2.0.1` and `^2.0.7`, then the package manager would install the `mydb` package into the top-level `node_modules` directory (a practice known as **dependency hoisting**), effectively sharing the same instance with `package-a`, `package-b`, and the root package.

At this point, we can easily say that the Singleton pattern, as described in the literature, does not exist in Node.js, unless we use a real *global variable* to store it, such as the following:

```
global.dbInstance = new Database('my-app-db', { /* ... */ })
```

This guarantees that the instance is the only one shared across the entire application and not just the same package. However, please consider that most of the time, we don’t really need a *pure* singleton. In fact, we usually create and import singletons within the main package of an application or, at worst, in a subcomponent of the application that has been modularized into a dependency.



If you are creating a package that is going to be used by third parties, try to keep it stateless to avoid the issues we've discussed in this section.

Throughout this book, for simplicity, we will use the term singleton to describe a class instance or a stateful object exported by a module, even if this doesn't represent a real singleton in the strict definition of the term.



Note that this implementation isn't a *strict* implementation of the Singleton pattern. While exporting a single instance (`dbInstance`) from a module gives you a shared reference throughout your application, it doesn't strictly enforce the Singleton pattern as originally defined in the GoF book. In fact, it's still possible to bypass this mechanism by importing the original `Database` class and creating a new instance manually. Depending on your use case, this level of flexibility might be perfectly fine, or it might be a loophole you'd rather avoid. If you want to make things stricter, consider defining the `Database` class directly inside the module where you instantiate it and export only the singleton instance. That way, no other part of your codebase can easily access the constructor or create additional instances. It's also worth noting that, due to the dynamic nature of JavaScript, even hiding the class definition inside a module only makes it harder—not impossible—to create additional instances. JavaScript allows you to access an object's original constructor through the `constructor` property. This means that even if the `Database` class isn't exported, someone with access to the singleton instance could still call `new dbInstance.constructor()` to create a new object. This is a good example of how traditional design patterns, such as Singleton, often take on a more relaxed and flexible form in dynamic languages such as JavaScript. Absolute enforcement is difficult, and in many cases, unnecessary. What matters most is setting clear boundaries and following conventions consistently across your codebase.

Next, we are going to see the two main approaches for dealing with dependencies between modules, one based on the Singleton pattern and the other leveraging the Dependency Injection pattern.

Wiring modules

Every application is the result of the aggregation of several components, and as the application grows, the way we connect these components becomes a win-or-lose factor for the maintainability and success of the project.

When component A needs component B to fulfill a given functionality, we say that “A is **dependent** on B” or, conversely, that “B is a **dependency** of A.” To appreciate this concept, let's present an example.

Let's say we want to write an API for a blogging system that uses a database to store its data. We can have a generic module implementing a database connection (`db.js`) and a blog module that exposes the main functionality to create and retrieve blog posts from the database (`blog.js`).

The following figure illustrates the relationship between the database module and the blog module:

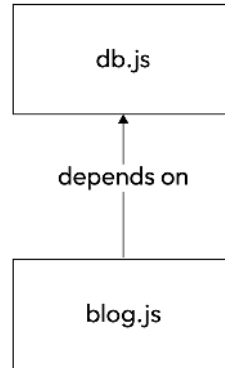


Figure 7.1: Dependency graph between the blog module and the database module

In this section, we are going to see how we can model this dependency using two different approaches, one based on the Singleton pattern and the other using the Dependency Injection pattern.

Singleton dependencies

The simplest way to wire two modules together is by leveraging the Node.js module system. Stateful dependencies wired in this way are de facto singletons, as we discussed in the previous section.

To see how this works in practice, we are going to implement the simple blogging application that we described earlier using a singleton instance for the database connection. Let's see a possible implementation of this approach in the file `db.js`:

```
import { join } from 'node:path'
import sqlite3 from 'sqlite3' // v5.1.7
import { open } from 'sqlite' // v5.1.1

export const db = await open({
  filename: join(import.meta.dirname, 'data.sqlite'),
  driver: sqlite3.Database,
})
```

In the previous code, we are using SQLite (nodejsdp.link/sqlite) as a database to store our posts. To interact with SQLite, we are using the package `sqlite3` (nodejsdp.link/sqlite3) from npm. SQLite is a database system that keeps all the data in a single local file. In our database module, we decided to use a file called `data.sqlite` saved in the same folder as the module.

The preceding code creates a new instance of the database pointing to our data file and exports the database connection object as a singleton with the name `db`.

Now, let's see how we can implement the `blog.js` module:

```
import { db } from './db.js'

export class Blog {
  initialize() {
    const initQuery = `CREATE TABLE IF NOT EXISTS posts (
      id TEXT PRIMARY KEY,
      title TEXT NOT NULL,
      content TEXT,
      created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );`

    return db.run(initQuery)
  }

  createPost(id, title, content, createdAt) {
    return db.run(
      'INSERT INTO posts VALUES (?, ?, ?, ?)',
      id,
      title,
      content,
      createdAt
    )
  }

  getAllPosts() {
    return db.all('SELECT * FROM posts ORDER BY created_at DESC')
  }
}
```

The `blog.js` module exports a class called `Blog` containing three methods:

- `initialize()`: This creates the `posts` table if it doesn't exist. The table will be used to store the blog post's data.
- `createPost()`: This takes all the necessary parameters needed to create a post. It will execute an `INSERT` statement to add the new post to the database.
- `getAllPosts()`: This retrieves all the posts available in the database and returns them as an array.

Now, let's create a module (`index.js`) to try out the functionality of the `blog` module we just created:

```
import { Blog } from './blog.js'

const blog = new Blog()
await blog.initialize()
const posts = await blog.getAllPosts()
if (posts.length === 0) {
  console.log(
    'No post available. Run `node import-posts.js` ' +
    'to load some sample posts'
  )
}

for (const post of posts) {
  console.log(post.title)
  console.log('-'.repeat(post.title.length))
  console.log(`Published on ${new Date(post.created_at).toISOString()}`)
  console.log(post.content)
}
```

The preceding module is very simple. We retrieve the array with all the posts using `blog.getAllPosts()`, and then we loop over it and display the data for every single post, giving it a bit of formatting.

You can use the provided `import-posts.js` module to load some sample posts into the database before running `index.js`. You can find `import-posts.js` in the code repository of this book, along with the rest of the files for this example.



As a fun exercise, you could try to modify the `index.js` module to generate HTML files: one for the blog index and then a dedicated file for each blog post. This way, you would build your own minimalistic static website generator!

As we can see from the preceding code, we can implement a very simple command-line blog management system by leveraging the Singleton pattern to pass the `db` instance around. Most of the time, this is how we manage stateful dependencies in our application; however, there are situations in which this may not be enough.

Using a singleton, as we have done in the previous example, is certainly the most simple, immediate, and readable solution to pass stateful dependencies around. However, what happens if we want to mock our database during our tests? What can we do if we want to let the user of the blogging CLI or the blogging API select another database backend, instead of the standard SQLite backend that we provide by default? For these use cases, a singleton can be an obstacle to implementing a properly structured solution.

We could introduce `if` statements in our `db.js` module to pick different implementations based on some environment condition or some configuration. Alternatively, we could fiddle with the Node.js module system to intercept the import of the database file and replace it with something else (mocking imports). But, as you can imagine, these solutions are far from elegant.

In the next section, we will learn about another strategy for wiring modules, which can be the ideal solution to some of the issues we discussed here.

Dependency Injection

The Node.js module system and the Singleton pattern can serve as great tools for organizing and wiring together the components of an application. However, these do not always guarantee success. If, on the one hand, they are simple to use and very practical, then on the other, they might introduce a tighter *coupling* between components.

In the previous example, we can see that the `blog.js` module is *tightly coupled* with the `db.js` module. In fact, our `blog.js` module cannot work without the `database.js` module by design, nor can it use a different database module if necessary. We can easily fix this tight coupling between the two modules by leveraging the **Dependency Injection (DI) pattern**.

DI is a very simple pattern in which the dependencies of a component are *provided as input* by an external entity, often referred to as the **injector**.

The injector initializes the different components and ties their dependencies together. It can be a simple initialization script or a more sophisticated *global container* that maps all the dependencies and centralizes the wiring of all the modules of the system. The main advantage of this approach is improved decoupling, especially for modules depending on stateful instances (for example, a database connection). Using DI, instead of being hardcoded into the module, each dependency is received from the outside. This means that the dependent module can be configured to use any compatible dependency, and therefore, the module itself can be reused in different contexts with minimal effort.

The following diagram illustrates this idea:

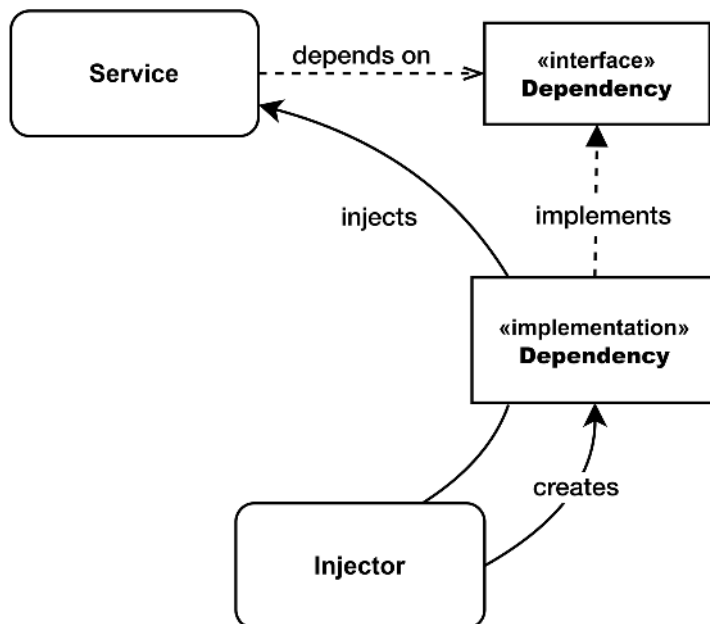


Figure 7.2: DI schematic

In Figure 7.2, we can see that a generic service expects a dependency with a predetermined interface. It's the responsibility of the **injector** to retrieve or create an actual concrete instance that implements such an interface and passes it (or “injects it”) into the service. In other words, the injector has the goal of providing an instance that fulfills the dependency for the service.

To demonstrate this pattern in practice, let's refactor the simple blogging system that we built in the previous section by using DI to wire its modules. Let's start by refactoring the `blog.js` module:

```
export class Blog {
  constructor(db) {
    this.db = db
  }

  initialize() {
    const initQuery = `CREATE TABLE IF NOT EXISTS posts (
      id TEXT PRIMARY KEY,
      title TEXT NOT NULL,
      content TEXT,
      created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    );`

    return this.db.run(initQuery)
  }
}
```



```
}

createPost(id, title, content, createdAt) {
  return this.db.run(
    'INSERT INTO posts VALUES (?, ?, ?, ?)',
    id,
    title,
    content,
    createdAt
  )
}

getAllPosts() {
  return this.db.all('SELECT * FROM posts ORDER BY created_at DESC')
}
}
```

If you compare the new version with the previous one, they are almost identical. There are only two small but important differences:

- We are not importing the database module anymore
- The Blog class constructor takes db as an argument

The new constructor argument db is the expected dependency that needs to be provided at runtime by the client component of the Blog class. This client component is going to be the injector of the dependency. Since JavaScript doesn't have any way to represent abstract interfaces, the provided dependency is expected to implement the db.run() and db.all() methods. This is called duck typing, as mentioned earlier in this book.

Let's now rewrite our db.js module. The goal here is to get rid of the Singleton pattern and to come up with an implementation that is more reusable and configurable:

```
import { open } from 'sqlite'
import sqlite3 from 'sqlite3'

export function createDb(filename) {
  return open({
    filename,
    driver: sqlite3.Database,
  })
}
```

This new implementation of the db module provides a factory function called `createDb()`, which allows us to create new instances of the database at runtime. It also allows us to pass the path to the database file at creation time so that we can create independent instances that can write to different database files if we need to.

At this point, we have almost all the building blocks in place; we are only missing the injector. We will give an example of the injector by reimplementing the `index.js` module:

```
import { join } from 'node:path'
import { Blog } from './blog.js'
import { createDb } from './db.js'

const db = await createDb( // 1
  join(import.meta.dirname, 'data.sqlite'))
const blog = new Blog(db) // 2
await blog.initialize()
const posts = await blog.getAllPosts()
if (posts.length === 0) {
  console.log(
    'No post available. Run `node import-posts.js` +
    ' to load some sample posts'
  )
}

for (const post of posts) {
  console.log(post.title)
  console.log('-'.repeat(post.title.length))
  console.log(`Published on ${new Date(post.created_at).toISOString()}`)
  console.log(post.content)
}
```

This code is quite similar to the previous implementation, except for two important changes (highlighted in the preceding code):

1. We create the database dependency (db) using the factory function `createDb()`.
2. We explicitly “inject” the database instance when we instantiate the `Blog` class.

In this implementation of our blogging system, the `blog.js` module is totally decoupled from the actual database implementation, making it more composable and easy to test in isolation.



We saw how to inject dependencies as constructor arguments (**constructor injection**), but dependencies can also be passed when invoking a function or method (**function injection**) or injected explicitly by assigning the relevant properties of an object (**property injection**).

Unfortunately, the advantages in terms of decoupling and reusability offered by the DI pattern come with a price to pay. In general, the inability to resolve a dependency at *coding time* makes it more difficult to understand the relationship between the various components of a system. This is especially true in large applications where we might have a significant number of services with a complex dependency graph.

Also, if we look at the way we instantiated our database dependency in our preceding example script, we can see that we had to make sure that the database instance was created before we could invoke any function from our `Blog` instance. This means that, when used in its raw form, DI forces us to build the dependency graph of the entire application by hand, making sure that we do it in the right order. This can become unmanageable when the number of modules to wire becomes too high.



Another pattern, called **Inversion of Control**, allows us to shift the responsibility of wiring the modules of an application to a third-party entity. This entity can be a **service locator** (a simple component used to retrieve a dependency, for example, `serviceLocator.get('db')`) or a **DI container** (a system that injects the dependencies into a component based on some metadata specified in the code itself or in a configuration file). You can find more about these components in Martin Fowler's blog at nodejsdp.link/ioc-containers. Even though these techniques derail a bit from the Node.js way of doing things, some of them have recently gained some popularity. Check out `inversify` (nodejsdp.link/inversify) and `awilix` (nodejsdp.link/awilix) to find out more.

Summary

In this chapter, you were introduced to a set of traditional design patterns concerning the creation of objects. Some of those patterns are so basic, and yet essential at the same time, that you have probably already used them in one way or another.

Patterns such as `Factory` and `Singleton` are, for example, two of the most ubiquitous in object-oriented programming in general. However, in JavaScript, their implementation and significance are very different from what was proposed by the *GoF* book. For example, `Factory` becomes a very versatile pattern that works in perfect harmony with the hybrid nature of the JavaScript language—that is, half object-oriented and half functional. On the other hand, `Singleton` becomes so trivial to implement that it's almost a non-pattern, but it carries a set of caveats that you should have learned to consider.

Among the patterns you've learned about in this chapter, the `Builder` pattern may seem to be the one that has retained most of its traditional object-oriented form. However, we've shown you that it can also be used to invoke complex functions and not just to build objects.

The Revealing Constructor pattern, on the other hand, deserves a category of its own. Born from necessities arising from the JavaScript language itself, it provides an elegant solution to the problem of “revealing” certain private object properties at construction time only. It provides strong guarantees in a language that is relaxed by nature.

Finally, you learned about the two main techniques for wiring components together: Singleton and DI. We’ve seen how the first is the simplest and most practical approach, while the second is more powerful but also potentially more complex to implement.

As we already mentioned, this was the first of a series of three chapters entirely dedicated to traditional design patterns. In these chapters, we will try to teach the right balance between creativity and rigor. We want to show not only that there are patterns that can be reused to improve our code, but also that their implementation is not the most important detail; in fact, it can vary a lot, or even overlap with other patterns. What really matters, however, is the blueprint, the guidelines, and the idea at the base of each pattern. This is the real reusable information that we can take advantage of to design better Node.js applications in a fun way.

In the next chapter, you will learn about another category of traditional design patterns, called **structural** patterns. As the name suggests, these patterns are aimed at improving the way we combine objects together to build more complex, yet flexible and reusable structures.

Exercises

- **7.1 Console color factory:** Create a class called `ColorConsole` that has just one empty method called `log()`. Then, create three subclasses: `RedConsole`, `BlueConsole`, and `GreenConsole`. The `log()` method of every `ColorConsole` subclass will accept a string as input and will print that string to the console using the color that gives the name to the class. Then, create a factory function that takes color as input, such as `'red'`, and returns the related `ColorConsole` subclass. Finally, write a small command-line script to try the new console color factory. You can use this [Stack Overflow answer](https://stackoverflow.com/questions/44203322/console-colors) as a reference for using colors in the console: `nodejsdp.link/console-colors`.
- **7.2 Request builder:** Create your own `Builder` class around the built-in `http.request()` function. The builder must be able to provide at least basic facilities to specify the HTTP method, the URL, the query component of the URL, the header parameters, and the eventual body data to be sent. To send the request, provide an `invoke()` method that returns a `Promise` for the invocation. You can find the docs for `http.request()` at the following URL: `nodejsdp.link/docs-http-request`.

- **7.3 A tamper-free queue:** Create a Queue class that has only one publicly accessible method called `dequeue()`. Such a method returns a Promise that resolves with a new element extracted from an internal queue data structure. If the queue is empty, then the Promise will resolve when a new item is added. The Queue class must also have a revealing constructor that provides a function called `enqueue()` to the executor that pushes a new element to the end of the internal queue. The `enqueue()` function can be invoked asynchronously, and it must also take care of “unblocking” any eventual Promise returned by the `dequeue()` method. To try out the Queue class, you could build a small HTTP server into the executor function. Such a server would receive messages or tasks from a client and would push them into the queue. A loop would then consume all those messages using the `dequeue()` method.

8

Structural Design Patterns

In this chapter, we'll dive into some of the most widely used structural design patterns and see how they apply in the world of Node.js. Structural patterns help us define clear relationships between components, enabling flexible and efficient architectures.

We'll focus on three key patterns:

- **Proxy:** Control access to an object by standing in for it
- **Decorator:** Dynamically extend or modify an object's behavior
- **Adapter:** Bridge incompatible interfaces to enable smooth collaboration

Along the way, we'll also touch on **reactive programming (RP)** and explore **Level**, a fast, lightweight key-value store that fits well in the Node.js ecosystem. You'll learn how to use it and even build your own plugin.

By the end of the chapter, you'll not only understand how these structural patterns work but also know when and how to apply them effectively in real-world Node.js applications.

Proxy

A **proxy** is an object that controls access to another object, called the **subject**. The proxy and the subject have an identical interface, and this allows us to swap one for the other transparently; in fact, the alternative name for this pattern is **surrogate**.

A proxy intercepts all or some of the operations that are meant to be executed on the subject, augmenting or complementing their behavior. *Figure 8.1* shows a schematic representation of this pattern:

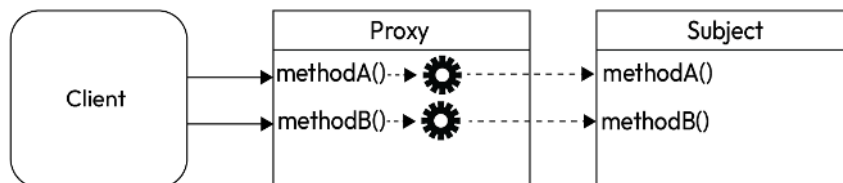


Figure 8.1: Proxy pattern schematic

Figure 8.1 shows us how the proxy and the subject have the same interface, and how this is transparent to the client, who can use one or the other interchangeably. The proxy forwards each operation to the subject, enhancing its behavior with additional preprocessing or postprocessing.



It's important to observe that we are not talking about proxying between classes; the Proxy pattern involves wrapping an actual instance of the subject, thus preserving its internal state.

A proxy can be useful in several circumstances, for example:

- **Data validation:** The proxy validates the input before forwarding it to the subject
- **Security:** The proxy verifies that the client is authorized to perform the operation, and it passes the request to the subject only if the outcome of the check is positive
- **Caching:** The proxy keeps an internal cache so that the proxied operations are executed on the subject only if the data is not yet present in the cache
- **Lazy initialization:** If creating the subject is expensive, the proxy can delay it until it's really necessary
- **Observability:** The proxy intercepts the method invocations and the relative parameters, re-coding them as they happen
- **Remote objects:** The proxy can take a remote object and make it appear local

There are more Proxy pattern applications, but these should give us an idea of its purpose.

Techniques for implementing proxies

When *proxying* an object, we can decide to intercept all of its methods or only some of them, while delegating the rest directly to the subject. There are several ways in which this can be achieved, and in this section, we will present some of them.

We will be working on a simple example, a `StackCalculator` class that looks like this:

```
class StackCalculator {
    constructor() {
        this.stack = []
    }

    putValue(value) {
        this.stack.push(value)
    }

    getValue() {
        return this.stack.pop()
    }
}
```

```
peekValue() {  
    return this.stack[this.stack.length - 1]  
}  
  
clear() {  
    this.stack = []  
}  
  
divide() {  
    const divisor = this.getValue()  
    const dividend = this.getValue()  
    const result = dividend / divisor  
    this.putValue(result)  
    return result  
}  
  
multiply() {  
    const multiplicand = this.getValue()  
    const multiplier = this.getValue()  
    const result = multiplier * multiplicand  
    this.putValue(result)  
    return result  
}  
}
```

This class implements a simplified version of a stack calculator. The idea of this calculator is to keep all operands (values) in a stack. When you perform an operation, for example a multiplication, the multiplicand and the multiplier are extracted from the stack and the result of the multiplication is pushed back into the stack. This is not too different from how the calculator application on your mobile phone is implemented.

Here's an example of how we might use `StackCalculator` to perform some multiplications and divisions:

```
const calculator = new StackCalculator()  
calculator.putValue(3)  
calculator.putValue(2)  
console.log(calculator.multiply()) // 3*2 = 6  
calculator.putValue(2)  
console.log(calculator.multiply()) // 6*2 = 12
```

There are also some utility methods, such as `peekValue()`, which allows us to peek at the value at the top of the stack (the last value inserted or the result of the last operation), and `clear()`, which allows us to reset the stack.

Fun fact: In JavaScript, when you perform a division by 0, you get back a *mysterious* value called *Infinity*. In many other programming languages, dividing by 0 is an illegal operation that results in the program panicking or throwing a runtime exception.

Our task in the next few sections will be to leverage the Proxy pattern to enhance a `StackCalculator` instance by providing a more conservative behavior for division by 0: rather than returning *Infinity*, we will throw an explicit error.

Object composition

Composition is a technique whereby an object is combined with another object for the purpose of extending or using its functionality. In the specific case of the Proxy pattern, a new object with the same interface as the subject is created, and a reference to the subject is stored internally in the proxy in the form of an instance variable or a closure variable. The subject can be injected from the client at creation time or created by the proxy itself.

The following example implements a safe calculator using object composition:

```
class SafeCalculator {
  constructor(calculator) {
    this.calculator = calculator
  }

  // proxied method
  divide() {
    // additional validation logic
    const divisor = this.calculator.peekValue()
    if (divisor === 0) {
      throw new Error('Division by 0')
    }
    // if valid delegates to the subject
    return this.calculator.divide()
  }

  // delegated methods
  putValue(value) {
    return this.calculator.putValue(value)
  }
  getValue() {
    return this.calculator.getValue()
  }
  peekValue() {
    return this.calculator.peekValue()
  }
}
```

```

    clear() {
        return this.calculator.clear()
    }
    multiply() {
        return this.calculator.multiply()
    }
}

const calculator = new StackCalculator()
const safeCalculator = new SafeCalculator(calculator)

calculator.putValue(3)
calculator.putValue(2)
console.log(calculator.multiply())    // 3*2 = 6

safeCalculator.putValue(2)
console.log(safeCalculator.multiply()) // 6*2 = 12

calculator.putValue(0)
console.log(calculator.divide())      // 12/0 = Infinity

safeCalculator.clear()
safeCalculator.putValue(4)
safeCalculator.putValue(0)
console.log(safeCalculator.divide())   // 4/0 -> Error

```

The `safeCalculator` object is a proxy for the original calculator instance. By invoking `multiply()` on `safeCalculator`, we will end up calling the same method on `calculator`. The same goes for `divide()`, but in this case, we can see that, if we try to divide by zero, we will get different outcomes depending on whether we perform the division on the subject or on the proxy.

To implement this proxy using composition, we had to intercept the methods that we were interested in manipulating (`divide()`), while simply delegating the rest of them to the subject (`putValue()`, `getValue()`, `peekValue()`, `clear()`, and `multiply()`).

Note that the calculator state (the values in the stack) is still maintained by the calculator instance; `safeCalculator` will only invoke methods on `calculator` to read or mutate the state as needed. In the `divide()` method, we need to access the state to check if the divisor is 0. We do this by using the `peekValue()` method directly in the internal calculator instance. Other than using `peekValue()`, the `safeCalculator` instance doesn't have a way to access all the other values in the stack.

An alternative implementation of the proxy presented in the preceding code fragment might just use an object literal and a factory function:

```
function createSafeCalculator(calculator) {
  return {
    // proxied method
    divide() {
      // additional validation logic
      const divisor = calculator.peekValue()
      if (divisor === 0) {
        throw new Error('Division by 0')
      }
      // if valid delegates to the subject
      return calculator.divide()
    },
    // delegated methods
    putValue(value) {
      return calculator.putValue(value)
    },
    getValue() {
      return calculator.getValue()
    },
    peekValue() {
      return calculator.peekValue()
    },
    clear() {
      return calculator.clear()
    },
    multiply() {
      return calculator.multiply()
    },
  }
}

const calculator = new StackCalculator()
const safeCalculator = createSafeCalculator(calculator)
// ...
```

This implementation is simpler and more concise than the class-based one, but, once again, it forces us to delegate all the methods to the subject explicitly.

Having to delegate many methods for complex classes can be very tedious and might make it harder to implement these techniques. One way to create a proxy that delegates most of its methods is to use a library that generates all the methods for us, such as `delegates` (nodejsdp.link/delegates).



`delegates` is a great library, and we would encourage you to read its source code to see how it implements method delegation. But at the same time, it is important to acknowledge that it's a very old library, it uses a few deprecated JavaScript features, and it hasn't been updated in a while, so you should avoid using it in production.

Another simple way to reduce code duplication could be to create the various methods on the proxy by doing a loop:

```
function createSafeCalculator(calculator) {
  const safeCalculator = {
    // proxied method
    divide() {
      // ...hidden for brevity
    },
  }

  // delegated methods
  for (const fn of [
    'putValue', 'getValue', 'peekValue', 'clear', 'multiply'
  ]) {
    safeCalculator[fn] = calculator[fn].bind(calculator)
  }

  return safeCalculator
}
```

A more modern and native alternative is to use the Proxy object, which we will discuss later in this chapter.

Object augmentation

Object augmentation (or **monkey patching**) is probably the simplest and most common way of proxying just a few methods of an object. It involves modifying the subject directly by replacing a method with its proxied implementation.

In the context of our calculator example, this could be done as follows:

```
function patchToSafeCalculator(calculator) {
  const divideOrig = calculator.divide
  calculator.divide = () => {
    // additional validation logic
  }
}
```

```
const divisor = calculator.peekValue()
if (divisor === 0) {
  throw new Error('Division by 0')
}
// if valid delegates to the subject
return divideOrig.apply(calculator)
}
return calculator
}

const calculator = new StackCalculator()
const safeCalculator = patchToSafeCalculator(calculator)
// ...
```

This technique is definitely convenient when we need to proxy only one or a few methods. Did you notice that we didn't have to reimplement the `multiply()` method and all the other delegated methods here?

Unfortunately, simplicity comes at the cost of having to mutate the subject object directly, which can be dangerous.



Mutations should be avoided at all costs when the subject is shared with other parts of the codebase. In fact, “monkey patching” the subject might create undesirable side effects that affect other components of our application. Use this technique only when the subject exists in a controlled context or in a private scope. If you want to appreciate why “monkey patching” is a dangerous practice, you could try to invoke a division by zero in the original calculator instance. If you do so, you will see that the original instance will now throw an error rather than returning `Infinity`. The original behavior has been altered, and this might have unexpected effects on other parts of the application.

In the next section, we will explore the built-in `Proxy` object, which is a powerful alternative for implementing the `Proxy` pattern and more.

The built-in `Proxy` object

The ES2015 specification introduced a native way to create powerful proxy objects.

We are talking about the ES2015 `Proxy` object, which consists of a `Proxy` constructor that accepts `target` and `handler` as arguments:

```
const proxy = new Proxy(target, handler)
```

Here, `target` represents the object on which the proxy is applied (the **subject** in our canonical definition), while `handler` is a special object that defines the behavior of the proxy.

The handler object contains a series of optional methods with predefined names called **trap methods** (for example, `apply`, `get`, `set`, and `has`) that are automatically called when the corresponding operations are performed on the proxy instance.

To better understand how this API works, let's see how we can use the Proxy object to implement our safe calculator proxy:

```
const safeCalculatorHandler = {
  get: (target, property) => {
    if (property === 'divide') {
      // proxied method
      return () => {
        // additional validation logic
        const divisor = target.peekValue()
        if (divisor === 0) {
          throw new Error('Division by 0')
        }
        // if valid delegates to the subject
        return target.divide()
      }
    }
    // delegated methods and properties
    return target[property]
  },
}

const calculator = new StackCalculator()
const safeCalculator = new Proxy(
  calculator,
  safeCalculatorHandler
)
// ...
```

In this implementation of the safe calculator proxy using the Proxy object, we adopted the `get` trap to intercept access to properties and methods of the original object, including calls to the `divide()` method. When access to `divide()` is intercepted, the proxy returns a modified version of the function that implements the additional logic to check for possible divisions by zero. Note that we can simply return all other methods and properties unchanged by using `target[property]`.

Finally, it is important to mention that the Proxy object inherits the prototype of the subject; therefore, running `safeCalculator instanceof StackCalculator` will return `true`.

With this example, it should be clear that the Proxy object allows us to avoid mutating the subject while giving us an easy way to proxy only the bits that we need to enhance, without having to explicitly delegate all the other properties and methods.

Additional capabilities and limitations of the Proxy object

The Proxy object is a feature deeply integrated into the JavaScript language itself, which enables developers to intercept and customize many operations that can be performed on objects. This characteristic opens new and interesting scenarios that were not easily achievable before, such as *meta-programming*, *operator overloading*, and *object virtualization*.

Let's see another example to clarify this concept:

```
const evenNumbers = new Proxy([], {
  get: (target, index) => index * 2,
  has: (target, number) => number % 2 === 0
})

console.log(2 in evenNumbers) // true
console.log(5 in evenNumbers) // false
console.log(evenNumbers[7])   // 14
```

In this example, we are creating a virtual array that contains all even numbers. It can be used as a regular array, which means we can access items in the array with the regular array syntax (for example, `evenNumbers[7]`) or check the existence of an element in the array with the `in` operator (for example, `2 in evenNumbers`). The array is considered *virtual* because we never store data in it.



It is very important to note that, while the previous code snippet is a very interesting example that aims to showcase some of the advanced capabilities of the Proxy object, it does not implement the Proxy pattern. This example allows us to see that, even though the Proxy object is commonly used to implement the Proxy pattern (hence the name), it can also be used to implement other patterns and use cases. As an example, we will see later in this chapter how to use the Proxy object to implement the Decorator pattern.

Looking at the implementation, this proxy uses an empty array as the target and then defines the `get` and `has` traps in the handler:

- The `get` trap intercepts access to the array elements, returning the even number for the given index
- The `has` trap instead intercepts the usage of the `in` operator and checks whether the given number is even or not

The Proxy object supports several other interesting traps, such as `set`, `delete`, and `construct`, and allows us to create proxies that can be revoked on demand, disabling all the traps and restoring the original behavior of the target object.

Analyzing all these features goes beyond the scope of this chapter; what is important here is understanding that the Proxy object provides a powerful foundation for implementing the Proxy design pattern.



If you are curious to discover all the capabilities and trap methods offered by the Proxy object, you can read more in the related MDN article at nodejsdp.link/mdn-proxy. Another good source is this detailed article from Google at nodejsdp.link/intro-proxy.

While the Proxy object is a powerful functionality of the JavaScript language, it suffers from a very important limitation: the Proxy object cannot be fully *transpiled* or *polyfilled*. This is because some of the Proxy object traps can be implemented only at the runtime level and cannot be simply rewritten in plain JavaScript. This is something to be aware of if you are working with very old browsers or old versions of Node.js that don't support the Proxy object directly.



Transpilation: Short for *transcompilation*. It is the action of compiling source code by translating it from one source programming language to another. In the case of JavaScript, this technique is used to convert a program using new capabilities of the language into an equivalent program that can also run on older runtimes that do not support these new capabilities.

Polyfill: Code that provides an implementation for a standard API in plain JavaScript and that can be imported in environments where this API is not available (generally older browsers or runtimes). `core-js` (nodejsdp.link/corejs) is one of the most complete polyfill libraries for JavaScript.

A comparison of the different proxying techniques

Composition can be considered a simple and *safe* way of creating a proxy because it leaves the subject untouched without mutating its original behavior. Its only drawback is that we must manually delegate all the methods, even if we want to proxy only one of them. Also, we might have to delegate access to the properties of the subject.

Object augmentation, on the other hand, modifies the subject, which might not always be ideal, but it does not suffer from the various inconveniences related to delegation. For this reason, between these two approaches, object augmentation is generally the preferred technique in all those circumstances in which modifying the subject is an option.

However, there is at least one scenario where composition becomes not only useful but necessary: when you need to control the initialization of the subject. This is especially important in cases where the object is expensive to create or might not be needed at all—what is commonly referred to as *lazy initialization*. With composition, you can defer the creation of the subject until one of its methods is actually called. This is a key distinction from the Proxy object approach, which requires an existing instance to wrap. Since Proxy operates on a fully constructed target, it doesn't allow this kind of lazy setup out of the box. Composition gives you full control over when and how the underlying object comes to life, making it the ideal solution in these situations.

Finally, the Proxy object is the go-to approach if you need to intercept function calls or have different types of access to object attributes, even dynamic ones. The Proxy object provides an advanced level of access control that is simply not available with the other techniques. For example, the Proxy object allows us to intercept the deletion of a key in an object and to perform property existence checks.

Once again, it's worth highlighting that the Proxy object does not mutate the subject, so it can be safely used in contexts where the subject is shared between different components of the application. We also saw that with the Proxy object, we can easily perform delegation of all the methods and attributes that we want to leave unchanged.

In the next section, we present a more realistic example leveraging the Proxy pattern and use it to compare the different techniques we have discussed so far for implementing this pattern.

Creating a logging Writable stream

To see the Proxy pattern applied to a real example, we will now build an object that acts as a proxy to a Writable stream, which intercepts all the calls to the `write()` method and logs a message every time this happens. We will use the Proxy object to implement our proxy. Let's write our code in a file called `logging-writable.js`:

```
export function createLoggingWritable(writable) {  
  return new Proxy(writable, { // 1  
    get(target, propKey, _receiver) { // 2  
      if (propKey === 'write') { // 3  
        return (...args) => { // 4  
          const [chunk] = args  
          console.log('Writing', chunk)  
          return writable.write(...args)  
        }  
      }  
      return target[propKey] // 5  
    },  
  })  
}
```

In the preceding code, we created a factory that returns a proxied version of the `writable` object passed as an argument. Let's see what the main points of the implementation are:

1. We create and return a proxy for the original `writable` object using the ES2015 Proxy constructor.
2. We use the `get` trap to intercept access to the object properties.
3. We check whether the property accessed is the `write` method. If that is the case, we return a function to proxy the original behavior.

4. The proxy implementation logic here is simple: we extract the current chunk from the list of arguments passed to the original function, we log the content of the chunk, and finally, we invoke the original method with the given list of arguments.
5. We return unchanged any other property.

We can now use this newly created function and test our proxy implementation:

```
import { createWriteStream } from 'node:fs'
import { createLoggingWritable } from './logging-writable.js'

const writable = createWriteStream('test.txt')
const writableProxy = createLoggingWritable(writable)

writableProxy.write('First chunk')
writableProxy.write('Second chunk')
writable.write('This is not logged')
writableProxy.end()
```

The proxy did not change the original interface of the stream or its external behavior, but if we run the preceding code, we will now see that every chunk that is written into the `writableProxy` stream is transparently logged to the console.

Change Observer with Proxy

The **Change Observer pattern** is a design pattern in which an object (the subject) notifies one or more observers of any state changes, so that they can “react” to changes as soon as they happen.



Although very similar, the Change Observer pattern should not be confused with the Observer pattern discussed in *Chapter 3, Callbacks and Events*. The Change Observer pattern focuses on allowing the detection of property changes, while the Observer pattern is a more generic pattern that adopts an event emitter to propagate information about events happening in the system.

Proxies turn out to be quite an effective tool to create observable objects. Let’s see a possible implementation with `create-observable.js`:

```
export function createObservable(target, observer) {
  const observable = new Proxy(target, {
    set(obj, prop, value) {
      if (value !== obj[prop]) {
        const prev = obj[prop]
        obj[prop] = value
        observer({ prop, prev, curr: value })
      }
    }
  })
  return true
}
```

```

    },
  })

  return observable
}

```

In the previous code, `createObservable()` accepts a target object (the object to observe for changes) and an observer (a function to invoke every time a change is detected).

Here, we create the observable instance through an ES2015 proxy. The proxy implements the set trap, which is triggered every time a property is set. The implementation compares the current value with the new one and, if they are different, the target object is mutated, and the observer gets notified. When the observer is invoked, we pass an object literal that contains information related to the change (the name of the property, the previous value, and the current value).



This is a simplified implementation of the Change Observer pattern. More advanced implementations support multiple observers and use more traps to catch other types of mutation, such as field deletions or changes of prototype. Moreover, our implementation does not recursively create proxies for nested objects or arrays—a more advanced implementation takes care of these cases as well.

Let's see now how we can take advantage of observable objects with a trivial invoice application where the invoice total is updated automatically based on observed changes in the various fields of the invoice:

```

import { createObservable } from './create-observable.js'

function calculateTotal(invoice) {
  // 1
  return invoice.subtotal - invoice.discount + invoice.tax
}

const invoice = {
  subtotal: 100,
  discount: 10,
  tax: 20,
}

let total = calculateTotal(invoice)
console.log(`Starting total: ${total}`)

const obsInvoice = createObservable(
  // 2
  invoice,
  ({ prop, prev, curr }) => {
    total = calculateTotal(invoice)
  }
)

```

```
    console.log(`TOTAL: ${total} (${prop} changed: ${prev} -> ${curr})`)
  }
)

// 3
obsInvoice.subtotal = 200 // TOTAL: 210
obsInvoice.discount = 20 // TOTAL: 200
obsInvoice.discount = 20 // no change: doesn't notify
obsInvoice.tax = 30 // TOTAL: 210

console.log(`Final total: ${total}`)
```

In the previous example, an invoice is composed of a subtotal value, a discount value, and a tax value. The total amount can be calculated from these three values. Let's discuss the implementation in greater detail:

1. We declare a function that calculates the total for a given invoice, then we create an invoice object and a value to hold the total for it.
2. Here, we create an observable version of the invoice object. Every time there is a change in the original invoice object, we recalculate the total, and we also print some logs to keep track of the changes.
3. Finally, we apply some changes to the observable invoice. Every time we mutate the `obsInvoice` object, the observer function is triggered, the total gets updated, and some logs are printed on the screen.

If we run this example, we will see the following output in the console:

```
Starting total: 110
TOTAL: 210 (subtotal changed: 100 -> 200)
TOTAL: 200 (discount changed: 10 -> 20)
TOTAL: 210 (tax changed: 20 -> 30)
Final total: 210
```

In this example, we could make the total calculation logic arbitrarily complicated, for instance, by introducing new fields in the computation (shipping costs, other taxes, and so on). In this case, it will be fairly trivial to introduce the new fields in the invoice object and update the `calculateTotal()` function. Once we do that, every change to the new properties will be observed and the total will be kept up to date with every change.



Observables are the cornerstone of **reactive programming (RP)** and **functional reactive programming (FRP)**. If you are curious to know more about these styles of programming, check out the *Reactive Manifesto* at nodejsdp.link/reactive-manifesto.

In the wild

The Proxy pattern and, more specifically, the Change Observer pattern are widely adopted patterns that can be found on backend projects and libraries as well as in the frontend world. Some popular projects that take advantage of these patterns include the following:

- LoopBack (nodejsdp.link/loopback) is a popular Node.js web framework that uses the Proxy pattern to provide the capability to intercept and enhance method calls on controllers. This capability can be used to build custom validation or authentication mechanisms.
- Vue.js (nodejsdp.link/vue), a very popular JavaScript reactive UI framework, has reimplemented observable properties using the Proxy pattern with the Proxy object.
- MobX (nodejsdp.link/mobx) is a famous reactive state management library commonly used in frontend applications in combination with React or Vue.js. Like Vue.js, MobX implements reactive observables using the Proxy object.

Decorator

Decorator is a structural design pattern that consists of dynamically augmenting the behavior of an existing object. It's different from classical inheritance because the behavior is not added to all the objects of the same class, but only to the instances that are explicitly decorated.

Implementation-wise, it is very similar to the Proxy pattern, but instead of enhancing or modifying the behavior of the existing interface of an object, it augments it with new functionalities, as depicted in *Figure 8.2*:

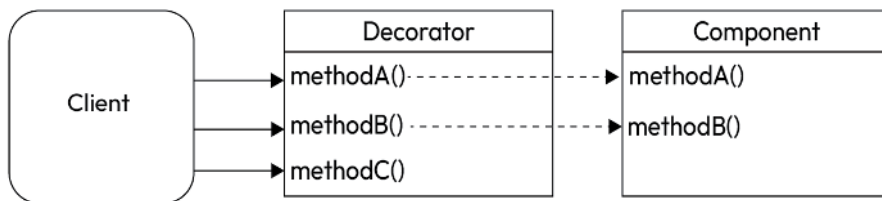


Figure 8.2: Decorator pattern schematic

In *Figure 8.2*, the Decorator object is extending the Component object by adding the `methodC()` operation. The existing methods are usually delegated to the decorated object without further processing, but in some cases, they might also be intercepted and augmented with extra behaviors.

Techniques for implementing decorators

Although Proxy and Decorator are conceptually two different patterns with different intents, they practically share the same implementation strategies. We will review them shortly. This time, we want to use the Decorator pattern to be able to take an instance of our `StackCalculator` class and “decorate it” so that it also exposes a new method called `add()`, which we can use to perform additions between two numbers. We will also use the decorator to intercept all the calls to the `divide()` method and implement the same division-by-zero check that we already saw in our `SafeCalculator` example.

Composition

Using composition, the decorated component is wrapped around a new object that usually inherits from it. The decorator in this case simply needs to define the new methods, while delegating the existing ones to the original component:

```
class EnhancedCalculator {
  constructor(calculator) {
    this.calculator = calculator
  }

  // new method
  add() {
    const addend2 = this.getValue()
    const addend1 = this.getValue()
    const result = addend1 + addend2
    this.putValue(result)
    return result
  }

  // modified method
  divide() {
    // additional validation logic
    const divisor = this.calculator.peekValue()
    if (divisor === 0) {
      throw new Error('Division by 0')
    }
    // if valid delegates to the subject
    return this.calculator.divide()
  }

  // delegated methods
  putValue(value) {
    return this.calculator.putValue(value)
  }

  getValue() {
    return this.calculator.getValue()
  }

  peekValue() {
    return this.calculator.peekValue()
  }
}
```

```

    }

    clear() {
        return this.calculator.clear()
    }

    multiply() {
        return this.calculator.multiply()
    }
}

const calculator = new StackCalculator()
const enhancedCalculator = new EnhancedCalculator(calculator)

enhancedCalculator.putValue(4)
enhancedCalculator.putValue(3)
console.log(enhancedCalculator.add()) // 4+3 = 7
enhancedCalculator.putValue(2)
console.log(enhancedCalculator.multiply()) // 7*2 = 14

```

If you remember our composition implementation for the Proxy pattern, you can probably see that the code here looks quite similar.

We created the new `add()` method and enhanced the behavior of the original `divide()` method (effectively replicating the feature we saw in the previous `SafeCalculator` example). Finally, we delegated the `putValue()`, `getValue()`, `peekValue()`, `clear()`, and `multiply()` methods to the original subject.

Object decoration

Object decoration can also be achieved by simply attaching new methods directly to the decorated object (monkey patching), as follows:

```

function patchCalculator(calculator) {
    // new method
    calculator.add = () => {
        const addend2 = calculator.getValue()
        const addend1 = calculator.getValue()
        const result = addend1 + addend2
        calculator.putValue(result)
        return result
    }

    // modified method
    const divideOrig = calculator.divide

```

```

calculator.divide = () => {
  // additional validation logic
  const divisor = calculator.peekValue()
  if (divisor === 0) {
    throw new Error('Division by 0')
  }
  // if valid delegates to the subject
  return divideOrig.apply(calculator)
}

return calculator
}

const calculator = new StackCalculator()
const enhancedCalculator = patchCalculator(calculator)
// ...

```

Note that in this example, `calculator` and `enhancedCalculator` reference the same object (`calculator == enhancedCalculator`). This is because `patchCalculator()` is mutating the original calculator object and then returning it. You can confirm this by invoking `calculator.add()` or `calculator.divide()`.

Decorating with the Proxy object

It's possible to implement object decoration by using the Proxy object. A generic example might look like this:

```

const enhancedCalculatorHandler = {
  get(target, property) {
    if (property === 'add') {
      // new method
      return function add() {
        const addend2 = target.getValue()
        const addend1 = target.getValue()
        const result = addend1 + addend2
        target.putValue(result)
        return result
      }
    }

    if (property === 'divide') {
      // modified method
      return () => {
        // additional validation logic

```



```

    const divisor = target.peekValue()
    if (divisor === 0) {
      throw new Error('Division by 0')
    }
    // if valid delegates to the subject
    return target.divide()
  }
}

// delegated methods and properties
return target[property]
},
}

const calculator = new StackCalculator()
const enhancedCalculator = new Proxy(
  calculator,
  enhancedCalculatorHandler
)
// ...

```

If we were to compare these different implementations, the same caveats discussed during the analysis of the Proxy pattern would also apply for the decorator. Let's focus instead on practicing the pattern with a real-life example!



Regular functions vs arrow functions in proxies

If you've been paying close attention, you might have noticed that our proxy handler uses a regular function for the `add()` method and an arrow function for `divide()`. This doesn't have any practical implication in our example, but it's something that is left there to make you think. In JavaScript, the value of `this` behaves differently depending on the type of function:

- Regular functions set `this` based on how they're called. In the context of a proxy trap, returning a regular function means that `this` inside that function will be bound to the proxy target.
- Arrow functions, on the other hand, don't have their own `this`. Instead, they inherit `this` from the surrounding lexical scope. In this case, the proxy trap itself.



In practice, this rarely matters when working with proxies, because you usually have direct access to the `target` from the trap's parameters (as we did in our example). But it's still a subtle distinction worth remembering, especially if you ever need to rely on this inside the function body.

In short, arrow functions are not a drop-in replacement for regular functions when this matters. Keep that in mind to avoid subtle and frustrating bugs!

Decorating a Level database

Before we start coding the next example, let's say a few words about **Level**, the module that we are now going to work with.

Introducing Level and LevelDB

Level is a powerful and flexible ecosystem of JavaScript modules for building key-value databases. Designed with transparency and modularity in mind, it provides a solid foundation of low-level primitives that let you create custom databases, tailored to your specific needs. These databases can be embedded or networked, persistent or in-memory.

At its core, Level builds on the principles of **LevelDB** (nodejsdp.link/leveldb), a high-performance key-value store developed at Google. Like LevelDB, Level databases support binary keys and values, efficient batched writes, and fast bi-directional iteration over sorted entries. This lexicographic ordering, combined with range queries and snapshots, enables powerful and consistent read operations. What makes Level particularly well suited to Node.js is its integration with familiar interfaces: streams, events, buffers, and async iterators. It also supports advanced features such as custom encodings and sublevels, which allow you to organize your data into modular, event-driven sections within a single database. Level can be used in both Node.js on the server and in the browser on the client, making it a great choice for full stack JavaScript applications.

Today, there is a vast ecosystem around Level made of plugins and modules that extend the tiny core to implement features such as replication, secondary indexes, live updates, query engines, and more. Complete databases were also built on top of Level, including CouchDB clones such as PouchDB (nodejsdp.link/pouchdb), and even a graph database, LevelGraph (nodejsdp.link/levelgraph), which can work both on Node.js and the browser!



Find out more about the Level ecosystem at nodejsdp.link/awesome-level.

Implementing a Level plugin

In the next example, we are going to show you how we can create a simple plugin for Level using the Decorator pattern, and in particular, the object augmentation technique, which is the simplest but also the most pragmatic and effective way to decorate objects with additional capabilities.

What we want to build is a plugin for Level that allows us to receive notifications every time an object with a certain pattern is saved into the database. For example, if we subscribe to a pattern such as `{a: 1}`, we want to receive a notification when objects such as `{a: 1, b: 3}` or `{a: 1, c: 'x'}` are saved into the database.

Let's start to build our small plugin by creating a new module called `level-subscribe.js`. We will then insert the following code:

```
export function levelSubscribe(db) {
  db.subscribe = (pattern, listener) => { // 1
    db.on('write', docs => { // 2
      for (const doc of docs) {
        const match = Object.keys(pattern).every(
          k => pattern[k] === doc.value[k] // 3
        )
        if (match) {
          listener(doc.key, doc.value) // 4
        }
      }
    })
  }

  return db
}
```

That's it for our plugin; it's extremely simple. Let's briefly analyze the preceding code:

1. We decorate the `db` object with a new method named `subscribe()`. We simply attach the method directly to the provided `db` instance (object augmentation).
2. We listen for any `write` operation performed on the database.
3. We perform a very simple pattern-matching algorithm, which verifies that all the properties in the provided pattern are also available in the data being inserted.
4. If we have a match, we notify the listener.

Let's now write some code to try out our new plugin:

```
import { join } from 'node:path'
import { Level } from 'level' // v9.0.0
import { levelSubscribe } from './level-subscribe.js'

const dbPath = join(import.meta.dirname, 'db')
const db = new Level(dbPath, { valueEncoding: 'json' }) // 1
levelSubscribe(db) // 2
```

```
db.subscribe( // 3
  { doctype: 'message', language: 'en' },
  (_k, val) => console.log(val)
)
await db.put('1', { // 4
  doctype: 'message',
  text: 'Hi',
  language: 'en',
})
await db.put('2', {
  doctype: 'company',
  name: 'ACME Co.',
})
```

This is how the preceding code works:

1. First, we initialize our Level database, choosing the directory where the files are stored and the default encoding for the values.
2. Then, we attach our plugin, which decorates the original db object.
3. At this point, we are ready to use the new feature provided by our plugin, which is the `subscribe()` method, where we specify that we are interested in all the objects with `doctype: 'message'` and `language: 'en'`.
4. Finally, we save some values in the database using `put`. The first call triggers the callback associated with our write subscription, and we should see the stored object printed to the console. This is because, in this case, the object matches the subscription. The second call does not generate any output because the stored object does not match the subscription criteria; in fact, the second record is missing the field `language`.

This example shows a real application of the Decorator pattern in its simplest implementation, which is object augmentation. It may look like a trivial pattern, but it has undeniable power if used appropriately.

In the wild

For more examples of how decorators are used in the real world, you can inspect the code of some more Level plugins:

- `levelgraph` (nodejsdp.link/levelgraph): a plugin that adds graph database capabilities to Level.
- `search-index` (nodejsdp.link/search-index): a plugin that adds full-text search capabilities to Level.
- `level-ttl` (nodejsdp.link/level-ttl): adds a TTL (Time To Live) to automatically expire keys in a Level instance.

Aside from Level plugins, the following projects are also good examples of the adoption of the Decorator pattern:

- `json-socket` (nodejsdp.link/json-socket): This module makes it easier to send JSON data over a TCP (or a Unix) socket. It is designed to decorate an existing instance of `net.Socket`, which gets enriched with additional methods and behaviors.
- `fastify` (nodejsdp.link/fastify) is a web application framework that exposes an API to decorate a Fastify server instance with additional functionality or configuration. With this approach, the additional functionality is made accessible to different parts of the application. This is a quite generalized implementation of the Decorator pattern. Check out the dedicated documentation page to find out more, at nodejsdp.link/fastify-decorators.

The decorator proposal for ECMAScript

It's important to distinguish the Decorator design pattern, as described here, from the Decorators proposal for ECMAScript (nodejsdp.link/proposal-decorators), at the time of writing, in Stage 3 of standardization. While both concepts share the name “Decorator,” they operate in different contexts and serve different purposes.

The Decorator design pattern is a structural pattern for dynamically adding responsibilities to individual objects at runtime. In contrast, the ECMAScript Decorators proposal aims to provide a new declarative syntax for modifying or extending classes and their members, and, to give you a concrete example, it looks like this:

```
@defineElement("my-class")
class C extends HTMLElement {
  @reactive accessor clicked = false;
}
```

`@defineElement` and `@reactive` are examples of the new proposed syntax that allows you to decorate a class or a class element. A similar syntax exists in other languages (e.g., Java and Python).

The ECMAScript Decorators proposal is already adopted in *transpiled* JavaScript environments and has gained significant interest in standardization. Frameworks like **Angular** (nodejsdp.link/angular) and **NestJS** (nodejsdp.link/nestjs) rely heavily on this decorator syntax, which requires transpilation to be used in Node.js and the browser.

Even TypeScript has added support for the decorator syntax proposed in the ECMAScript Decorators proposal. When utilizing the TypeScript compiler, the necessary *polyfill code* is automatically generated for you, ensuring compatibility with JavaScript environments that do not natively support the syntax.

The core distinction between the Decorator syntax and the Decorator pattern lies in their purpose and application. The Decorator pattern is a general design technique used to enhance individual object instances with additional functionality or to modify their behavior. In contrast, the Decorator syntax proposed by the TC39 committee focuses on extending the definition of JavaScript classes through class-level modifiers.

While the name might suggest a connection, the two concepts address different use cases. The latest TC39 proposal builds on years of iteration and previous experiments, but it remains centered on class extension rather than instance augmentation, as seen in the design pattern.



The TC39 committee has been working on the specification for class-level decorators for several years now. However, it's currently unclear when or whether this proposal will reach Stage 4, the final and stable stage, indicating it's ready for implementation and wide adoption. Even though it's not uncommon to see this syntax already being adopted in the wild, we would advise caution against relying on this feature in production code until it is finalized, as any changes to its final design could result in breaking changes and negatively impact existing codebases. In addition, because decorators in TypeScript are implemented as compiler transforms that produce runtime code, Node.js built-in type stripping is not enough. You still need a transpilation step, which is another reason to avoid this approach for now.

The line between Proxy and Decorator

At this point in the book, you might have some legitimate doubts about the differences between the Proxy and the Decorator patterns. These two patterns are indeed very similar, and they can sometimes be used interchangeably.

In its classic incarnation, the Decorator pattern is defined as a mechanism that allows us to enhance an existing object with new behavior, while the Proxy pattern is used to control access to a concrete or virtual object.

There is a conceptual difference between the two patterns, and it's mostly based on the way they are used at runtime.

You can look at the Decorator pattern as a wrapper; you can take different types of objects and decide to wrap them with a decorator to enhance their capabilities with extra functionality. A proxy, instead, is used to control the access to an object, and it does not change the original interface. For this reason, once you have created a proxy instance, you can pass it over to a context that expects the original object.

When it comes to implementation, these differences are generally much more obvious with strongly typed languages where the type of the objects you pass around is checked at compile time. In the Node.js ecosystem, given the dynamic nature of the JavaScript language, the line between the Proxy and the Decorator patterns is quite blurry, and often the two names are used interchangeably. We have also seen how the same techniques can be used to implement both patterns.

When dealing with JavaScript and Node.js, our advice is to avoid getting bogged down with the nomenclature and the canonical definition of these two patterns. We encourage you to look at the class of problems that proxy and decorator solve as a whole and treat these two patterns as complementary and sometimes interchangeable tools.

Adapter

The Adapter pattern allows us to access the functionality of an object using a different interface.

A real-life example of an adapter would be a device that allows you to plug a USB Type-A cable into a USB Type-C port. In a generic sense, an adapter converts an object with a given interface so that it can be used in a context where a different interface is expected.

In software, the Adapter pattern is used to take the interface of an object (the **adaptee**) and make it compatible with another interface that is expected by a given client. Let's have a look at *Figure 8.3* to clarify this idea:

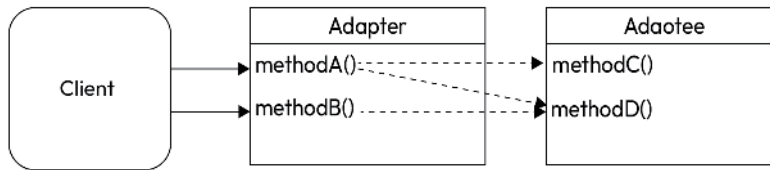


Figure 8.3: Adapter pattern schematic

In *Figure 8.3*, we can see how the adapter is essentially a wrapper for the adaptee, exposing a different interface. The diagram also highlights the fact that the operations of the adapter can also be a composition of one or more method invocations on the adaptee. From an implementation perspective, the most common technique is composition, where the methods of the adapter provide a bridge to the methods of the adaptee. This pattern is straightforward, so let's immediately work on an example.

Using Level through the filesystem API

We are now going to build an adapter around the Level API, transforming it into an interface that is loosely compatible with some functionality provided by the core `node:fs/promises` module. In particular, we will make sure that every call to `readFile()` and `writeFile()` will translate into calls to `db.get()` and `db.put()`. This way, we will be able to use a Level database as a storage backend for simple filesystem operations.



While implementing a full Level adapter for `node:fs/promises` is definitely possible, doing that would take us beyond the focus of this section. Once you have completed this section, feel free to take on that challenge as an exercise to help you solidify your understanding of the concepts we've covered.

Let's start by creating a new module named `fs-adapter.js`. We will begin by loading the dependencies and exporting the `createFsAdapter()` factory that we are going to use to build the adapter:

```
import { resolve } from 'node:path'

export function createFsAdapter (db) {
  return ({
    async readFile (filename, options = undefined) {
```

```

    // ...
  },
  async writeFile (filename, contents, options = undefined) {
    // ...
  }
})
}

```

Next, we will implement the `readFile()` function inside the factory and ensure that its interface is compatible with the original function from the `fs` module:

```

async readFile(filename, options = undefined) {
  const valueEncoding =
    typeof options === 'string' ? options : options?.encoding
  const opt = valueEncoding ? { valueEncoding } : undefined // 1
  const value = await db.get(resolve(filename), opt) // 2
  if (typeof value === 'undefined') { // 3
    const e = new Error(
      `ENOENT: no such file or directory, open '${filename}'`
    )
    e.code = 'ENOENT'
    e.errno = 34
    e.path = filename
    throw e
  }
  return value // 4
}

```

In the preceding code, we had to do some extra work to make sure that the behavior of our new function is as close as possible to the original `readFile()` function from the `node:fs/promises` module. The steps performed by the function are described as follows:

1. The only option we are interested in is encoding. This can be passed as a plain string (e.g., `options = 'utf8'`) or as an object with a key named `encoding` (e.g., `options = { encoding: 'utf8' }`). If no value is specified, the default option is used (`undefined`), which maps to a raw binary encoding. The encoding is used to automatically decode the value read from the database. If `'utf8'` (or another text encoding) is provided, the value returned will be a string; otherwise, it will be a `Buffer`. This is consistent with how the original `readFile()` function works.
2. To retrieve a file from the `db` instance, we invoke `db.get()`, using `filename` as a key, by making sure to always use its full path (using `resolve()`). We set the value of the `valueEncoding` option used by the database to be equal to any eventual encoding option received as input.

3. If the key is not found in the database, we create and throw an error with `ENOENT` as the error code, which is the code used by the original `fs` module to indicate a missing file.
4. If the key-value pair is retrieved successfully from the database, we will return the value to the caller.

The function that we created does not want to be a perfect replacement for the `readFile()` function, but it definitely does its job in the most common situations.

To complete our small adapter, let's now see how to implement the `writeFile()` function:

```
async writeFile(filename, contents, options = undefined) {
  const valueEncoding =
    typeof options === 'string' ? options : options?.encoding
  const opt = valueEncoding ? { valueEncoding } : undefined
  await db.put(resolve(filename), contents, opt)
}
```

As we can see, we don't have a perfect wrapper in this case either. We are ignoring some options, such as file permissions (`options.mode`), and we are forwarding any error that we receive from the database as is.

Our new adapter is now ready, but before using it, let's write a small test module that only uses the original `node:fs/promises` API:

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

await fs.writeFile('file.txt', 'Hello!', 'utf8')
const res = await fs.readFile('file.txt', 'utf8')
console.log(res)

// try to read a missing file (throws an error)
await fs.readFile('missing.txt')
```

The preceding code uses the original `fs` API to perform a few read and write operations on the filesystem, and should print something like the following to the console:

```
Hello!
Error: ENOENT: no such file or directory, open 'missing.txt'
```

The first line shows the content of the file that we created with `writeFile()` and then read with `readFile()`, while the second line displays the error that was thrown when trying to access the non-existent file `missing.txt`.

Now, we can try to replace the `fs` module with our adapter, as follows:

```
import { join } from 'node:path'
import { Level } from 'level' // v9.0.0
import { createFsAdapter } from './fs-adapter.js'

const db = new Level(join(import.meta.dirname, 'db'), {
  valueEncoding: 'binary',
})
const fs = createFsAdapter(db)
// ...
```

Running our program again should produce the same output, except for the fact that no parts of the file that we specified are read or written using the filesystem API directly. Instead, any operation performed using our adapter will be converted into an operation performed on a Level database.

The adapter that we just created might look silly; what's the purpose of using a database in place of the real filesystem? However, we should remember that Level itself has adapters that enable the database to also run in the browser. One of these adapters is `browser-level` (nodejsdp.link/browser-level). Now our adapter makes perfect sense. We could use something similar to allow code leveraging the `fs` module to run on both Node.js and a browser.

In the wild

There are plenty of real-world examples of the Adapter pattern. We've listed some of the most notable examples here for you to explore and analyze:

- We already know that Level can run with different storage backends, from the default LevelDB to IndexedDB in the browser. This is made possible by the various adapters that are created to replicate the internal (private) Level API. Take a look at some of them to see how they are implemented at nodejsdp.link/level-stores.
- Prisma ORM (nodejsdp.link/prisma), a famous Node.js ORM, supports several different databases (PostgreSQL, MySQL, MongoDB, etc.) using the Adapter pattern to implement database-specific drivers.
- The perfect complement to the example that we created is `level-file-system` (nodejsdp.link/level-file-system), which is a more complete implementation of the `fs` API on top of Level.

Summary

Structural design patterns are definitely some of the most widely adopted design patterns in software engineering, and it is important to be confident with them. In this chapter, we explored the Proxy, the Decorator, and the Adapter patterns, and we discussed different ways to implement these in the context of Node.js.

We saw how the Proxy pattern can be a very valuable tool to control access to existing objects. In this chapter, we also mentioned how the Proxy pattern can enable different programming paradigms, such as reactive programming using the Change Observer pattern.

In the second part of the chapter, we found out that the Decorator pattern is an invaluable tool to be able to add additional functionality to existing objects. We saw that its implementation doesn't differ much from the Proxy pattern, and we explored some examples built around the LevelDB ecosystem.

Finally, we discussed the Adapter pattern, which allows us to wrap an existing object and expose its functionality through a different interface. We saw that this pattern can be useful to expose a piece of existing functionality to a component that expects a different interface. In our examples, we saw how this pattern can be used to implement an alternative storage layer that is compatible with the interface provided by the `fs` module to interact with files.

Proxy, Decorator, and Adapter are very similar; the difference between them can be appreciated from the perspective of the interface consumer: Proxy provides the same interface as the wrapped object, Decorator provides an enhanced interface, and Adapter provides a different interface.

In the next chapter, we will complete our journey through traditional design patterns in Node.js by exploring the category of behavioral design patterns. This category includes important patterns such as the Strategy pattern, the Middleware pattern, and the Iterator pattern. Are you ready to discover behavioral design patterns?

Exercises

- **8.1 HTTP client cache:** Write a proxy for your favorite HTTP client library that caches the response of a given HTTP request, so that if you make the same request again, the response is immediately returned from the local cache, rather than being fetched from the remote URL. If you need inspiration, you can check out the `superagent-cache` module ([nodejsdp.link/superagent-cache](https://github.com/indutny/superagent-cache)).
- **8.2 Timestamped logs:** Create a proxy for the console object that enhances every logging function (`log()`, `error()`, `debug()`, and `info()`) by prepending the current timestamp to the message you want to print in the logs. For instance, executing `consoleProxy.log('hello')` should print something like `2020-02-18T15:59:30.699Z hello` in the console.

- **8.3 Colored console output:** Write a decorator for the console that adds the `red(message)`, `yellow(message)`, and `green(message)` methods. These methods will have to behave like `console.log(message)` except they will print the message in red, yellow, or green, respectively. In one of the exercises from the previous chapter, we already pointed you to some useful packages to create colored console output. If you want to try something different this time, have a look at `ansi-styles` (nodejsdp.link/ansi-styles).
- **8.4 Virtual filesystem:** Modify our LevelDB filesystem adapter example to write the file data in memory rather than in LevelDB. You can use an object or a Map instance to store the key-value pairs of filenames and the associated data.
- **8.5 The lazy buffer:** Can you implement `createLazyBuffer(size)`, a factory function that generates a virtual proxy for a Buffer of the given size? The proxy instance should instantiate a Buffer object (effectively allocating the given amount of memory) only when `write()` is being invoked for the first time. If no attempt to write into the buffer is made, no Buffer instance should be created.

9

Behavioral Design Patterns

In the last two chapters, we learned about patterns that can help us in the creation of objects and in building complex object structures. Now, we are ready to shift our focus to another critical aspect of software design: behavior.

In this chapter, we will learn how objects interact, how they can be combined, and how their communication can be structured to produce extensible, modular, reusable, and adaptable systems. We will address questions such as “*How can we modify parts of an algorithm at runtime?*”, “*How can an object change its behavior based on its state?*”, and “*How can we iterate over a collection without knowing its underlying implementation?*”. These are the kinds of challenges that behavioral design patterns help solve.

We have already encountered a notable member of this category of patterns, and that is the Observer pattern, which we presented in *Chapter 3, Callbacks and Events*. The Observer pattern is one of the foundational patterns of the Node.js platform as it provides us with a simple interface for dealing with events and subscriptions, which are the life force of Node’s event-driven architecture.

If you are already familiar with the **Gang of Four (GoF)** design patterns, in this chapter, you will see, once again, how the implementation of some of these patterns can be radically different in JavaScript compared to purer object-oriented languages such as Java or C++. A striking example is the Iterator pattern, which we will explore later in the chapter. Unlike in classical OOP, where implementing an iterator often requires defining interfaces, extending classes, and defining complex hierarchies, JavaScript allows us to achieve the same functionality simply by adding a special method to a class or directly to an object.

Additionally, one pattern covered in this chapter, the Middleware pattern, closely resembles the GoF’s Chain of Responsibility pattern. However, its widespread adoption in Node.js has made it so fundamental that it is often considered a distinct pattern in its own right.

Now, it's time to roll up our sleeves and get our hands dirty with some behavioral design patterns. In this chapter, you will learn about the following:

- The Strategy pattern, which helps us change parts of a component to adapt it to specific needs
- The State pattern, which allows us to change the behavior of a component based on its state
- The Template pattern, which allows us to reuse the structure of a component to define new ones
- The Iterator pattern, which provides us with a common interface to iterate over a collection
- The Middleware pattern, which allows us to define a modular chain of processing steps
- The Command pattern, which materializes the information required to execute a routine, allowing such information to be easily transferred, stored, and processed

Strategy

The **Strategy** pattern allows an object, known as the **context**, to *adapt* its behavior by delegating specific parts of its logic to interchangeable objects called **strategies**. The context defines the common structure of an algorithm, while each strategy encapsulates a variation of its mutable aspects. This approach enables the context to modify its behavior dynamically based on factors such as input values, system configurations, or user preferences.



Here's a real-life story (from Mario) about how the Strategy pattern helped me optimize a performance-critical operation. While this example isn't directly related to Node.js, it demonstrates just how fundamental and universal this pattern is.

Years ago, I was working on a 3D game engine as a personal project. Game engines are notoriously resource-intensive, where every optimization matters to ensure smooth performance. A key component of 3D rendering is *matrix multiplication*, and at the time (early 2000s), CPU manufacturers such as Intel and AMD were introducing specialized instructions to accelerate mathematical operations through parallelism. These single instruction, multiple data (SIMD) instructions were ideal for speeding up matrix multiplications, but there was a catch: different processors supported different instruction sets, primarily *3DNow!* on AMD and *SSE* on Intel.

To maximize performance across all mainstream processors, my engine needed to support multiple instruction sets. The challenge, however, was that leveraging these instructions required writing low-level assembly code. My initial approach looked something like this (in pseudo-code):

```
function matrixMultiply(data) {  
  if (is3DNowAvailable) {  
    return matrixMultiply3DNow(data);  
  }  
  if (isSSEAvailable) {  
    return matrixMultiplySSE(data);  
  }  
  return slowMatrixMultiplyFallback(data);  
}
```

At first glance, this seemed like a reasonable solution. However, there was a critical issue: the `matrixMultiply` function was called tens of thousands of times per second. The conditional checks on every call introduced unnecessary overhead, far from ideal when optimizing for peak efficiency.



While searching for a better approach, I discovered the Strategy pattern. The concept of “defining a family of algorithms and making them interchangeable” was exactly what I needed. I refactored my implementation to select the optimal function once at startup:

```
matrixMultiply = null;  
if (is3DNowAvailable) {  
  matrixMultiply = matrixMultiply3DNow;  
} else if (isSSEAvailable) {  
  matrixMultiply = matrixMultiplySSE;  
} else {  
  matrixMultiply = matrixMultiplyFallback;  
}
```

With this approach, the correct algorithm was chosen only once during initialization, eliminating the need for repeated conditional checks. The performance gain was immediate, and I was struck by the elegance of the solution. I couldn’t believe I hadn’t thought of it earlier, but that’s the kind of insight that experience brings.

Since then, the Strategy pattern has become one of my go-to techniques, proving invaluable across many different projects.

On a side note, it’s fascinating that, two decades later, matrix multiplications remain at the forefront of hardware innovation, powering everything from neural networks to modern AI applications.

Strategies are usually part of a family of solutions, and all of them implement the same interface expected by the context. The following figure shows the situation we just described:

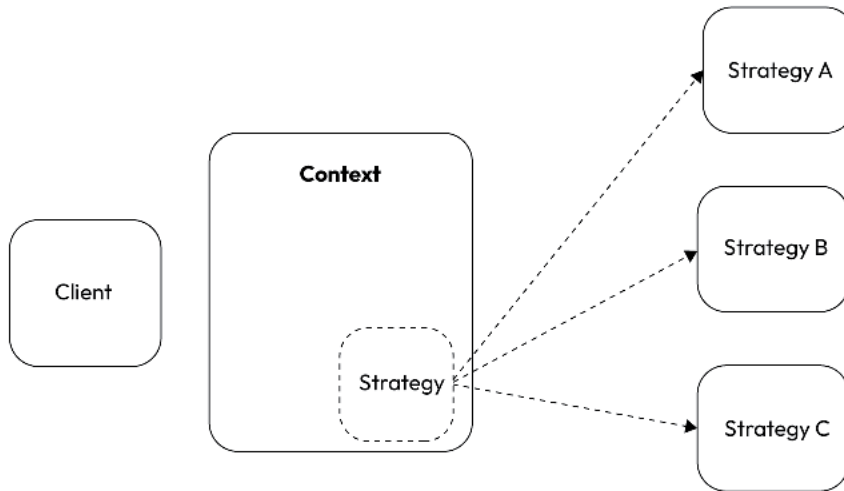


Figure 9.1: General structure of the Strategy pattern

Figure 9.1 shows you how the context object can plug different strategies into its structure as if they were replaceable parts of a piece of machinery. Imagine a car; its tires can be considered its strategy for adapting to different road conditions. We can fit winter tires to go on snowy roads thanks to their studs, while we can decide to fit high-performance tires for traveling mainly on motorways for a long trip. On the one hand, we don't want to change the entire car for this to be possible, and on the other, we don't want a car with eight wheels so that it can go on every possible road.

We quickly understand how powerful this pattern is. Not only does it help with separating the concerns within a given problem, but it also enables our solution to have better flexibility and adapt to different variations of the same problem.

The Strategy pattern is particularly useful in all those situations where supporting variations in the behavior of a component requires complex conditional logic (lots of `if...else` or `switch` statements) or mixing different components of the same family.

Here's another example: imagine an object called `Order` that represents an online order on an e-commerce website. The object has a method called `pay()` that, as it says, finalizes the order and transfers the funds from the user to the online store.

To support different payment systems, we have a couple of options:

- Use an `if...else` statement in the `pay()` method to complete the operation based on the chosen payment option
- Delegate the logic of the payment to a strategy object that implements the logic for the specific payment gateway selected by the user

In the first solution, our `Order` object cannot support other payment methods unless its code is modified. Also, this can become quite complex when the number of payment options grows. Instead, using the Strategy pattern enables the `Order` object to support a virtually unlimited number of payment methods and keeps its scope limited to only managing the details of the user, the purchased items, and the relative price while delegating the job of completing the payment to another object.

Let's now demonstrate this pattern with a simple, realistic example.

Multi-format configuration objects

Let's consider an object called `Config` that holds a set of configuration parameters used by an application server, the listening port and the listening address of the server, timeout configuration, environment variables, and so on.

To make this example a little stricter, we are going to use TypeScript to define what the configuration object shape looks like:

```
// configData.ts
export type ConfigData = {
  listen: {
    port: number
    host: string
  }
  timeouts: {
    headersTimeoutMs: number
    keepAliveTimeoutMs: number
    requestTimeoutMs: number
  }
  env: Record<string, string>
}
```



If you prefer not to use TypeScript, you can simply ignore the type definitions and annotations—the code will function just the same. The primary benefit of using TypeScript in this context is that it provides type safety, ensuring that once the configuration object is loaded, you can only access and modify supported configuration options. This helps prevent errors and improves maintainability.

The `Config` object should be able to provide a simple interface to access these parameters, but also a way to import and export the configuration using persistent storage, such as a file. We want to be able to support different formats to store the configuration, for example, JSON, YAML, or TOML.

By applying what we learned about the Strategy pattern, we can immediately identify the variable part of the `Config` object, which is the functionality that allows us to serialize and deserialize the configuration. This is going to be implemented by our strategies.

Let's create a new module called `config.ts`, and let's define the *generic* part of our configuration manager:

```
import { readFile, writeFile } from 'node:fs/promises'
import type { FormatStrategy } from './strategies.ts'
import type { ConfigData } from './configData.ts'

export class Config {
  data?: ConfigData
  formatStrategy: FormatStrategy

  constructor(formatStrategy: FormatStrategy) { // 1
    this.data = undefined
    this.formatStrategy = formatStrategy
  }

  async load(filePath: string): Promise<void> { // 2
    console.log(`Deserializing from ${filePath}`)
    this.data = this.formatStrategy.deserialize(
      await readFile(filePath, 'utf-8')
    )
  }

  async save(filePath: string): Promise<void> { // 3
    if (!this.data) {
      throw new Error('No data to save')
    }

    console.log(`Serializing to ${filePath}`)
    await writeFile(filePath, this.formatStrategy.serialize(this.data))
  }
}
```

This is what's happening in the preceding code:

1. In the constructor, we create an instance variable called `data` to hold the configuration data (initially undefined). This variable will hold the configuration once loaded from a file, and, therefore, it has type `ConfigData`. Then, we also store `formatStrategy`, which represents the component that we will use to parse and serialize the data. This is the variable part of our module, and it should be passed into the constructor. Note that this is typed as `FormatStrategy`; we will define it in a bit.

2. We provide a method called `load()` that allows us to read the content of a given configuration file and parse it with our current strategy. Once the parsing is complete, the resulting object will be stored in the `data` instance variable.
3. Similarly, we provide a `save()` method, which, if configuration data has been loaded (and potentially modified), allows us to persist these changes on a given file path using our current configuration strategy for serialization.

As we can see, this very simple and neat design allows the `Config` object to seamlessly support different file formats when loading and saving its data. The best part is that the logic to support those various formats is not hardcoded anywhere, so the `Config` class can adapt without any modification to virtually any file format, given the right strategy.

To demonstrate this characteristic, let's now create a couple of format strategies in a file called `strategies.ts`.



For simplicity, we're defining all the strategies for parsing configuration files (JSON, YAML, and TOML) within a single file. However, in a real-world scenario where additional formats might be introduced over time, it would be more practical to organize each format's strategy into its own dedicated file. This approach improves maintainability and makes it easier to extend the system as needed.

Let's start by providing a TypeScript definition for what a valid `FormatStrategy` would look like:

```
// strategies.ts
export type FormatStrategy = {
  deserialize: (data: string) => ConfigData
  serialize: (data: ConfigData) => string
}
```

Any object that implements these two methods qualifies as a format strategy:

- `deserialize()`: Converts a raw configuration string into a structured `ConfigData` object.
- `serialize()`: Performs the reverse operation, transforming a `ConfigData` object into its serialized string representation.

At this point, we are ready to implement our first strategy. Let's start with the JSON file format:

```
// strategies.ts
// ...

export const jsonStrategy: FormatStrategy = {
  deserialize(data): ConfigData {
    return JSON.parse(data)
  },
  serialize(data: ConfigData): string {
    return JSON.stringify(data, null, 2)
  }
}
```

```
  },
}
```

Nothing really complicated! Our strategy simply implements the agreed interface, so that it can be used by the Config object.

Similarly, we can implement equivalent strategies for the YAML and TOML formats. The key difference is that, unlike JSON, these formats are not natively supported in the Node.js standard library. To handle serialization and deserialization correctly for these formats, we'll need to rely on external libraries:

```
// strategies.ts
import YAML from 'yaml' // v2.7.0
import TOML from 'smol-toml' // v1.3.1
// ...

export const yamlStrategy: FormatStrategy = {
  deserialize(data): ConfigData {
    return YAML.parse(data)
  },
  serialize(data: ConfigData): string {
    return YAML.stringify(data, { indent: 2 })
  },
}

export const tomlStrategy: FormatStrategy = {
  deserialize(data): ConfigData {
    return TOML.parse(data) as ConfigData
  },
  serialize(data: ConfigData): string {
    return TOML.stringify(data)
  },
}
```



From a type-safety perspective, it's worth noting that we've used a few `as ConfigData` type assertions. While this can be convenient, it's not the safest approach since it essentially tells TypeScript, *“Trust me, this data matches the expected type,”* even though nothing prevents a configuration file from being manually modified in a way that breaks this assumption. For greater safety, a better approach would be to validate the data after deserialization to ensure it adheres to the expected schema. A library such as `zod` (nodejstip.link/zod) can help with this by providing runtime validation, preventing unexpected errors caused by malformed configuration files.



It's also important to note that our current implementation does not handle exceptions. If deserializing a string fails (for example, if the given file doesn't conform to the format specification), the exception from the specific deserialization library (JSON, YAML, or TOML) will simply bubble up. These exceptions can differ significantly, making upstream error handling more cumbersome. A more robust implementation might introduce a standard `DeserializationError`, forcing each formatter to catch format-specific exceptions and throw a unified error instead.

Now, to show you how everything comes together, let's create a file named `index.ts`, and let's try to load and save a sample configuration using different formats:

```
import { join } from 'node:path'
import { Config } from './config.ts'
import { jsonStrategy, tomlStrategy, yamlStrategy } from './strategies.ts'

const SAMPLES = join(import.meta.dirname, 'samples')

const jsonConfig = new Config(jsonStrategy)
await jsonConfig.load(join(SAMPLES, 'config.json'))
if (jsonConfig.data?.env) {
  jsonConfig.data.env.NODE_ENV = 'production'
  jsonConfig.data.env.NODE_OPTIONS = '--enable-source-maps'
}
await jsonConfig.save(join(SAMPLES, 'config_mod.json'))

const yamlConfig = new Config(yamlStrategy)
await yamlConfig.load(join(SAMPLES, 'config.yaml'))
if (yamlConfig.data?.env) {
  yamlConfig.data.env.NODE_ENV = 'production'
  yamlConfig.data.env.NODE_OPTIONS = '--enable-source-maps'
}
await yamlConfig.save(join(SAMPLES, 'config_mod.yaml'))

const tomlConfig = new Config(tomlStrategy)
await tomlConfig.load(join(SAMPLES, 'config.toml'))
if (tomlConfig.data?.env) {
  tomlConfig.data.env.NODE_ENV = 'production'
  tomlConfig.data.env.NODE_OPTIONS = '--enable-source-maps'
}
await tomlConfig.save(join(SAMPLES, 'config_mod.toml'))
```

Our test module reveals the core properties of the Strategy pattern. We defined only one `Config` class, which implements the common parts of our configuration manager. Then, by using different strategies for serializing and deserializing data, we created different `Config` class instances supporting different file formats.

The example we've just seen showed us only one of the possible alternatives that we had for selecting a strategy. Other valid approaches might have been the following:

- **Creating two different strategy families**, one for the deserialization and the other for the serialization: This would have allowed reading from one format and saving to another.
- **Dynamically selecting the strategy**: Depending on the extension of the file provided, the `Config` object could have maintained a map extension -> strategy and used it to select the right algorithm for the given extension.

As we can see, we have several options for selecting the strategy to use, and the right one only depends on your requirements and the trade-off in terms of features and the simplicity you want to obtain.

Furthermore, the implementation of the pattern itself can vary a lot as well. For example, in its simplest form, the context and the strategy can both be simple functions:

```
function context(strategy) {...}
```

Even though this may seem insignificant, it should not be underestimated in a programming language such as JavaScript, where functions are first-class citizens and used as much as fully-fledged objects.

Between all these variations, though, what does not change is the idea behind the pattern; as always, the implementation can slightly change, but the core concepts that drive the pattern are always the same.



The structure of the Strategy pattern may look similar to that of the Adapter pattern. However, there is a substantial difference between the two. The adapter object does not add any behavior to the adaptee; it just makes it available under another interface. This can also require some extra logic to be implemented to convert one interface into another, but this logic is limited to this task only. In the Strategy pattern, however, the context and the strategy implement two different parts of an algorithm, and therefore, both implement some kind of logic, and both are essential to build the final algorithm (when combined together).

In the wild

Passport.js (nodejsdp.link/passportjs) is a popular authentication library for Node.js. It's designed to streamline the integration of various authentication methods into your web applications. It cleverly employs the Strategy pattern to offer a flexible and extensible authentication system. At its core, Passport.js separates the core authentication workflow (handling sessions and managing user data) from the specific steps required for different authentication methods. For instance, one strategy might leverage OAuth to retrieve an access token for accessing Facebook or LinkedIn profiles. Another could simply validate a username/password combination against a local database. Passport.js treats each of these as a distinct strategy for achieving authentication.

This design allows Passport.js to support a wide range of authentication approaches. In fact, at the time of writing, their website lists over 500 different strategies (nodejssdp.link/passport-strategies), each available on npm as an installable plugin. This makes it easy to add new authentication methods to your application without modifying the core Passport.js library.

State

The **State** pattern is a specialization of the Strategy pattern where the strategy changes depending on the *state* of the context.

We have seen in the previous section how a strategy can be selected based on different variables, such as a configuration property or an input parameter, and once this selection is done, the strategy remains unchanged for the rest of the lifespan of the context object. In the State pattern, instead, the strategy (also called the **state** in this circumstance) is dynamic and can change during the lifetime of the context, thus allowing its behavior to adapt depending on its internal state.

The following figure shows us a representation of the pattern:

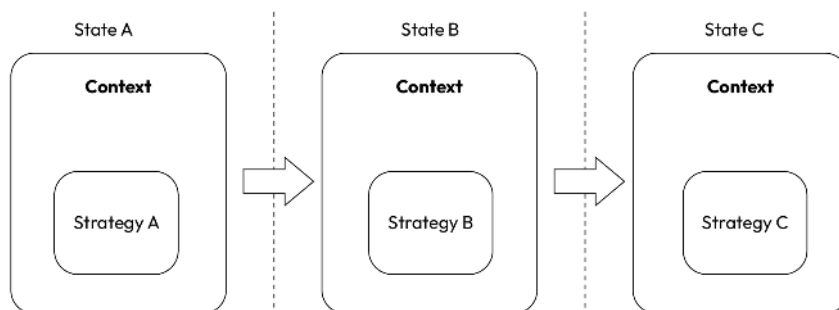


Figure 9.2: The State pattern

Figure 9.2 shows how a context object transitions through three states (A, B, and C). With the State pattern, at each different context state, we select a different strategy. This means that the context object will adopt a different behavior based on the state it's in.

To make this easier to understand, let's consider an example: imagine we have a hotel booking system and an object called `Reservation` that models a room reservation. This is a typical situation where we must adapt the behavior of an object based on its state.

Consider the following series of events:

- When the reservation is initially created, the user can confirm (using a method called `confirm()`) the reservation. Of course, they cannot cancel it (using `cancel()`), because it's still not confirmed (the caller would receive an exception, for example). They can, however, delete it (using `delete()`) if they change their mind before buying.
- Once the reservation is confirmed, using the `confirm()` method again does not make any sense; however, now it should be possible to cancel the reservation but no longer delete it, because it has to be kept for the records.

- On the day before the reservation date, it should not be possible to cancel the reservation anymore; it's too late for that.

Now, imagine that we have to implement the reservation system that we just described in one monolithic object. We can already picture all the `if...else` or `switch` statements that we would have to write to enable/disable each action depending on the state of the reservation.

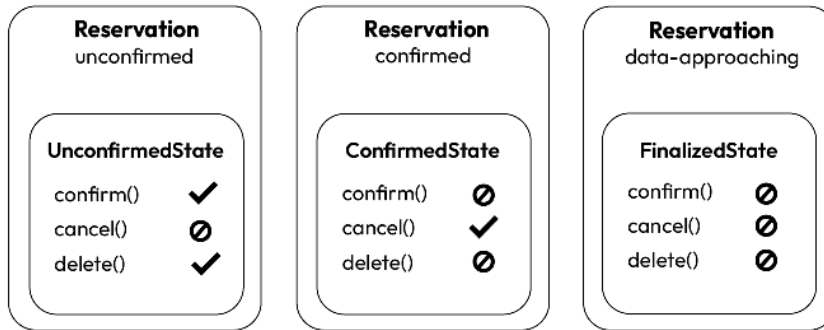


Figure 9.3: An example application of the State pattern

As illustrated in Figure 9.3, the State pattern is perfect in this situation: there would be three strategies, all implementing the three methods described (`confirm()`, `cancel()`, and `delete()`) and each one implementing only one behavior—the one corresponding to the modeled state. By using this pattern, it should be very easy for the `Reservation` object to switch from one behavior to another; this would simply require the **activation** of a different strategy (state object) on each state change.



The **state transition** can be initiated and controlled by the context object, by the client code, or by the state objects themselves. This last option usually provides the best results in terms of flexibility and decoupling, as the context does not have to know about all the possible states and how to transition between them.

Let's now work on a more concrete example so that we can apply what we learned about the State pattern.

Implementing a basic failsafe socket

Let's build a TCP client socket that does not fail when the connection with the server is lost; instead, we want to queue all the data sent during the time in which the server is offline and then try to send it again as soon as the connection is reestablished. We want to leverage this socket in the context of a simple monitoring system, where a set of machines sends some statistics about their resource utilization at regular intervals. If the server that collects these resources goes down, our socket will continue to queue the data locally until the server comes back online.

Let's start by creating a new module called `failsafeSocket.js` that defines our context object:

```
import { OfflineState } from './offlineState.js'
import { OnlineState } from './onlineState.js'

export class FailsafeSocket {
  constructor(options) { // 1
    this.options = options
    this.queue = []
    this.currentState = null
    this.socket = null
    this.states = {
      offline: new OfflineState(this),
      online: new OnlineState(this),
    }
    this.changeState('offline')
  }

  changeState(state) { // 2
    console.log(`Activating state: ${state}`)
    this.currentState = this.states[state]
    this.currentState.activate()
  }

  send(data) { // 3
    this.currentState.send(data)
  }
}
```

The `FailsafeSocket` class is made of three main elements:

1. The constructor initializes various data structures, including the queue that will contain any data sent while the socket is offline. Also, it creates a set of two states: one for implementing the behavior of the socket while it's offline, and another one when the socket is online.
2. The `changeState()` method is responsible for transitioning from one state to another. It simply updates the `currentState` instance variable and calls `activate()` on the target state.
3. The `send()` method contains the main functionality of the `FailsafeSocket` class. This is where we want to have a different behavior based on the offline/online state. As we can see, this is done by delegating the operation to the currently active state.

Let's now see what the two states look like, starting from the `offlineState.js` module:

```
import { createConnection } from 'node:net'

export class OfflineState {
  constructor(failsafeSocket) {
    this.failsafeSocket = failsafeSocket
  }

  send(data) { // 1
    this.failsafeSocket.queue.push(data)
  }

  activate() { // 2
    const retry = () => {
      setTimeout(() => this.activate(), 1000)
    }

    console.log(
      `Trying to connect (${this.failsafeSocket.queue.length} queued ` +
        `messages)`
    )
    this.failsafeSocket.socket = createConnection(
      this.failsafeSocket.options,
      () => {
        console.log('Connection established')
        this.failsafeSocket.socket.removeListener('error', retry)
        this.failsafeSocket.changeState('online')
      }
    )
    this.failsafeSocket.socket.once('error', retry)
  }
}
```

The module that we just created is responsible for managing the behavior of the socket while it's offline. This is how it works:

1. The `send()` method is only responsible for queuing any data it receives. We are assuming that we are offline, so we'll save those data objects for later. That's all we need to do here.
2. The `activate()` method tries to establish a connection with the server using the `createConnection()` function from the `node:net` core module. If the operation fails, it tries again after one second. It continues trying until a valid connection is established, in which case the state of `failsafeSocket` is transitioned to online.

Next, let's create the `onlineState.js` module, which is where we will implement the `OnlineState` class:

```
export class OnlineState {
  constructor(failsafeSocket) {
    this.failsafeSocket = failsafeSocket
  }

  send(data) { // 1
    this.failsafeSocket.queue.push(data)
    this._tryFlush()
  }

  async _tryFlush() { // 2
    try {
      let success = true
      while (this.failsafeSocket.queue.length > 0) {
        const data = this.failsafeSocket.queue[0]
        const flushed = await this._tryWrite(data)
        if (flushed) {
          this.failsafeSocket.queue.shift()
        } else {
          success = false
          break
        }
      }
      if (!success) {
        this.failsafeSocket.changeState('offline')
      }
    } catch (err) {
      console.error('Error during flush', err.message)
      this.failsafeSocket.changeState('offline')
    }
  }

  _tryWrite(data) { // 3
    return new Promise(resolve => {
      this.failsafeSocket.socket.write(data, err => {
        if (err) {
          console.error('Error writing data', err.message)
          resolve(false)
        } else {
          resolve(true)
        }
      })
    })
  }
}
```

```
        }  
      })  
    })  
  }  
  
  activate() { // 4  
    this._tryFlush()  
  }  
}
```

The `OnlineState` class models the behavior of the `FailsafeSocket` when there is an active connection with the server. This is how it works:

1. The `send()` method queues the data and then immediately tries to flush all the accumulated data into the socket, as it assumes that we are online. It'll use the internal `_tryFlush()` method to do that.
2. The `_tryFlush()` method attempts to send all messages currently in the queue. It's the core logic for ensuring that messages are eventually delivered when the socket is online. This method loops as long as there are messages in the queue; it retrieves the first message from the queue (FIFO: first-in, first-out) and then it attempts to write the message to the socket using the internal `_tryWrite()` method. This method returns a promise that resolves to either `true` (success) or `false` (failed to write). If the operation is successful, the current message is removed from the queue; otherwise, we set the local success flag to `false` and break the loop. After the loop completes, if the success flag is `false` (meaning at least one message failed to send), the `failsafeSocket.changeState('offline')` method is called. This transitions the `failsafeSocket` to the offline state if we failed to write because we have lost the connection with the server. In the case that there's an exception in this logic, we catch it, and we also assume the socket has gone offline. This prevents potential uncaught rejections.
3. The `_tryWrite()` method encapsulates the actual writing of data to the underlying socket, handling potential errors and resolving a promise to indicate success (`true`) or failure (`false`).
4. Finally, we have the `activate()` function. This method is called when the online state becomes the active state for the `failsafeSocket`. It ensures that any queued messages are immediately attempted to be sent by calling the internal `_tryFlush()`.

That's it for `FailsafeSocket`. Now that we've established the core structure, let's build a sample client and server to demonstrate the failsafe socket in action. However, because we're working with raw TCP sockets, we need to explicitly define a communication protocol. Unlike higher-level protocols such as HTTP, raw sockets transmit only a stream of bytes. To send meaningful, structured messages, we must agree on how those messages are encoded into a byte stream on the sender's side and decoded back into structured data on the receiver's side.

Our goal is to build a simple monitoring system where client machines periodically send resource utilization statistics to a central server. To structure these data transmissions, we need to define the shape of the data contained in every message. For our use case, we can use the following fields:

- **ts**: A Unix timestamp indicating the message generation time on the client.
- **client**: A string uniquely identifying the client. We'll generate this by concatenating the client's hostname and process ID (PID).
- **mem**: An object representing the output of `process.memoryUsage()`. This provides detailed memory usage information, including:
 - **rss** (resident set size): The amount of memory occupied by the process in RAM
 - **heapTotal**: The total size of the V8 JavaScript engine's heap for this process
 - **heapUsed**: The amount of the V8 heap currently in use
 - Other memory-related metrics, all measured in bytes

While we could transmit this information as a simple JSON-encoded message, that approach presents a challenge: the server wouldn't easily be able to distinguish individual messages within the continuous stream of bytes, as each message could have a different length.

To solve this, we'll use length-prefix framing. This technique involves structuring each message into two parts:

1. **Length field (4 bytes)**: A fixed-size field (4 bytes, encoded in Big Endian order) indicating the length, in bytes, of the subsequent data field
2. **Data field (variable length)**: The actual message data, encoded as a UTF-8 JSON string

This means that the server, upon receiving a stream of bytes from a client, will first read the initial 4 bytes to determine the length of the following message. It will then consume that many bytes to obtain the complete JSON message, which can then be deserialized. The process then repeats for the next incoming message.



Length-prefix framing provides a straightforward and reliable way to delineate variable-length messages over a network connection, and it's widely used. Some notable examples are **gRPC**, the Redis protocol (**RESP**), and **BitTorrent**.

Now that we have defined the protocol we want to use, we are ready to start working on our server implementation. We want to keep this implementation simple, so, for now, the only thing we will do is accept the connection, parse incoming messages, and print them in the standard output. Let's put the server code in a module named `server.js`:

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

const server = createServer(socket => {
  socket.on('error', err => {
    console.error('Server error', err.message)
```

```
    })

    // Accumulate incoming data
    let buffer = Buffer.alloc(0)

    // When a chunk of data is received
    socket.on('data', data => {
      // Append new data to the buffer
      buffer = Buffer.concat([buffer, data])

      // Ensure we have enough bytes for the length prefix
      while (buffer.length >= 4) {
        // Read the message length (Big Endian)
        const messageLength = buffer.readUInt32BE(0)

        if (buffer.length < 4 + messageLength) {
          // Not enough data yet; wait for more
          break
        }

        // Check if we have the complete message
        const message = buffer.subarray(4, 4 + messageLength).toString('utf8')

        // Process the message (just log it)
        console.log('Received message:', JSON.parse(message))

        // Remove the processed message from the buffer
        buffer = buffer.subarray(4 + messageLength)
      }
    })
  })

  // Start the server and listen on port 4545
  server.listen(4545, () => console.log('Server started'))
```

The preceding code implements a TCP server using the `createServer()` function from the `core node:net` module. A socket is established for each connection, enabling communication with clients. The server uses these sockets to receive and process JSON messages, adhering to the length-prefix framing protocol we defined. The comments within the code should offer a comprehensive explanation of how this protocol is implemented.

The last piece of the puzzle is the client-side code. This code is especially interesting because it shows how we can take advantage of the State pattern implemented within our failsafe socket to handle connection management and message sending. This code goes into `client.js`:

```
import { hostname } from 'node:os'
import { FailsafeSocket } from './failsafeSocket.js'

const clientId = `${hostname()}@${process.pid}` // 1
console.log(`Starting client ${clientId}`)

const failsafeSocket = new FailsafeSocket({ port: 4545 }) // 2

setInterval(() => { // 3
  // constructs the message
  const messageData = Buffer.from(
    JSON.stringify({
      ts: Date.now(),
      client: clientId,
      mem: process.memoryUsage(),
    }),
    'utf-8'
  )
  // creates a 4-byte buffer to store the message length
  const messageLength = Buffer.alloc(4)
  messageLength.writeUInt32BE(messageData.length, 0)
  // concatenates the message length and message data
  const message = Buffer.concat([messageLength, messageData])
  // sends the message
  failsafeSocket.send(message)
}, 5000)
```

This code is, perhaps unsurprisingly, simple. Since we have abstracted all the connectivity logic in our failsafe socket implementation (using the State pattern), here, we only have to worry about a few basic things:

1. Generate and print the client ID that will be used to identify this client with the server.
2. Create a new instance of our failsafe socket (which will immediately try to connect to the server on the given port).
3. Every 5 seconds, we create a message containing the memory information for the current client process and, using our length-prefix protocol, we send it through the failsafe socket.

To try the small system that we built, we should run both the client and the server, then we can test the features of `failsafeSocket` by stopping and then restarting the server. We should see that the state of the client changes between `online` and `offline` and that any memory measurement collected while the server is offline is queued and then resent as soon as the server goes back online.

This sample should be a clear demonstration of how the State pattern can help increase the modularity and readability of a component that has to adapt its behavior depending on its state.



The `FailsafeSocket` class that we built in this section is only for demonstrating the State pattern and is not intended to be a complete and 100% reliable solution for handling connectivity issues with TCP sockets. For example, we are not verifying that all the data written into the socket stream is received by the server, which would require some more code not strictly related to the pattern that we wanted to describe. For a production alternative, you can count on ZeroMQ (nodejsdp.link/zeromq). We'll talk about some patterns using ZeroMQ later in the book in *Chapter 13, Messaging and Integration Patterns*. Another shortcut we took is that we haven't included a mechanism to manage the queue's growth during prolonged disconnections. If a client remains disconnected for an extended period, the queue will continue to accumulate messages, increasing the process's memory footprint. If this persists, the client will eventually exhaust all the available memory and crash, resulting in the loss of all queued messages. There are several strategies that could be implemented to mitigate this issue, such as persisting the queue data to a file (or a local database) or implementing a capped queue size, dropping older messages once the maximum capacity is reached. However, implementing these solutions is outside the scope of this section and is left as an exercise for the interested reader. If you need a hint, check out the `cbuffer` (nodejsdp.link/cbuffer) or `tracked-queue` (nodejsdp.link/tracked-queue) library.

In the wild

`XState` (nodejsdp.link/xstate) is a popular library for creating state machines (a mathematical model of computation) in JavaScript and TypeScript. While state machines are a broader concept than the State design pattern, `XState` provides a powerful library that can also be used to implement the State pattern. Although `XState` is more commonly used in frontend projects due to its reactive features, `XState` is also usable in the context of Node.js projects. For example, `Mastra` (nodejsdp.link/mastra), an AI agent framework written in TypeScript, uses `XState` to model the behavior of an AI workflow as it transitions through the various states of its lifecycle (pending, executing, completed, failed, etc.).

Template

The next pattern that we are going to analyze is called **Template**, and it has a lot in common with the Strategy pattern. The Template pattern defines an abstract class that implements the skeleton (representing the common parts) of a component, where some of its steps are left undefined. Subclasses can then *fill* the gaps in the component by implementing the missing parts, called **template methods**. The intention of this pattern is to make it possible to define a family of classes that are all variations of a family of components. The following UML diagram shows the structure that we just described:

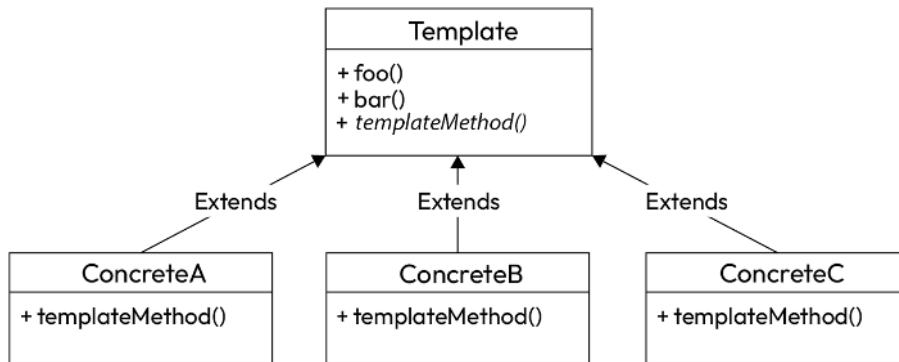


Figure 9.4: UML diagram of the Template pattern

The three concrete classes shown in *Figure 9.4* extend the template class and provide an implementation for `templateMethod()`, which is *abstract* or *pure virtual*, to use C++ terminology. In plain JavaScript, we don't have a formal way to define abstract classes, so all we can do is leave the method undefined or assign it to a function that always throws a runtime exception, indicating that the method must be implemented.



TypeScript allows you to annotate a class as `abstract`. When you do this, TypeScript will check at compile time that all the concrete implementations of the class are implementing all the necessary methods. In essence, JavaScript relies on discovering these errors during execution, whereas TypeScript proactively prevents them before the code even runs. More details are available in the official documentation at nodejsdp.link/ts-abstract.

The Template pattern can be considered a more traditionally object-oriented pattern than the other patterns we have seen so far, because inheritance is a core part of its implementation.

The purpose of the Template and Strategy patterns is very similar, but the main difference between the two lies in their structure and implementation. Both allow us to change the variable parts of a component while reusing the common parts. However, while Strategy allows us to do it *dynamically* at runtime, with Template, the complete component is determined the moment the concrete class is defined.

Under these assumptions, the Template pattern might be more suitable in those circumstances where we want to create prepackaged variations of a component. As always, the choice between one pattern and the other is up to the developer, who must consider the various pros and cons for each use case.

Let's now work on an example.

A configuration manager template

To have a better idea of the differences between Strategy and Template, let's now reimplement the Config object that we defined in the *Strategy* pattern section, but this time, using Template. As in the previous version of the Config object, we want to have the ability to load and save a set of configuration properties using different file formats.

Let's start by defining the template class. We will call it ConfigTemplate:

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

export class ConfigTemplate {
  async load(file) {
    console.log(`Deserializing from ${file}`)
    this.data = this._deserialize(await readFile(file, 'utf-8'))
  }

  async save(file) {
    console.log(`Serializing to ${file}`)
    await writeFile(file, this._serialize(this.data))
  }

  _serialize() {
    throw new Error('_serialize() must be implemented')
  }

  _deserialize() {
    throw new Error('_deserialize() must be implemented')
  }
}
```

The ConfigTemplate class implements the common parts of the configuration management logic—namely, setting and getting properties, plus loading and saving it to the disk. However, it leaves the implementation of _serialize() and _deserialize() open; those are in fact our template methods, which will allow the creation of concrete Config classes supporting specific configuration formats. The underscore at the beginning of the template methods' names indicates that they are for internal use only, an easy way to flag protected methods. Since, in JavaScript, we cannot declare a method as abstract, we simply define them as **stubs**, throwing an error if they are invoked (in other words, if they are not overridden by a concrete subclass).

Let's now create a concrete class using our template as an example, one that allows us to load and save the configuration using the JSON format:

```
// jsonConfig.js
import { ConfigTemplate } from './configTemplate.js'

export class JsonConfig extends ConfigTemplate {
  _deserialize(data) {
    return JSON.parse(data)
  }

  _serialize(data) {
    return JSON.stringify(data, null, '  ')
  }
}
```

The `JsonConfig` class extends our template class, `ConfigTemplate`, and provides a concrete implementation for the `_deserialize()` and `_serialize()` methods.

Similarly, we can implement a `YamlConfig` class supporting the `.yaml` file format using the same template class:

```
// yamlConfig.js
import { ConfigTemplate } from './configTemplate.js'
import YAML from 'yaml' // v2.7.0

export class YamlConfig extends ConfigTemplate {
  _deserialize(data) {
    return YAML.parse(data)
  }

  _serialize(data) {
    return YAML.stringify(data)
  }
}
```

Now, we can use our concrete configuration manager classes to load and save some configuration data:

```
// index.js
import { join } from 'node:path'
import { JsonConfig } from './jsonConfig.js'
import { YamlConfig } from './yamlConfig.js'

const SAMPLES = join(import.meta.dirname, 'samples')
```

```
const jsonConfig = new JsonConfig()
await jsonConfig.load(join(SAMPLES, 'config.json'))
jsonConfig.data.env.NODE_ENV = 'production'
jsonConfig.data.env.NODE_OPTIONS = '--enable-source-maps'
await jsonConfig.save(join(SAMPLES, 'config_mod.json'))

const yamlConfig = new YamlConfig()
await yamlConfig.load(join(SAMPLES, 'config.yaml'))
yamlConfig.data.env.NODE_ENV = 'production'
yamlConfig.data.env.NODE_OPTIONS = '--enable-source-maps'
await yamlConfig.save(join(SAMPLES, 'config_mod.yaml'))
```

Note the difference with the Strategy pattern: the logic for formatting and parsing the configuration data is *baked into* the class itself, rather than being chosen at runtime.

With minimal effort, the Template pattern allowed us to obtain a new, fully working configuration manager by reusing the logic and the interface inherited from the parent template class and providing only the implementation of a few abstract methods.



At this point, as a fun exercise, it should be very easy for you to implement a template that can deal with configuration in TOML format. You should be able to borrow most of the code from our Strategy pattern example, and if you get stuck, you can find a solution in our code repository.

In the wild

This pattern should not look entirely new to us. We already encountered it in *Chapter 6, Coding with Streams*, when we were extending the different stream classes to implement our custom streams. In that context, the template methods were the `_write()`, `_read()`, `_transform()`, or `_flush()` methods, depending on the stream class that we wanted to implement. To create a new custom stream, we needed to inherit from a specific abstract stream class, providing an implementation for the template methods.

Next, we are going to learn about a very important and ubiquitous pattern that is also built into the JavaScript language itself, which is the Iterator pattern.

Iterator

The **Iterator** pattern is a fundamental pattern, and it's so important and commonly used that it's usually built into the programming language itself. All major programming languages implement the pattern in one way or another, including, of course, JavaScript (starting from the ECMAScript 2015 specification).

The Iterator pattern defines a common interface or protocol for iterating over the elements of a container, such as an array, a stack, or a tree data structure. Usually, the algorithm for iterating over the elements of a container is different depending on the actual structure of the data. Think about iterating over an array versus traversing a tree. In the first situation, we need just a simple sequential loop; in the second, a more complex tree traversal algorithm is required (nodejsdp.link/tree-traversal). With the Iterator pattern, we hide the details about the algorithm being used or the structure of the data and provide a common interface for iterating over any type of container. In essence, the Iterator pattern allows us to decouple the implementation of the traversal algorithm from the way we consume the results (the elements) of the traversal operation.

In JavaScript, however, iterators work great even with other types of constructs, which are not necessarily containers, such as event emitters and streams. Therefore, we can say in more general terms that the Iterator pattern defines an interface to iterate over elements produced or retrieved in sequence.

The iterator protocol

In JavaScript, the Iterator pattern is implemented through **protocols** rather than through formal constructs, such as inheritance. This essentially means that the interaction between the implementer and the consumer of the Iterator pattern will communicate using interfaces and objects whose shape is agreed upon in advance.

The first thing we need to learn about is the **iterator protocol**. This protocol defines an interface for producing a sequence of values. The protocol iterator defines an **iterator object** as any object implementing a `next()` method with the following behavior: each time the method is called, the function returns the next element in the iteration wrapped in an object, called the **iterator result**, having two properties—`done` and `value`:

- `done` is set to `true` when the iteration is complete, or in other words, when there are no more elements to return. Otherwise, `done` will be `undefined` or `false`.
- `value` contains the current element of the iteration, and it can be left `undefined` if `done` is `true`. If `value` is set even when `done` is `true`, then it is said that `value` contains the **return value** of the iteration, a value which is not part of the elements being iterated, but it's related to the iteration itself as a whole (for example, the time spent iterating all the elements or the average of all the elements iterated if the elements are numbers).



Nothing prevents us from adding extra properties to the object returned by an iterator. However, those properties will be simply ignored by the built-in constructs or APIs consuming the iterator (we'll discuss those in a moment).

If we want to represent the Iterator type in TypeScript, we might define it as follows:

```
type Iterator<T> = {  
  next(): {  
    done: boolean  
    value: T  
  }  
}
```

This type definition is generic over `T`, meaning that an iterator can produce values of any specific type `T`. In other words, you could have an iterator that produces strings, another that produces numbers, or one that yields more complex objects. The key point is that the type remains consistent across successive invocations of the `next()` method on the same iterator.



Keep in mind that the actual TypeScript type for the `Iterator` is a bit more complex, as it captures several advanced features that aren't particularly relevant at this level.

Let's now focus on a quick example to demonstrate how to implement the iterator protocol. Let's implement a factory function called `createAlphabetIterator()`, which creates an iterator that allows us to traverse all the letters of the English alphabet. Such a function would look like this:

```
const A_CHAR_CODE = 'A'.charCodeAt(0)  
const Z_CHAR_CODE = 'Z'.charCodeAt(0)  
  
function createAlphabetIterator() {  
  let currCode = A_CHAR_CODE  
  
  return {  
    next() {  
      const currChar = String.fromCharCode(currCode)  
      if (currCode > Z_CHAR_CODE) {  
        return { done: true }  
      }  
  
      currCode++  
      return { value: currChar, done: false }  
    },  
  }  
}
```

The logic of the iteration is actually very straightforward; at each invocation of the `next()` method, we simply increment a number representing the letter's character code, convert it to a character, and then return it using the object shape defined by the iterator protocol.



It's not a requirement for an iterator to ever return `done: true`. In fact, there can be many situations in which an iterator is **infinite**. An example is an iterator that returns a random number at each iteration. Another example is an iterator that calculates a mathematical series, such as the Fibonacci series or the digits of the constant pi (as an exercise, you can try to convert the following algorithm to use iterators: nodejsdp.link/pi-js). Note that even if an iterator is theoretically infinite, it doesn't mean that it won't have computational or spatial limits. For example, the number returned by the Fibonacci sequence will get very big very soon.

The important aspect to note is that an iterator is very often a stateful object since we must keep track in some way of the current *position* of the iteration. In the previous example, we managed to keep the state in a closure (the `currCode` variable), but this is just one of the ways we can do so. We could have, for example, kept the state in an instance variable. This is usually better in terms of debuggability since we can read the status of the iteration from the iterator itself at any time, but on the other hand, it does not prevent external code from modifying the instance variable and hence tampering with the status of the iteration. It's up to you to decide the pros and cons of each option.

Iterators can indeed be fully stateless as well. Examples are iterators returning random elements and either completing randomly or never completing, and iterators stopping at the first iteration.

Now, let's see how we can use the iterator we just built. Consider the following code fragment:

```
const iterator = createAlphabetIterator()

let iterationResult = iterator.next()
while (!iterationResult.done) {
  console.log(iterationResult.value)
  iterationResult = iterator.next()
}
```

As we can see from the previous code, the code that consumes an iterator can be considered a pattern itself. It is effectively a loop that iterates until there are no more items available. However, we could call this the *low-level* way of using iterators. As we will see later in this section, this is not the only way we have to consume an iterator. In fact, JavaScript offers much more convenient and elegant ways to use iterators.



Iterators can optionally specify two additional methods: `return([value])` and `throw(error)`. The first is by convention used to signal to the iterator that the consumer has stopped the iteration before its completion, while the second is used to communicate to the iterator that an error condition has occurred. Both methods are rarely used by built-in iterators.

The iterable protocol

The **iterable protocol** defines a standardized means for an object to return an iterator. These kinds of objects are called **iterables**. Usually, an iterable is a container of elements, for example, a list, a set, a tree structure, and so on. It can also be an object virtually representing a set of elements, such as a Directory object, which would allow us to iterate over the files in a directory.

In JavaScript, we can define an iterable by making sure it implements the **@@iterator method**, or in other words, a method accessible through the built-in symbol `Symbol.iterator`.



The @@name convention indicates a *well-known* symbol according to the ES6 specification. To find out more, you can check out the relevant section of the ES6 specification at nodejsdp.link/es6-well-known-symbols.

Such an @@iterator method should return an iterator object, which can be used to iterate over the elements represented by the iterable. For example, if our iterable is a class, we would have something like the following:

```
class MyIterable {  
  // other methods...  
  [Symbol.iterator] () {  
    // return an iterator  
  }  
}
```

Or if we want to try to provide a TypeScript Iterable type, it could look like this:

```
type Iterable<T> = {  
  [Symbol.iterator](): Iterator<T>  
}
```

As you can see, this leverages the `Iterator` type we defined in the previous section. This should give you a clear idea of how the iterable protocol builds on top of the iterator protocol.



Once again, we have simplified things a bit. The actual TypeScript type definition for `Iterable` is a bit more complex than what is presented here, as it captures several advanced features that aren't particularly relevant right now.

To make this definition a bit more concrete, let's work through an example. We'll build a class to manage data organized in a two-dimensional matrix structure. We want this class to implement the iterable protocol, so that we can scan all the elements in the matrix using an iterator. Let's create a file called `matrix.js` containing the following content:

```
export class Matrix {
  constructor(inMatrix) {
    this.data = inMatrix
  }

  get(row, column) {
    if (row >= this.data.length || column >= this.data[row].length) {
      throw new RangeError('Out of bounds')
    }
    return this.data[row][column]
  }

  set(row, column, value) {
    if (row >= this.data.length || column >= this.data[row].length) {
      throw new RangeError('Out of bounds')
    }
    this.data[row][column] = value
  }

  [Symbol.iterator]() {
    let nextRow = 0
    let nextCol = 0

    return {
      next: () => {
        if (nextRow === this.data.length) {
          return { done: true }
        }

        const currVal = this.data[nextRow][nextCol]

        if (nextCol === this.data[nextRow].length - 1) {
          nextRow++
          nextCol = 0
        } else {
          nextCol++
        }
      }
    }
  }
}
```

```
        return { value: currVal }  
    },  
    }  
}  
}
```

As we can see, the class contains the basic methods for getting and setting values in the matrix, as well as the `@@iterator` method, implementing our iterable protocol. The `@@iterator` method will return an iterator, as specified by the iterable protocol, and such an iterator adheres to the iterator protocol. The logic of the iterator is very straightforward: we are simply traversing the matrix's cells from the top left to the bottom right, by scanning each column of each row. We are doing that by leveraging two indexes, `nextRow` and `nextCol`.

Now, it's time to try out our iterable `Matrix` class. We can do that in a file called `index.js`:

```
import { Matrix } from './matrix.js'  
  
const matrix2x2 = new Matrix([  
  ['11', '12'],  
  ['21', '22']  
)  
  
const iterator = matrix2x2[Symbol.iterator]()  
let iterationResult = iterator.next()  
while (!iterationResult.done) {  
  console.log(iterationResult.value)  
  iterationResult = iterator.next()  
}
```

All we are doing in the previous code is creating a sample `Matrix` instance and then obtaining an iterator using the `@@iterator` method. What comes next, as we already know, is just boilerplate code that iterates over the elements returned by the iterator. The output of the iteration should be `'11'`, `'12'`, `'21'`, and `'22'`.

Iterators and iterables as a native JavaScript interface

At this point, you may ask, “What’s the point of having all these protocols for defining iterators and iterables?” Well, having a standardized interface allows third-party code as well as the language itself to be modeled around the two protocols we’ve just seen. This way, we can have APIs (even native ones) as well as syntax constructs accepting iterables as input.

For example, the most obvious syntax construct accepting an iterable is the `for...of` loop. We've just seen in the last code sample that iterating over a JavaScript iterator is a pretty standard operation, and its code is mostly boilerplate. In fact, we'll always have an invocation to `next()` to retrieve the next element and a check to verify if the `done` property of the iteration result is set to `true`, which indicates the end of the iteration. But, worry not, simply pass an iterable to the `for...of` instruction to seamlessly loop over the elements returned by its iterator. This allows us to process the iteration with an intuitive and compact syntax:

```
for (const element of matrix2x2) {  
  console.log(element)  
}
```

Another construct compatible with iterables is the spread operator:

```
const flattenedMatrix = [...matrix2x2]  
console.log(flattenedMatrix)
```

Similarly, we can use an iterable with the destructuring assignment operation:

```
const [oneOne, oneTwo, twoOne, twoTwo] = matrix2x2  
console.log(oneOne, oneTwo, twoOne, twoTwo)
```

The following are some JavaScript built-in APIs accepting iterables:

- `Map([iterable]):` nodejsdp.link/map-constructor
- `WeakMap([iterable]):` nodejsdp.link/weakmap-constructor
- `Set([iterable]):` nodejsdp.link/set-constructor
- `WeakSet([iterable]):` nodejsdp.link/weakset-constructor
- `Promise.all(iterable):` nodejsdp.link/promise-all
- `Promise.race(iterable):` nodejsdp.link/promise-race
- `Array.from(iterable):` nodejsdp.link/array-from

On the Node.js side, one notable API accepting an iterable is `stream.Readable.from(iterable, [options])` (nodejsdp.link/readable-from), which creates a readable stream out of an iterable object.

JavaScript itself defines many iterables that can be used with the APIs and constructs we've just seen. The most notable iterable is `Array`, but also other data structures, such as `Map`, `Set`, and even `String`, all implement the `@@iterator` method. In Node.js land, `Buffer` is another notable iterable object.



A simple trick to create a copy of an array without duplicate elements in it is the following: `const uniqArray = Array.from(new Set(arrayWithDuplicates))`. This snippet shows us how iterables offer a way for different components to talk to each other using a shared interface.

Implementing the iterable protocol on iterators

The functions described in the previous section accept an iterable object as input, not an iterator. This raises a question: What can we do if we have a function returning an iterator, such as our `createAlphabetIterator()` example? Is there a way to make such an iterator object interoperable with built-in APIs and syntax constructs designed for iterables? Well, yes and no! No, because to be directly interoperable, the object needs to implement the iterable protocol. Yes, because nothing prevents us from *also* implementing the iterable protocol on an object that is already an iterator (like the ones produced by `createAlphabetIterator()`). In fact, the two protocols are not mutually exclusive; they can coexist on the same JavaScript object. Let's explore how to achieve this by updating our `createAlphabetIterator()` example:

```
const A_CHAR_CODE = 'A'.charCodeAt(0)
const Z_CHAR_CODE = 'Z'.charCodeAt(0)

function createAlphabetIterableIterator() {
  let currCode = A_CHAR_CODE

  return {
    next() {
      const currChar = String.fromCharCode(currCode)
      if (currCode > Z_CHAR_CODE) {
        return { done: true }
      }

      currCode++
      return { value: currChar, done: false }
    },
    [Symbol.iterator]() {
      return this
    },
  }
}
```

Can you spot what's changed? Yes, we only added three lines of code in the returned object:

```
[Symbol.iterator]() {
  return this
}
```

Why does this work? Remember the definition of the iterable protocol? An object is iterable if it implements the `@@iterator` method to return an iterator object. Well, we have originally designed the `createAlphabetIterator()` to return an iterator object, so implementing the `@@iterator` method is as simple as returning a reference to itself (using `this`).

To prove that this change works, we can try to use our new function with a `for...of` loop to print all the letters in the alphabet, each one on a separate line:

```
for (const letter of createAlphabetIterableIterator()) {  
  console.log(letter)  
}
```

Or use it with the spread syntax to accumulate the letters in an array:

```
const letters = [...createAlphabetIterableIterator()]  
console.log(letters)
```

If you are still struggling to grasp the conceptual difference between an iterable and an iterator, you can think of them this way:

- **Iterable:** An object that represents a collection of items that you can iterate on
- **Iterator:** An object that allows you to move from one item to the next in a collection

Can you see the natural correlation between these two definitions? An object is iterable if you can get an iterator from it. If it's already an iterator, you can make it iterable by simply returning a reference to itself in the `@@iterator` method.

Implementing both protocols is generally a good practice because it allows the users of our code to pick the interface that is more suitable for them.

Iterator utilities

Remember how we discussed that, in JavaScript, the Iterator pattern is primarily implemented through **protocols** (the `next()` method) rather than formal constructs such as inheritance? Well, while that's still fundamentally true, recent JavaScript versions have introduced an `Iterator` prototype. This prototype can be extended to create iterator classes that offer additional benefits.

To see a practical example, let's implement a `RangeIterator` class that allows us to iterate over a range of integer values. This kind of utility exists in the standard library of many programming languages, but not in JavaScript nor in Node.js (although there is an ECMAScript proposal to introduce it as a first-class citizen in the language, currently in stage 2 of the acceptance process: nodejsdp.link/proposal-range). Here is the code:

```
class RangeIterator extends Iterator {  
  #start  
  #end  
  #step  
  #current  
  
  constructor(start, end, step = 1) {  
    super()  
    this.#start = start
```

```
    this.#end = end
    this.#step = step
    this.#current = undefined
  }

  next() {
    this.#current =
      this.#current === undefined ? this.#start : this.#current +
      this.#step

    if (
      this.#step > 0 ? this.#current < this.#end : this.#current >
      this.#end
    ) {
      return { done: false, value: this.#current }
    }

    return { done: true }
  }
}
```

If you look closely at the implementation of the `next()` method, you should be able to appreciate that every instance of this class is an iterator object.

Let's see how we can use this new class to enumerate all the positive numbers from 1 to 5, as follows:

```
const range = new RangeIterator(1, 6)
let iterationResult = range.next()
while (!iterationResult.done) {
  console.log(iterationResult.value)
  iterationResult = range.next()
}
```

As you might expect, this prints:

```
1
2
3
4
5
```



Note that our implementation is inclusive on the left side of the range (1 is included) and exclusive on the right side (6 is not included). This is just a common implementation detail. You are welcome to try to make this class more configurable to also support fully inclusive ranges if you like.

The important implementation detail is that, other than implementing the iterator protocol (by defining the `next()` method), we are also extending the `Iterator` prototype. This comes with a few interesting advantages:

- **Type checking:** You can easily check whether an object is an `Iterator` instance using `instanceof`. This provides a reliable way to verify the iterator's type at runtime (`range instanceof Iterator` evaluates to `true`).
- **Helper methods:** The `Iterator` prototype provides a standard location to add built-in and custom helper methods to all your iterators. We will discuss some of the built-in ones later in this section.
- **Conversion and interoperability:** You can easily convert a protocol-based iterator (an object that only implements `next()`) to an `Iterator` instance using the static `Iterator.from()` method, gaining access to any helper methods that may be available.
- **Iterable iterators:** The `Iterator` prototype conveniently implements the iterable protocol behind the scenes. Now, you should understand that this simply means that the base `Iterator` class has an `@@iterator` method that simply returns `this`. Therefore, all the instances of our `Iterator` class are not just iterators but also iterable objects. So we can use the `for...of` syntax, the spread operator, and all the other standard utilities that work with iterable objects. If you need an example, try running `const numbers = [...new RangeIterator(1,6)]`.

Now, let's explore the iterator utilities provided directly by the `Iterator` prototype. Currently, the most prevalent ones are likely `Iterable.map()`, `Iterable.filter()`, and `Iterable.reduce()`, which may sound familiar if you've worked with their counterparts on the `Array` prototype. However, significant differences exist between the two sets of methods, as we'll demonstrate shortly.



For a comprehensive list of available methods, consult the MDN documentation at nodejsdp.link/iterator.

To illustrate the behavior of these methods, let's use our `RangeIterator` class to create an iterator of numbers ranging from 0 to 10. We'll then filter out the odd numbers and double the remaining even numbers:

```
const zeroToTen = new RangeIterator(0, 10)
const doubledEven = zeroToTen
  .filter(n => n % 2 === 0)
  .map(n => n * 2)
  .toArray()
console.log(doubledEven)
```


As you might expect, this example prints [0, 4, 8, 12, 16]. But why do we need to explicitly call the `toArray()` helper at the end?

If we try to remove the `toArray()` call and execute our code again, it will output:

```
Object [Iterator Helper] {}
```

This is because `filter()` and `map()` (as well as most other Iterator helper methods) are lazy. They don't perform any computation until you attempt to consume data from the resulting iterator. Instead, they construct a processing pipeline (another iterator) that performs the necessary computations only when its values are requested. In essence, these methods allow you to define a series of transformations that are executed on demand, rather than immediately.

Another way to appreciate this lazy behavior is to examine the step-by-step execution of the iterator pipeline. Consider this code:

```
const zeroToTenIt = new RangeIterator(0, 10)
const doubledEvenIt = zeroToTenIt
  .filter(n => n % 2 === 0)
  .map(n => n * 2)
console.log(doubledEvenIt.next()) // { done: false, value: 0 }
console.log(doubledEvenIt.next()) // { done: false, value: 4 }
```

Let's examine what happens after every line is executed:

```
const zeroToTenIt = new RangeIterator(0, 10);
```

This creates a new `RangeIterator` instance that will generate numbers from 0 up to (but not including) 10. No values are actually generated at this point; the iterator is simply initialized.

```
const doubledEvenIt = zeroToTenIt
  .filter(n => n % 2 === 0)
  .map(n => n * 2);
```

This chains the `filter()` and `map()` methods onto the `zeroToTenIt` iterator. Again, no filtering or mapping operations are performed at this stage. Instead, a new iterator (`doubledEvenIt`) is created, representing the composition of these operations. This new iterator knows how to pull values from the `zeroToTenIt` iterator, filter them, and then map them, but it only does so when explicitly asked for a value.

```
console.log(doubledEvenIt.next()); // { done: false, value: 0 }
```

This is where the magic starts to happen. Calling `next()` on the `doubledEvenIt` iterator triggers the execution of the processing pipeline for a single value:

1. The `doubledEvenIt` iterator requests the first value from `zeroToTenIt`.
2. `zeroToTenIt` generates the value 0.
3. `doubledEvenIt`'s filter method checks whether 0 is even (`0 % 2 === 0` is true).

4. Since 0 is even, `doubledEvenIt`'s `map()` method doubles it ($0 * 2 = 0$).
5. The `next()` method returns `{ done: false, value: 0 }` as shown here:

```
console.log(doubledEvenIt.next()); // { done: false, value: 4 }
```

The second call to `next()` repeats the process:

1. `doubledEvenIt` requests the next value from `zeroToTenIt`.
2. `zeroToTenIt` generates the value 1.
3. `doubledEvenIt`'s `filter` method checks whether 1 is even ($1 \% 2 === 0$ is false).
4. Since 1 is not even, the `filter` method discards it and requests the next value from `zeroToTenIt`.
5. `zeroToTenIt` generates the value 2.
6. `doubledEvenIt`'s `filter` method checks whether 2 is even ($2 \% 2 === 0$ is true).
7. Since 2 is even, `doubledEvenIt`'s `map` method doubles it ($2 * 2 = 4$).
8. The `next()` method returns `{ done: false, value: 4 }`.

Hopefully, you now have a clear understanding of what we mean by “lazy” iterator helper methods and how they can be used to construct expressive and efficient processing pipelines. It’s important to remember that we don’t need to consume the entire iterator (unless we explicitly choose to, for example, by calling `toArray()`). This is precisely what makes this lazy approach so convenient: no computation is performed unless we explicitly request it by attempting to consume the iterator.

Now, how does this compare with similar `Array` helpers? Let’s contrast the lazy evaluation of `Iterator` prototype helpers with the eager evaluation of their `Array` counterparts. Consider the following example, mirroring the previous one:

```
const numbersArray = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
const doubledEvenArray = numbersArray
  .filter(n => n % 2 === 0)
  .map(n => n * 2)
console.log(doubledEvenArray)
```

Let’s see what happens at every step:

```
const numbersArray = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The array `numbersArray` is created, containing the integers from 0 to 9. All these numbers are loaded in memory.

```
const doubledEvenArray = numbersArray
  .filter(n => n % 2 === 0)
  .map(n => n * 2)
```

This is where the eager evaluation takes place:

1. The `filter()` method iterates over each element in `numbersArray`. For each element `n`, it evaluates the condition `n % 2 === 0` (is `n` even?). If the condition is true, the element is included in a new array (also loaded into memory). If the condition is false, the element is excluded. This results in a new array containing only the even numbers: `[0, 2, 4, 6, 8]`.
2. The `map()` method then iterates over each element in the new array created by `filter` (i.e., `[0, 2, 4, 6, 8]`). For each element `n`, it applies the mapping function `n * 2` (doubles the number). This results in another new array, `[0, 4, 8, 12, 16]`, where each element is the double of the corresponding even number.
3. Finally, this second new array, `[0, 4, 8, 12, 16]`, is assigned to the variable `doubledEvenArray`.

In the final step, we just print the resulting array into the console.

While the result of this array-based approach might be the same as the iterator-based approach, there are some substantial (and very important) differences that we must highlight:

- **Eager evaluation, executed immediately:** With Array's `filter()` and `map()` methods, the moment the JavaScript engine encounters that line of code, it performs the computation. The filtering and mapping operations are performed right there and then. This is what we mean by “eager” evaluation: the work is done upfront.
- **Intermediate arrays:** A critical consequence of eager evaluation is the creation of intermediate arrays. In our example, `filter()` produces one new array containing the even numbers, and then `map()` creates another new array containing the doubled even numbers. These arrays are held in memory, demanding space directly proportional to their size. But what if, instead of a few numbers, we had millions? This seemingly simple pipeline would then allocate three (slightly different) copies of the array, each containing millions of records. This isn't merely a minor CPU inefficiency; it can rapidly escalate into a major problem, potentially exhausting all available memory and causing the program to crash. The memory saturation can easily result in out-of-memory errors, and even if the memory is available, copying millions of records multiple times takes a significant amount of CPU processing. This makes eager evaluation unsuitable for large datasets or situations where memory is constrained.
- **All elements processed:** Perhaps the most significant drawback is that Array's `filter()` and `map()` always process every element in the input array, regardless of whether you actually need all the resulting values. Imagine you had an array with a million values, but you only needed the first 10 elements of the transformed result. The Array approach would still perform the filtering and mapping operations on all million values, potentially wasting a significant amount of CPU time and memory.

The good news is that you can easily convert an array-based pipeline into an iterator-based one by using `Iterator.from()`:

```
Iterator.from([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  .filter(n => n % 2 === 0)
  .map(n => n * 2)
```

This code will generate an efficient lazy iterator-based pipeline that you can consume as needed. `Iterator.from()` accepts any iterable object (and arrays are iterable by default).

If you're working with arrays, you can also use `Array.prototype.values()` to get an iterator over the array's values:



```
const arr = [10, 20, 30]
const iterator = arr.values()
console.log(iterator.next()) // { value: 10, done: false }
```

This is a convenient and direct way to get an iterator from an array. However, `Iterator.from()` is a more generic and flexible solution. It works not only with arrays but also with any object that implements the iterable or iterator protocol, including custom iterators returned by libraries or other data structures.

Generators

The ES2015 specification introduced a syntax construct closely related to iterators: **generators**, also known as **semicoroutines** or **generator functions**. Generators generalize standard JavaScript functions by allowing multiple entry points.

In a standard JavaScript function, there's only one entry point: the function invocation itself. Once a function is invoked, the function code executes sequentially until the end (or until a return statement is reached or an error is thrown).

The execution of a generator, however, can be suspended at any point using the `yield` statement and resumed later, starting from the line following the last `yield` statement that suspended execution. Generators are particularly well suited for implementing iterable collections. In fact, as we'll see shortly, the **generator object** returned by a generator function is both an iterator and an iterable.

Generators in theory

To define a **generator function**, we need to use the `function*` declaration (the `function` keyword followed by an asterisk):

```
function* myGenerator() {
  // generator body
}
```

Invoking a generator function doesn't execute its body immediately. Instead, it returns a **generator object**, which, as we've mentioned, is both an iterator and an iterable. However, that's just the beginning. Calling `next()` on the generator object starts or resumes execution of the generator until it encounters a `yield` statement or reaches the end (either implicitly or explicitly with a return statement).

Inside the generator function, using `yield x` is equivalent to returning `{ done: false, value: x }` from an iterator. If the generator function returns a value `x`, it's equivalent to returning `{ done: true, value: x }`.

A simple generator function

To demonstrate what we just learned, let's see a simple generator called `fruitGenerator()`, which will yield two names of fruits and return their ripening season:

```
function* fruitGenerator () {  
  yield 'peach'  
  yield 'watermelon'  
  return 'summer'  
}  
  
const fruitGeneratorObj = fruitGenerator()  
console.log(fruitGeneratorObj.next()) // 1  
console.log(fruitGeneratorObj.next()) // 2  
console.log(fruitGeneratorObj.next()) // 3
```

The preceding code will print the following text:

```
{ value: 'peach', done: false }  
{ value: 'watermelon', done: false }  
{ value: 'summer', done: true }
```

This is a short explanation of what happened:

1. The first time `fruitGeneratorObj.next()` was invoked, the generator started its execution until it reached the first `yield` command, which put the generator on pause and returned the value `'peach'` to the caller.
2. At the second invocation of `fruitGeneratorObj.next()`, the generator resumed, starting from the second `yield` command, which, in turn, put the execution on pause again, while returning the value `'watermelon'` to the caller.
3. The last invocation of `fruitGeneratorObj.next()` caused the execution of the generator to resume from its last instruction, a `return` statement, which terminates the generator, returns the value `'summer'`, and sets the `done` property to `true` in the result object.

Since a generator object is also an iterable, we can use it in a `for...of` loop, as in this example:

```
for (const fruit of fruitGenerator()) {  
  console.log(fruit)  
}
```

The preceding loop will print:

```
peach  
watermelon
```



Why is summer not being printed? Well, summer is not yielded by our generator, but instead, it is returned, which indicates that the iteration is complete with summer as a return value (not as an element).

Controlling a generator iterator

A lesser-known feature of iterators, which we have yet to discuss, is that the `next()` method can accept an optional argument: a value that becomes available inside the body of the `next()` function and can be used to alter the method's original behavior. This allows passing extra context between successive calls to `next()`, enabling a form of two-way communication between the iterator and its consumer.

For example, think about our earlier `RangeIterator`. In theory, we could modify its `next()` function to accept a value that overrides the step parameter, allowing the consumer to skip a number of elements dynamically. This would create a mechanism where each call to `next()` not only retrieves a value from the iterator but also influences its behavior. While this adds flexibility, it is rarely used in practice.

It's worth noting that generators (which are effectively an abstraction over iterators) provide the same functionality. The `next()` method can accept an optional value, which becomes the result of the corresponding `yield` expression inside the generator function.

To illustrate this, consider the following example:

```
function* twoWayGenerator() {  
  const who = yield null  
  yield `Hello ${who}`  
}  
  
const twoWay = twoWayGenerator()  
twoWay.next()  
console.log(twoWay.next('world'))
```

When executed, the preceding code prints `Hello world`. This means that the following has happened:

1. The first time the `next()` method was invoked, the generator reached the first `yield` statement and was then put on pause.
2. When `next('world')` was invoked, the generator resumed from the point where it was put on pause, which is on the `yield` instruction, but this time, we had a value that was passed back to the generator. This value was then set to the `who` variable. The generator then appended the `who` variable to the string `'Hello '` and yielded the result.

Two other extra features provided by generator objects are the optional `throw()` and `return()` iterator methods. The first behaves like `next()` but it will also throw an exception within the generator as if it was thrown at the point of the last `yield`, and returns the canonical iterator object with the `done` and `value` properties. The second, the `return()` method, forces the generator to terminate and returns an object such as the following: `{done: true, value: returnArgument}`, where `returnArgument` is the argument passed to the `return()` method.

The following code shows a demonstration of these two methods:

```
function* twoWayGenerator() {
  try {
    const who = yield null
    yield `Hello ${who}`
  } catch (err) {
    yield `Hello error: ${err.message}`
  }
}

console.log('Using throw():')
const twoWayException = twoWayGenerator()
twoWayException.next()
console.log(twoWayException.throw(new Error('Boom!')))

console.log('Using return():')
const twoWayReturn = twoWayGenerator()
console.log(twoWayReturn.return('myReturnValue'))
```

Running the previous code will print the following to the console:

```
Using throw():
{ value: 'Hello error: Boom!', done: false }
Using return():
{ value: 'myReturnValue', done: true }
```

As we can see, the `twoWayGenerator()` function will receive an exception as soon as the first `yield` instruction returns. This works exactly as if an exception were thrown from inside the generator, and this means that it can be caught and handled like any other exception using a `try...catch` block. The `return()` method, instead, will simply stop the execution of the generator, causing the given value to be provided as a return value by the generator.

How to use generators in place of iterators

Generator objects are also iterators. This means that generator functions can be used to implement the `@@iterator` method of iterable objects. To demonstrate this, let's convert our previous `Matrix` iteration example to generators. Let's update our `matrix.js` file as follows:

```
export class Matrix {
  // ... rest of the methods (unchanged)

  *[Symbol.iterator]() {
    for (const row of this.data) {
      for (const cell of row) {
```

```
        yield cell
    }
}
}
```

There are a few interesting aspects in the code fragment we've just seen. Let's analyze them in more detail:

1. The first thing to notice is that the `@@iterator` method is now a generator (note the asterisk, `*`, before the method name).
2. The variables we use to maintain the state of the iteration are now just local variables (`row` and `cell`) for the generator, while in the previous version of the `Matrix` class, those two variables were part of a closure. This is possible because when a generator is invoked, its local state is preserved between reentries.
3. We are using two nested loops to iterate over the elements of the matrix (`rows` and `cells`, respectively). This is certainly more intuitive than trying to imagine a loop that invokes the `next()` method of an iterator, and we don't need to do all the book-keeping with indices that we were doing in the previous implementation.

As we can see, generators are an excellent alternative to writing iterator and iterable objects from scratch. They will improve the readability of our iteration routine while offering the same level of functionality.

Wait! There is one more trick that generators offer us to make this code even more concise. We are talking about the **generator delegation** syntax: `yield * iterable`. This is another example of a JavaScript built-in syntax accepting an iterable as an argument. The instruction will loop over the elements of the iterable and yield each element one by one. It's effectively a convenient syntactic sugar to deal with nested iterators. So, if we want to leverage this, we could simplify the iteration logic of our `Matrix` class even more:

```
export class Matrix {
  // ... rest of the methods (unchanged)

  *[Symbol.iterator]() {
    for (const row of this.data) {
      yield* row
    }
  }
}
```


Look at that! No more nested loops! Now, can we do even better and remove even the remaining loop? Look at this code:

```
export class Matrix {  
  // ... rest of the methods (unchanged)  
  
  *[Symbol.iterator]() {  
    yield* this.data.flat()  
  }  
}
```

Sweet, *loop-less*!

Note that here we are leveraging the `Array.flat()` function to turn a bi-dimensional array into a flat array. Then, we use the generator delegation syntax to yield all its elements, one by one.

Async iterators

The iterators we've seen so far return a value synchronously from their `next()` method. However, in JavaScript (and especially in Node.js), it's very common to have iterations over collections that require an asynchronous operation to be completed before an item can be produced.

Imagine, for example, iterating over the requests received by an HTTP server, or over the results of an SQL query, or over the elements of a paginated REST API. In all those situations, it would be handy to be able to return a promise from the `next()` method of an iterator, or even better, use the `async/await` construct.

Well, that's exactly what **async iterators** are; they are iterators returning a promise when you call `next()`. Since that's the only extra requirement, it means that we can simply define `next()` as an `async` function. Similarly, **async iterables** are objects that implement an `@@asyncIterator` method, or in other words, a method accessible through the `Symbol.asyncIterator` key, which returns (synchronously) an async iterator.

You can loop over the items of async iterables using the special `for await...of` syntax, which can only be used inside an `async` context. With the `for await...of` syntax, we are essentially implementing a sequential asynchronous execution flow on top of the Iterator pattern. Essentially, if we have the following code:

```
for await (const value of iterable) {  
  console.log(value);  
}
```

It is just syntactic sugar for the following loop:

```
const asyncIterator = iterable[Symbol.asyncIterator]()  
let iterationResult = await asyncIterator.next()  
while (!iterationResult.done) {  
  console.log(iterationResult.value)  
}
```

```

    iterationResult = await asyncIterator.next()
  }

```

This means that the `for await...of` syntax can also be used to iterate over a simple iterable (not just async iterables), such as over an array of promises. It will work even if not all (or none of) the elements of the iterator are promises.

To see a more practical use case for async iterables and the `for await...of` syntax, let's build a class that takes a list of URLs as input and allows us to iterate over their availability status (up/down). Let's call the class `CheckUrls`:

```

// checkUrls.js
export class CheckUrls {
  #urls

  constructor(urls) {
    this.#urls = urls // 1
  }

  [Symbol.asyncIterator]() {
    const urlsIterator = Iterator.from(this.#urls) // 2

    return {
      async next() { // 3
        const iteratorResult = urlsIterator.next() // 4
        if (iteratorResult.done) {
          return { done: true }
        }

        const url = iteratorResult.value
        try {
          const checkResult = await fetch(url, { // 5
            method: 'HEAD',
            redirect: 'follow',
            signal: AbortSignal.timeout(5000),
          })
          if (!checkResult.ok) {
            return { // 5a
              done: false,
              value: `${url} is down, error: ${checkResult.status} ` +
                `${checkResult.statusText}`,
            }
          }
        }
      }
    }
  }
}

```

```

        return { // 5b
            done: false,
            value: `${url} is up, status: ${checkResult.status}`,
        }
    } catch (err) {
        return { // 5c
            done: false,
            value: `${url} is down, error: ${err.message}`,
        }
    }
},
}
}
}

```

Let's analyze the previous code's most important parts:

1. The `CheckUrls` class constructor takes as input a list of URLs. This list can be an array or any other iterable object.
2. In our `@@asyncIterator` method, we obtain an iterator from the `this.#urls` object, which, as we just said, should be an iterable. We can do that by simply using the `Iterator.from()` helper.
3. Note how the `next()` method is now an `async` function. This means that it will always return a promise, as requested by the `async` iterable protocol.
4. In the `next()` method, we use the `urlsIterator` to get the next URL in the list, unless there are no more, in which case we simply return `{done: true}`.
5. This is where the asynchronous nature of the check comes to life. The `await` keyword pauses execution until the `fetch()` call completes, sending a HEAD request to the current URL and retrieving the response. The `fetch()` function is configured to follow redirects (`redirect: 'follow'`) and to abort after 5 seconds (`signal: AbortSignal.timeout(5000)`). The logic then handles three potential outcomes:
 - a. **Request failure (error):** If the `fetch()` operation encounters an error (e.g., a network connection issue or a timeout), it will throw an exception. This exception is caught by the `catch` block. The code then constructs an error result indicating that the URL is down, including the specific error message from the exception (`err.message`).
 - b. **Successful response (OK):** If `fetch()` receives a successful response and `checkResult.ok` is `true`, the code constructs a success result, indicating that the URL is up and providing the HTTP status code.

- c. **Successful response (but not OK):** If `fetch()` successfully receives a response from the server, the code checks the `checkResult.ok` property. This property is true if the HTTP status code is in the 200–299 range (indicating success), and false otherwise. If `checkResult.ok` is false, the code interprets this as a problem (e.g., a 404 Not Found or a 500 Internal Server Error). It constructs a result indicating that the URL is down, including the HTTP status code and status text from the response.

Now, let's use the `for await...of` syntax we mentioned earlier to iterate over a `CheckUrls` object:

```
import { CheckUrls } from './checkUrls.js'

const checkUrls = new CheckUrls([
  'https://nodejsdesignpatterns.com',
  'https://example.com',
  'https://mustbedownforsurehopefully.com',
  'https://loige.co',
  'https://mario.fyi',
  'https://httpstat.us/200',
  'https://httpstat.us/301',
  'https://httpstat.us/404',
  'https://httpstat.us/500',
  'https://httpstat.us/200?sleep=6000',
])

for await (const status of checkUrls) {
  console.log(status)
}
```

At the time of writing, this is the output we see if we run this code:

```
https://nodejsdesignpatterns.com is up, status: 200
https://example.com is up, status: 200
https://mustbedownforsurehopefully.com is down, error: fetch failed
https://loige.co is up, status: 200
https://mario.fyi is up, status: 200
https://httpstat.us/200 is up, status: 200
https://httpstat.us/301 is up, status: 200
https://httpstat.us/404 is down, error: 404 Not Found
https://httpstat.us/500 is down, error: 500 Internal Server Error
https://httpstat.us/200?sleep=6000 is down, error: The operation was aborted
due to timeout
```

Of course, you could get a different response depending on your connectivity and the current availability of the given URLs.

As we can see, the `for await...of` syntax is a very intuitive way to iterate over an async iterable, and as we will see in a while, it can be used in conjunction with some interesting built-in iterables to obtain alternative new ways to access asynchronous information.



The `for await...of` loop (as well as its synchronous version) will call the optional `return()` method of the iterator if it's prematurely interrupted with a `break`, a `return`, or an exception. This can be used to immediately perform any cleanup task that would usually be performed when the iteration completes.

Async generators

As well as async iterators, we can also have **async generators**. To define an **async generator function**, simply prepend the keyword `async` to the function definition:

```
async function* generatorFunction() {
  // ...generator body
}
```

As you can imagine, async generators allow the use of the `await` instruction within their body, and the return value of their `next()` method is a promise that resolves to an object having the canonical `done` and `value` properties. This way, **async generator objects** are also valid async iterators. They are also valid async iterables, so they can be used in `for await...of` loops.

To demonstrate how async generators can simplify the implementation of async iterators, let's convert the `CheckUrls` class we saw in the previous example to use an async generator:

```
export class CheckUrls {
  constructor(urls) {
    this.urls = urls
  }

  async *[Symbol.asyncIterator]() {
    for (const url of this.urls) {
      try {
        const checkResult = await fetch(url, {
          method: 'HEAD',
          redirect: 'follow',
          signal: AbortSignal.timeout(5000),
        })
        checkResult.ok
          ? yield `${url} is up, status: ${checkResult.status}`
          : yield `${url} is down, error: ${checkResult.status} ` +
            `${checkResult.statusText}`
      } catch (err) {
```

```
        yield `${url} is down, error: ${err.message}`
    }
  }
}
```

Interestingly, using an async generator in place of a bare async iterator allowed us to save quite a few lines of code, and the resulting logic is also more readable and explicit.

Async iterators and Node.js streams

If we stop for a second and think about the relationship between async iterators and Node.js readable streams, we would be surprised by how similar they are in both purpose and behavior. In fact, we can say that async iterators are indeed a stream construct, as they can be used to process the data of an asynchronous resource piece by piece, exactly as it happens for readable streams.

It's not a coincidence that `stream.Readable` implements the `@@asyncIterator` method, making it an async iterable. This provides us with an additional, and probably even more intuitive, mechanism to read data from a readable stream, thanks to the `for await...of` construct.

To quickly demonstrate this, consider the following example, where we take the `stdin` stream of the current process and pipe it into the `split()` transform stream, which will emit a new chunk when it finds a newline character. Then, we iterate over each line using the `for await...of` loop:

```
import split from 'split2' // v4.2.0

const stream = process.stdin.pipe(split())
for await (const line of stream) {
  console.log(`You wrote: ${line}`)
}
```

This sample code will print back whatever we have written to the standard input only after we have pressed the *Return* key. To quit the program, you can just press *Ctrl + C*.

As we can see, this alternative way of consuming a readable stream is indeed very intuitive and compact. The previous example also shows us how similar the two paradigms—iterators and streams—are. They are so similar that they can interoperate almost seamlessly. To prove this point even further, just consider that the function `stream.Readable.from(iterable, [options])` takes an iterable as an argument, which can be both synchronous or asynchronous. The function will return a readable stream that wraps the provided iterable, “adapting” its interface to that of a readable stream (this is also a good example of the Adapter pattern, which we have already met in *Chapter 8, Structural Design Patterns*).

So, if streams and async iterators are so closely related, which one should you actually use? As always, it depends on your use case, but here are some key differences that can help guide your choice:

- Streams (in flowing mode) use a *push* model: data is produced and pushed into internal buffers as it becomes available. Async iterators use a *pull* model: the consumer explicitly requests each chunk of data, giving it full control over the pace.
- Streams are ideal for binary data and high-throughput scenarios, thanks to built-in buffering and backpressure mechanisms. They're especially useful when working with files, network sockets, or any continuous flow of data.
- Async iterators shine in composability and clarity, especially in asynchronous workflows where data is produced lazily or on demand. They integrate naturally with `for await...of`, making them perfect for APIs, paginated data, or custom logic.

In short: if you need performance, buffering, or integration with Node.js internals, go with streams. If you want simplicity, readability, and pull-based control, async iterators might be the better fit.

- Keep in mind that you can also mix and match streams and async iterators. For example, the `pipeline()` helper (discussed in *Chapter 6, Coding with Streams*) allows you to create a processing pipeline by concatenating an array of streams. Well, this helper supports more than just streams! In the input you provide to the helper, you can mix and match streams, (sync) iterables, and async iterables.



We can iterate over an `EventEmitter` as well. Using the `events.on(emitter, eventName)` utility function, we can, in fact, get an async iterable whose iterator will return all the events matching the specified `eventName`.

Async iterator utilities

As we've seen, synchronous iterators in JavaScript benefit from the dedicated `Iterator` prototype that can be extended, along with a rich set of built-in helper functions such as `map()`, `filter()`, and `reduce()`. Unfortunately, as of this writing, asynchronous iterators don't yet have the same level of built-in convenience.

However, the situation is evolving. An open TC39 proposal (nodejsdp.link/async-iterator-helpers) is actively exploring how to best implement similar helper methods for async iterators. A primary focus of the conversation revolves around managing concurrency between operations, a complex but crucial consideration for asynchronous data streams.

This means that, for now, you lack the ease of use offered by synchronous iterator helpers. However, this could change rapidly as the TC39 proposal progresses, so keep a close watch on its development!

In the meantime, a simple but powerful workaround exists: Node.js streams. We mentioned in the previous section that you can easily transform an async iterator into a readable stream using `Readable.from()`, and crucially, readable streams already support helper methods such as `map()`, `filter()`, and `reduce()`.

Therefore, by combining these two techniques, you can create elegant and efficient asynchronous processing pipelines based on async iterators.

To see this idea in action, let's update our previous URL status-checking application. The `CheckUrls` async iterator remains unchanged, but this time, we want to do a bit of analysis on the data coming from the iterator. Let's say we want to count how many links are up and how many are down. Here's a possible solution:

```
import { Readable } from 'node:stream'
import { CheckUrls } from './checkUrls.js'

const checkUrls = new CheckUrls([
  'https://nodejsdesignpatterns.com',
  'https://loige.co',
  'https://mario.fyi',
  'https://httpstat.us/200',
  'https://httpstat.us/200?sleep=6000',
])

const stats = await Readable.from(checkUrls)
  .map(status => {
    console.log(status)
    return status
  })
  .reduce(
    (acc, status) => {
      if (status.includes(' is up, ')) {
        acc.up++
      } else {
        acc.down++
      }
      return acc
    },
    { up: 0, down: 0 }
  )

console.log(stats)
```


As you can see, we create a readable stream with `Readable.from()`, and then we use `map()` to print all the status messages that come through the stream. Then, we use `reduce()` and do some string matching to check whether the current status is up or down. Finally, we increment the respective field in the accumulator object. Note that the `reduce()` helper consumes the readable stream and returns a promise that resolves to the result of the `reduce()` operation once the stream finishes.

If you execute this code, you should get an output similar to this:

```
https://nodejsdesignpatterns.com is up, status: 200
https://loige.co is up, status: 200
https://mario.fyi is up, status: 200
https://httpstat.us/200 is up, status: 200
https://httpstat.us/200?sleep=6000 is down, error: The operation was aborted
due to timeout
{ up: 4, down: 1 }
```

In the wild

Iterators (in particular, async iterators) are quickly gaining popularity in the Node.js ecosystem. In fact, in many circumstances, they are becoming a preferred alternative to streams and are replacing custom-built iteration mechanisms.

For example, the `@databases/pg`, `@databases/mysql`, and `@databases/sqlite` packages are popular libraries for accessing Postgres, MySQL, and SQLite databases, respectively (more at nodejsdp.link/atdatabases).

They all expose a function called `queryStream()`, which returns an async iterable that can be used to easily iterate over the results of a query, as in this example:

```
for await (const record of db.queryStream(sql`SELECT * FROM my_table`)) {
  // do something with record
}
```

Internally, the iterator will automatically handle the cursor for a query result, so all we have to do is simply loop with the `for await...of` construct.

Another example of a library heavily relying on iterators for its API is the `zeromq` package (nodejsdp.link/npm-zeromq). We'll see a detailed example of it in the next section, about the Middleware pattern, as we move on to other behavioral patterns.

Middleware

One of the most distinctive patterns in Node.js is the **Middleware** pattern. Unfortunately, it's also one of the most confusing, especially for developers coming from the enterprise programming world. The reason for the disorientation is probably connected to the traditional meaning of the term *middleware*, which, in enterprise architecture jargon, represents the various software suites that help to abstract lower-level mechanisms such as operating system APIs, network communications, memory management, and so on, allowing the developer to focus only on the business case of the application. In this context, the term *middleware* recalls topics such as CORBA, enterprise service bus, Spring, JBoss, and WebSphere, but in its more generic meaning, it can also define any kind of software layer that acts as glue between lower-level services and the application (literally, the *software in the middle*).

Middleware in Express

Express (nodejsdp.link/express) popularized the term *middleware* in the Node.js world, binding it to a very specific design pattern. In Express, in fact, *middleware* represents a set of services, typically functions, that are organized in a pipeline and are responsible for processing incoming HTTP requests and related responses.

Express is famous for being a very non-opinionated and minimalist web framework, and the *Middleware* pattern is the main reason for that. Express *middleware* is, in fact, an effective strategy for allowing developers to easily create and distribute new features that can be easily added to an application, without the need to grow the minimalistic core of the framework.

An Express *middleware* has the following signature:

```
function (req, res, next) { ... }
```

Here, *req* is the incoming HTTP request, *res* is the response, and *next* is the callback to be invoked when the current *middleware* has completed its tasks, and that, in turn, triggers the next *middleware* in the pipeline.

Examples of the tasks carried out by Express *middleware* include the following:

- Parsing the body of the request
- Compressing/decompressing requests and responses
- Producing access logs
- Managing sessions
- Managing encrypted cookies
- Providing **cross-site request forgery (CSRF)** protection

If we think about it, these are all tasks that are not strictly related to the main business logic of an application, nor are they essential parts of the minimal core of a web server. They are accessories, components providing support to the rest of the application and allowing the actual request handlers to focus only on their main business logic. Essentially, those tasks are “*software in the middle*.”

Middleware as a pattern

The technique used to implement middleware in Express is not new; in fact, it can be considered the Node.js incarnation of the **Intercepting Filter** pattern and the **Chain of Responsibility** pattern. In more generic terms, it also represents a processing **pipeline**, which reminds us of streams. Today, in Node.js, the word middleware is used well beyond the boundaries of the Express framework, and indicates a particular pattern whereby a set of processing units, filters, and handlers, under the form of functions, are connected to form an asynchronous sequence in order to perform the preprocessing and postprocessing of any kind of data. The main advantage of this pattern is *flexibility*. The Middleware pattern allows us to obtain a plugin infrastructure with incredibly little effort, providing an unobtrusive way to extend a system with new filters and handlers.



If you want to know more about the Intercepting Filter pattern, the following article is a good starting point: nodejsdp.link/intercepting-filter. Similarly, a nice overview of the Chain of Responsibility pattern is available at this URL: nodejsdp.link/chain-of-responsibility.

The following diagram shows the components of the Middleware pattern:

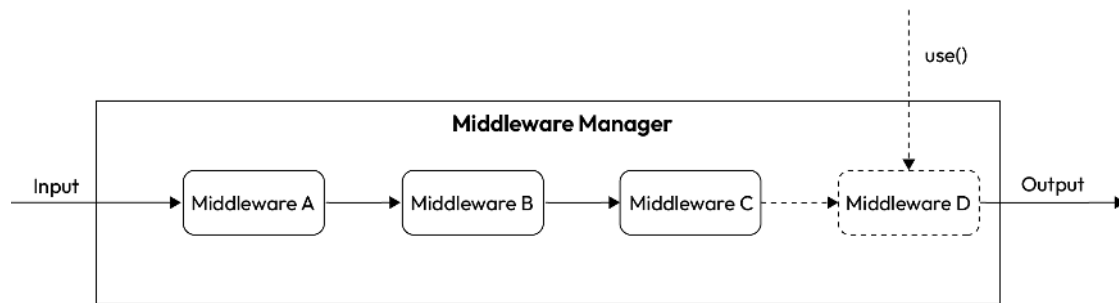


Figure 9.5: The structure of the Middleware pattern

The essential component of the pattern is the **middleware manager**, which is responsible for organizing and executing the middleware functions. The most important implementation details of the pattern are as follows:

- New middleware can be registered by invoking the `use()` function (the name of this function is a common convention in many implementations of the Middleware pattern, but we can choose any name). Usually, new middleware can only be appended at the end of the pipeline, but this is not a strict rule.
- When new data is received for processing, the registered middleware is invoked in an asynchronous sequential execution flow. Each unit in the pipeline receives the result of the execution of the previous unit as input.

- Each piece of middleware can decide to stop further processing of the data. This can be done by invoking a special function, by not invoking the callback (in case the middleware uses callbacks), or by propagating an error. An error situation usually triggers the execution of another sequence of middleware that is specifically dedicated to handling errors.

There is no strict rule on how the data is processed and propagated in the pipeline. The strategies for propagating the data modifications in the pipeline include:

- Augmenting the data received as input with additional properties or functions
- Maintaining the immutability of the data and always returning fresh copies as a result of the processing

The right approach depends on the way the middleware manager is implemented and on the type of processing carried out by the middleware itself.

Creating a middleware framework for ZeroMQ

Let's now demonstrate the pattern by building a middleware framework around the **ZeroMQ** (nodejsdp.link/zeromq) messaging library. ZeroMQ (also known as ZMQ, or ØMQ) provides a simple interface for exchanging atomic messages across the network using a variety of protocols. It shines for its performance, and its basic set of abstractions is specifically built to facilitate the implementation of custom messaging architectures. For this reason, ZeroMQ is often chosen to build complex distributed systems.



In *Chapter 13, Messaging and Integration Patterns*, we will have the chance to analyze the features of ZeroMQ in more detail.

The interface of ZeroMQ is pretty low-level as it only allows us to use strings and binary buffers for messages. So, any encoding or custom formatting of data has to be implemented by the users of the library.

In the next example, we are going to build a middleware infrastructure to abstract the preprocessing and postprocessing of the data passing through a ZeroMQ socket, so that we can transparently work with JSON objects, but also seamlessly compress messages traveling over the wire.

The middleware manager

The first step toward building a middleware infrastructure around ZeroMQ is to create a component that is responsible for executing the middleware pipeline when a new message is received or sent. For this purpose, let's create a new module called `zmqMiddlewareManager.js`, and let's define it:

```
export class ZmqMiddlewareManager {  
  #socket  
  #inboundMiddleware = []  
  #outboundMiddleware = []  
}
```

```
constructor(socket) { // 1
  this.#socket = socket
  this.#handleIncomingMessages()
}

async #handleIncomingMessages() { // 2
  for await (const [message] of this.#socket) {
    await this.#executeMiddleware(this.#inboundMiddleware,
      message).catch(
        err => {
          console.error('Error while processing the message', err)
        }
      )
  }
}

async send(message) { // 3
  const finalMessage = await this.#executeMiddleware(
    this.#outboundMiddleware,
    message
  )
  return this.#socket.send(finalMessage)
}

use(middleware) { // 4
  if (middleware.inbound) {
    this.#inboundMiddleware.push(middleware.inbound)
  }
  if (middleware.outbound) {
    this.#outboundMiddleware.unshift(middleware.outbound)
  }
}

async #executeMiddleware(middlewares, initialMessage) { // 5
  let message = initialMessage
  for await (const middlewareFunc of middlewares) {
    message = await middlewareFunc.call(this, message)
  }
  return message
}
```

Let's discuss in detail how we implemented `ZmqMiddlewareManager`:

1. In the first part of the class, we define the constructor that accepts a ZeroMQ socket as an argument, managed as a private property of the class. We also have two private properties containing two empty lists that will contain our middleware functions, one for inbound messages and another for outbound messages. Next, we immediately start processing the messages coming from the socket. We do that in the private `handleIncomingMessages()` method.
2. In the `handleIncomingMessages()` method, we use the ZeroMQ socket as an async iterable, and with a `for await...of` loop, we process any incoming message and we pass it down the `inboundMiddleware` list using the `executeMiddleware()` function.
3. In a similar fashion, the `send()` method will pass the message received as an argument down the `outboundMiddleware` pipeline using the `executeMiddleware()` function. The result of the processing is stored in the `finalMessage` variable and then sent through the socket.
4. The `use()` method is used for appending new middleware functions to our internal pipelines. In our implementation, each middleware comes in pairs; it's an object that contains two properties, `inbound` and `outbound`. Each property can be used to define the middleware function to be added to the respective list. It's important to observe here that the inbound middleware is pushed to the end of the `inboundMiddleware` list, while the outbound middleware is inserted (using `unshift()`) at the beginning of the `outboundMiddleware` list. This is because complementary inbound/outbound middleware functions usually need to be executed in the inverted order. For example, if we want to decompress and then deserialize an inbound message using JSON, it means that for the outbound, we should instead first serialize and then compress. This convention for organizing the middleware in pairs is not strictly part of the general pattern, but only an implementation detail of our specific example.
5. The last method, `executeMiddleware()`, represents the core of our component as it's the part responsible for executing the middleware functions. Each function in the middleware array received as input is executed one after the other, and the result of the execution of a middleware function is passed to the next. Note that we are using the `await` instruction on each result returned by each middleware function; this allows the middleware function to return a value synchronously as well as asynchronously using a promise. Finally, the result of the last middleware function is returned to the caller.



For brevity, we are not supporting an error middleware pipeline. Normally, when a middleware function propagates an error, another set of middleware functions specifically dedicated to handling errors is executed. This can be easily implemented using the same technique that we are demonstrating here. For instance, we could accept an extra (optional) `errorMiddleware` function in addition to `inboundMiddleware` and `outboundMiddleware`.

Implementing the middleware to process messages

Now that we have implemented our middleware manager, we can create our first pair of middleware functions to demonstrate how to process inbound and outbound messages. As we said, one of the goals of our middleware infrastructure is to have a filter that serializes and deserializes JSON messages. So, let's create a new middleware to take care of this. In a new module called `jsonMiddleware.js`, let's include the following code:

```
export function jsonMiddleware() {
  return {
    inbound(message) {
      return JSON.parse(message.toString())
    },
    outbound(message) {
      return Buffer.from(JSON.stringify(message))
    },
  }
}
```

The inbound part of our middleware deserializes the message received as input, while the outbound part serializes the data into a string, which is then converted into a buffer.

In a similar way, we can implement a pair of middleware functions in a file called `zlibMiddleware.js`, to inflate/deflate the message using the `zlib` core module ([nodejsdp.link/zlib](https://nodejs.org/api/zlib.html)):

```
import { inflateRaw, deflateRaw } from 'node:zlib'
import { promisify } from 'node:util'

const inflateRawAsync = promisify(inflateRaw)
const deflateRawAsync = promisify(deflateRaw)

export function zlibMiddleware() {
  return {
    inbound(message) {
      return inflateRawAsync(Buffer.from(message))
    },
    outbound(message) {
      return deflateRawAsync(message)
    },
  }
}
```

Compared to the JSON middleware, our `zlib` middleware functions are asynchronous and return a promise as a result. As we already know, this is perfectly supported by our middleware manager.

You can note how the middleware used by our framework is quite different from the one used in Express. This is totally normal and a perfect demonstration of how we can adapt this pattern to fit our specific needs.

Using the ZeroMQ middleware framework

We are now ready to use the middleware infrastructure that we just created. To do that, we are going to build a very simple application, with a client sending a *ping* to a server at regular intervals and the server echoing back the message received.

From an implementation perspective, we are going to rely on a request/reply messaging pattern using the req/rep socket pair provided by ZeroMQ (nodejsdp.link/zmq-req-rep). We will then wrap the sockets with our `ZmqMiddlewareManager` to get all the advantages from the middleware infrastructure that we built, including the middleware for serializing/deserializing JSON messages.



We'll analyze the request/reply pattern and other messaging patterns in *Chapter 13, Messaging and Integration Patterns*.

The server

Let's start by creating the server side of our application in a file called `server.js`:

```
import zmq from 'zermq' // v6.3.0 // 1
import { ZmqMiddlewareManager } from './zmqMiddlewareManager.js'
import { jsonMiddleware } from './jsonMiddleware.js'
import { zlibMiddleware } from './zlibMiddleware.js'

const socket = new zmq.Reply() // 2
await socket.bind('tcp://127.0.0.1:5000')

const zmqm = new ZmqMiddlewareManager(socket) // 3
zmqm.use(zlibMiddleware())
zmqm.use(jsonMiddleware())
zmqm.use({ // 4
  async inbound(message) {
    console.log('Received', message)
    if (message.action === 'ping') {
      await this.send({ action: 'pong', echo: message.echo })
    }
    return message
  },
})

console.log('Server started')
```


The server side of our application works as follows:

1. We first load the necessary dependencies. The zeromq package is essentially a JavaScript interface over the native ZeroMQ library (see nodejsdp.link/npm-zeromq).
2. Next, we create a new ZeroMQ Reply socket and bind it to port 5000 on localhost.
3. Then comes the part where we wrap ZeroMQ with our middleware manager and then add the zlib and JSON middlewares.
4. Finally, we are ready to handle a request coming from the client. We will do this by simply adding another middleware—this time, using it as a request handler.

Since our request handler comes after the zlib and JSON middlewares, we will receive a decompressed and deserialized version of the received message. On the other hand, any data passed to `send()` will be processed by the outbound middleware, which, in our case, will serialize and then compress the data.

The client

On the client side of our little application, in a file called `client.js`, we will have the following code:

```
import zeromq from 'zeromq' // v6.3.0
import { ZmqMiddlewareManager } from './zmqMiddlewareManager.js'
import { jsonMiddleware } from './jsonMiddleware.js'
import { zlibMiddleware } from './zlibMiddleware.js'

const socket = new zeromq.Request() // 1
await socket.connect('tcp://127.0.0.1:5000')

const zmqm = new ZmqMiddlewareManager(socket)
zmqm.use(zlibMiddleware())
zmqm.use(jsonMiddleware())
zmqm.use({
  inbound(message) {
    console.log('Echoed back', message)
    return message
  },
})

setInterval(() => { // 2
  zmqm
    .send({ action: 'ping', echo: Date.now() })
    .catch(err => console.error(err))
}, 1000)

console.log('Client connected')
```

Most of the code of the client application is very similar to that of the server. The notable differences are:

1. We create a `Request` socket rather than a `Reply` socket, and we connect it to a remote (or local) host rather than binding it to a local port. The rest of the middleware setup is the same as in the server, except for the fact that our request handler now just prints any message it receives. Those messages should be the *pong* reply to our *ping* requests.
2. The core logic of the client application is a timer that sends a *ping* message every second.

Now, we're ready to try our client/server pair and see the application in action. First, start the server:

```
node server.js
```

We can then start the client in another terminal with the following command:

```
node client.js
```

At this point, we should see the client sending messages and the server echoing them back.

Our middleware framework did its job. It allowed us to decompress/compress and deserialize/serialize our messages transparently, leaving the handlers free to focus on their business logic.

In the wild

We opened this section by saying that the library that popularized the Middleware pattern in Node.js is Express (nodejsdp.link/express). So, we can easily say that Express is also the most notable example of the Middleware pattern out there.

Three other interesting examples are as follows:

- Koa (nodejsdp.link/koa) is another alternative to Express and certainly draws inspiration from it.
- Middy (nodejsdp.link/middy) is a classic example of the Middleware pattern applied to something other than a web framework. Middy is, in fact, a middleware engine for AWS Lambda functions.
- Hono (nodejsdp.link/hono) is an emerging web framework built on web standards that promises support for any JavaScript runtime (not just Node.js). One interesting detail is that Hono's middleware engine offers a built-in `combine` middleware (nodejsdp.link/hono-combine), which allows combining multiple middleware functions into a single middleware.

It provides three functions: `some` (runs only one of the given middleware), `every` (runs all given middleware), and `except` (runs all given middleware only if a condition is not met). This can allow you to build powerful conditional behavior with very little effort. For example, you could implement “skip rate limiting if the user has a given IP” without having to create a custom middleware for that.

Next, we are going to explore the Command pattern, which, as we will see shortly, is a very flexible and multiform pattern.

Command

Another design pattern with huge importance in Node.js is **Command**. In its most generic definition, we can consider a command any object that encapsulates all the information necessary to perform an action later. So, instead of invoking a method or a function directly, we create an object representing the intention to perform such an invocation. It will then be the responsibility of another component to materialize the intent, transforming it into an actual action. Traditionally, this pattern is built around four major components, as shown in *Figure 9.6*:

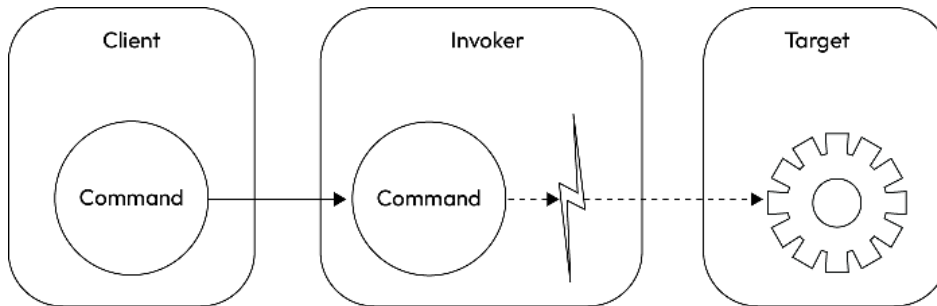


Figure 9.6: The components of the Command pattern

The typical configuration of the Command pattern can be described as follows:

- **Command** is the object encapsulating the information necessary to invoke a method or function.
- **Client** is the component that creates the command and provides it to the invoker.
- **Invoker** is the component responsible for executing the command on the target.
- **Target** (or **receiver**) is the subject of the invocation. It can be a lone function or a method of an object.

As we will see, these four components can vary a lot depending on the way we want to implement the pattern. This should not sound new at this point. For example, it's not uncommon to see the client and the invoker parts being implemented together in the same class.

Using the Command pattern instead of directly executing an operation has several applications:

- A command can be scheduled for execution later.
- A command can be easily serialized and sent over the network. This simple property allows us to distribute jobs across remote machines, transmit commands from the browser to the server, create **remote procedure call (RPC)** systems, and so on.
- Commands make it easy to keep a history of all the operations executed on a system.
- Commands are an important part of some algorithms for data synchronization and conflict resolution.
- A command scheduled for execution can be canceled if it hasn't yet been executed. It can also be reverted (undone), bringing the state of the application to the point before the command was executed.

- Several commands can be grouped together. This can be used to create atomic transactions or to implement a mechanism whereby all the operations in the group are executed at once.
- Different kinds of transformation can be performed on a set of commands, such as duplicate removal, joining and splitting, or applying more complex algorithms such as **operational transformation** (OT) (nodejsdp.link/operational-transformation), which is the base for most of today's real-time collaborative software, such as collaborative text editing.

The preceding list clearly shows us how important this pattern is, especially on a platform such as Node.js, where networking and asynchronous execution are essential players.

Now, we are going to explore in more detail a couple of different implementations of the Command pattern, just to give you an idea of its scope.

The Task pattern

We can start with the most basic and trivial implementation of the Command pattern: the **Task pattern**. The easiest way in JavaScript to create an object representing an invocation is by creating a closure around a function definition or a **bound function**:

```
function createTask(target, ...args) {  
  return () => {  
    target(...args)  
  }  
}
```

This is (mostly) equivalent to doing:

```
const task = target.bind(null, ...args)
```

This should not look new at all. In fact, we have used this pattern already so many times throughout the book, and in particular, in *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*. This technique allowed us to use a separate component to control and schedule the execution of our tasks, which is essentially equivalent to the invoker component of the Command pattern.

A more complex command

Let's now work on a more articulated example leveraging the Command pattern. This time, we want to support *undo* and *serialization*. Let's start with the *target* of our commands, a little object that is responsible for sending status updates to a service such as a social platform. We will use a mockup of such a service for simplicity (the `statusUpdateService.js` file):

```
const statusUpdates = new Map()  
  
export const statusUpdateService = {  
  postUpdate(status) {  
    const id = Math.floor(Math.random() * 1000000)  
    statusUpdates.set(id, status)  
  }  
}
```

```
    console.log(`Status posted: ${status} (${id})`)
    return id
  },

  destroyUpdate(id) {
    statusUpdates.delete(id)
    console.log(`Status removed: ${id}`)
  },
}
```

The `statusUpdateService` we just created represents the target of our Command pattern. Now, let's implement a factory function that creates a command to represent the posting of a new status update. We'll do that in a file called `createPostStatusCmd.js`:

```
export function createPostStatusCmd(service, status) {
  let postId = null

  return {
    run() {
      postId = service.postUpdate(status)
    },
    undo() {
      if (postId) {
        service.destroyUpdate(postId)
        postId = null
      }
    },
    serialize() {
      return { type: 'status', action: 'post', status }
    },
  }
}
```

The preceding function is a factory function that produces commands to model “post status” intentions. Each command implements the following three functionalities:

- A `run()` method that, when invoked, will trigger the action. In other words, it implements the *Task* pattern that we have seen before. The command, when executed, will post a new status update using the methods of the target service.
- An `undo()` method that reverts the effects of the *post* operation. In our case, we are simply invoking the `destroyUpdate()` method on the target service.
- A `serialize()` method that builds a JSON object that contains all the necessary information to reconstruct the same command object.

After this, we can build an Invoker. We can start by implementing its `run()` method (in the `invoker.js` file):

```
export class Invoker {
  #history = []

  run(cmd) {
    this.#history.push(cmd)
    cmd.run()
    console.log('Command executed', cmd.serialize())
  }

  // ...rest of the class
}
```

The `run()` method is the basic functionality of our Invoker. It is responsible for saving the command into the history instance variable and then triggering the execution of the command itself.

Next, we can add to the Invoker a new method that delays the execution of a command:

```
delay(cmd, delay) {
  setTimeout(() => {
    console.log('Executing delayed command', cmd.serialize())
    this.run(cmd)
  }, delay)
}
```

Then, we can implement an `undo()` method that reverts the last command:

```
undo() {
  const cmd = this.#history.pop()
  if (cmd) {
    cmd.undo()
    console.log('Command undone', cmd.serialize())
  }
}
```

Finally, we also want to be able to run a command on a remote server, by serializing and then transferring it over the network using a web service:

```
async runRemotely(cmd) {
  await fetch('http://localhost:3000/cmd', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(cmd.serialize()),
  })
}
```

```
    })  
  
    console.log('Command executed remotely', cmd.serialize())  
  }  
}
```

Now that we have the command, the invoker, and the target, the only component missing is the client, which we will implement in a file called `client.js`. Let's start by importing all the necessary dependencies and by instantiating `Invoker`:

```
import { createPostStatusCmd } from './createPostStatusCmd.js'  
import { statusUpdateService } from './statusUpdateService.js'  
import { Invoker } from './invoker.js'  
  
const invoker = new Invoker()
```

Then, we can create a command using the following line of code:

```
const command = createPostStatusCmd(statusUpdateService, 'HI!')
```

We now have a command representing the posting of a status message. We can then decide to dispatch it immediately:

```
invoker.run(command)
```

Oops, we made a mistake! Let's revert our timeline to the state it was before posting the last message:

```
invoker.undo()
```

We can also decide to schedule the message to be sent 3 seconds from now:

```
invoker.delay(command, 1000 * 3)
```

Alternatively, we can distribute the load of the application by migrating the task to another machine:

```
invoker.runRemotely(command)
```

The example we've just gone through demonstrates how encapsulating an operation within a command can unlock a world of possibilities. While this simple example barely scratches the surface, it hints at the numerous real-world applications of this design pattern. For instance, consider a real-time collaboration tool where commands are utilized to track and apply user actions across multiple remote clients. This approach ensures that every action taken by any client is recorded and replicated in real time, fostering seamless collaboration among users. Another compelling application is in a distributed batch processing environment. Here, each job can be queued as a command, allowing multiple workers to retrieve tasks from the queue and execute them independently and in parallel. This setup not only enhances scalability but also ensures efficient resource utilization across a network of workers.

In both scenarios, the Command pattern proves its versatility by enabling modular and scalable solutions. It transforms operations into manageable units that can be tracked, executed, and coordinated across diverse systems.

On the flipside, it is worth noting that a fully-fledged Command pattern should be used only when strictly necessary. We saw how much additional code we had to write to simply invoke a method of the `statusUpdateService`. If all that we need is only an invocation, then a complex command would be overkill. If, however, we need to schedule the execution of a task or run an asynchronous operation, then the simpler *Task pattern* offers the best compromise. If, instead, we need more advanced features, such as undo support, transformations, conflict resolution, or one of the other fancy use cases that we described previously, using a more complex representation for the command is almost necessary.

In the wild

A great real-life example that uses the Command pattern is the **AWS SDK for JavaScript v3** ([nodejsdp.link/aws-sdk](https://www.npmjs.com/package/@aws-sdk/client-s3)). In previous versions of the AWS SDK for JavaScript, interacting with AWS services often involved directly calling methods on service objects. While functional, this approach often led to tightly coupled code and made it difficult to manage cross-cutting concerns such as request retries, logging, and authentication. With the release of v3, the AWS SDK for JavaScript has embraced the Command pattern, offering a more structured and flexible way to interact with AWS services. This pattern encapsulates each AWS API call as a separate command object, decoupling the client code from the specific details of how the API call is executed.

Here's an example of how to upload a file to the S3 object storage service using the Command pattern in the AWS SDK for JavaScript v3:

```
import { PutObjectCommand, S3Client } from '@aws-sdk/client-s3' // v3.750.0
import { createReadStream } from 'node:fs'
import { basename } from 'node:path'

const [bucketName, filePath] = process.argv.slice(2)

if (!(bucketName && filePath)) {
  console.error('Usage: node index.js <bucketName> <filePath>')
  process.exit(1)
}

const s3Client = new S3Client()
const fileStream = createReadStream(filePath)
const key = basename(filePath)

// Set the parameters for the PutObject command.
const params = {
  Bucket: bucketName,
  Key: key,
  Body: fileStream,
}
```



```
try {  
  // Create a PutObject command object.  
  const putObjectCommand = new PutObjectCommand(params)  
  // Send the command to S3.  
  const data = await s3Client.send(putObjectCommand)  
  console.log('Successfully uploaded file to S3', data)  
} catch (err) {  
  console.log('Error', err)  
}
```

This example implements a simple CLI utility that allows uploading a given file from the local disk to a remote S3 bucket. Note how the PutObject (upload) operation is abstracted through the command pattern; in fact, we need to create a PutObjectCommand that contains all the details of the operation we want to perform. Then, we can send this command using an instance of the S3Client class.

Summary

We opened this chapter with three closely related patterns: Strategy, State, and Template.

Strategy allows us to extract the common parts of a family of closely related components into a component called the context and allows us to define strategy objects that the context can use to implement specific behaviors. The State pattern is a variation of the Strategy pattern, where the strategies are used to model the behavior of a component when under different states. The Template pattern can be considered the “static” version of the Strategy pattern, where the different specific behaviors are implemented as subclasses of the template class, which models the common parts of the component.

Next, we learned about what has now become a core pattern in Node.js, which is the Iterator pattern. We learned how JavaScript offers native support for the pattern (with the iterator and iterable protocols), and how async iterators can be used as an alternative to complex async iteration patterns and even to Node.js streams.

Then, we examined Middleware, which is a very distinctive pattern born from within the Node.js ecosystem. We learned how it can be used to preprocess and postprocess data and requests.

Finally, we had a taste of the possibilities offered by the Command pattern, which can be used to implement a myriad of functionality, from simple undo/redo and serialization to more complex operational transformation algorithms.

We have now arrived at the end of the last chapter dedicated to “traditional” design patterns. By now, you should have added to your toolbelt a series of patterns that will be enormously useful in your everyday programming endeavors.

Having now reached the end of our exploration into “traditional” design patterns, it’s time to turn our attention to an equally critical aspect of software development: testing. In the next chapter, we’ll shift gears and dive deep into the principles and practices that ensure these patterns (and your applications as a whole) function reliably. We’ll explore the different types of tests, from unit tests to integration tests (mapping the testing pyramid), and delve into the world of test doubles, learning how to create stubs, mocks, and spies to isolate and verify your code. We’ll also introduce Node.js’s built-in test runner, showing you how to write and execute tests effectively. Finally, we’ll demonstrate how to combine test doubles with dependency injection to write highly testable code, and briefly touch on other, more specialized, testing approaches. So, get ready to level up your testing game and build more robust Node.js applications!

Exercises

- **Exercise 9.1 Logging with Strategy:** Implement a logging component having at least the following methods: `debug()`, `info()`, `warn()`, and `error()`. The logging component should also accept a strategy that defines where the log messages are sent. For example, we might have `ConsoleStrategy` to send the messages to the console, or `FileStrategy` to save the log messages to a file.
- **Exercise 9.2 Logging with Template:** Implement the same logging component we defined in the previous exercise, but this time, using the Template pattern. We would then obtain a `ConsoleLogger` class to log to the console or a `FileLogger` class to log to a file. Appreciate the differences between the Template and the Strategy approaches.
- **Exercise 9.3 Warehouse item:** Imagine we are working on a warehouse management program. Our next task is to create a class to model a warehouse item and help track it. Such a `WarehouseItem` class has a constructor, which accepts an `id` and the initial state of the item (which can be one of `arriving`, `stored`, or `delivered`). It has three public methods:
 - `store(locationId)` moves the item into the `stored` state and records the `locationId` where it’s stored.
 - `deliver(address)` changes the state of the item to `delivered`, sets the delivery address, and clears the `locationId`.
 - `describe()` returns a string representation of the current state of the item (for example, “Item 5821 is on its way to the warehouse,” or “Item 3647 is stored in location 1ZH3,” or “Item 3452 was delivered to John Smith, 1st Avenue, New York”).

The `arriving` state can be set only when the object is created, as it cannot be transitioned to from the other states. An item can’t move back to the `arriving` state once it’s `stored` or `delivered`, it cannot be moved back to `stored` once it’s `delivered`, and it cannot be `delivered` if it’s not `stored` first. Use the State pattern to implement the `WarehouseItem` class.

- **Exercise 9.4 Logging with Middleware:** Rewrite the logging component you implemented for exercises 9.1 and 9.2, but this time, use the Middleware pattern to postprocess each log message, allowing different middlewares to customize how to handle the messages and how to output them. We could, for example, add a `serialize()` middleware to convert the log messages to a string representation ready to be sent over the wire or saved somewhere. Then, we could add a `saveToFile()` middleware that saves each message to a file. This exercise should highlight the flexibility and universality of the Middleware pattern.
- **Exercise 9.5 Queues with iterators:** Implement an `AsyncQueue` class similar to one of the `TaskQueue` classes we defined in *Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await*, but with a slightly different behavior and interface. Such an `AsyncQueue` class will have a method called `enqueue()` to append new items to the queue and then expose an `@@asyncIterable` method, which should provide the ability to process the elements of the queue asynchronously, one at a time (so, with a concurrency of 1). The async iterator returned from `AsyncQueue` should terminate only after the `done()` method of `AsyncQueue` is invoked *and* only after all items in the queue are consumed. Consider that the `@@asyncIterable` method could be invoked in more than one place, thus returning an additional async iterator, which would allow you to increase the concurrency with which the queue is consumed.

10

Testing: Patterns and Best Practices

It has been a long journey; congratulations on making it this far in this book! Throughout the first nine chapters of this book, we have explored the intricate world of JavaScript and Node.js. We understood how the event loop works and how to fully leverage JavaScript's asynchronous nature in Node.js through events, callbacks, promises, `async/await`, and streams. We learned various techniques and design patterns to build scalable, maintainable, and reliable software. We covered creational patterns to manage object instantiation, structural patterns to assemble objects and classes into larger structures, and behavioral patterns to define how objects interact. This robust foundation has prepared us for the next critical step: ensuring our code behaves as expected through comprehensive testing.

Even the most carefully designed code is not immune to bugs. As Edsger W. Dijkstra, one of the most influential figures in the history of software engineering and computer science, famously said: *“Program testing can be used to show the presence of bugs, but never to show their absence!”*

This candid observation reminds us that while testing may never guarantee perfection, it is an indispensable tool in our arsenal for minimizing risks and building reliable software. The design patterns we've explored so far enable us to write elegant and maintainable code, but testing is what can truly give us the confidence that our code is resilient and can withstand the uncertainties of real-world use.

In this chapter, we will dive into the world of testing in Node.js. We will explore essential questions such as: *How do we structure our tests effectively? How can we isolate dependencies to ensure our tests are meaningful? What tools and techniques can we use to automate and streamline our testing processes?* By answering these questions, we will equip you with the skills needed to confidently verify that your code behaves as intended across a variety of scenarios.

More specifically, in this chapter, you will learn the following:

- The definitions and the principles of testing, and the various types of tests, including unit, integration, and End-to-End (E2E) tests
- The testing pyramid, which provides guidance on how to balance different kinds of tests within your project
- How to use Node.js’s built-in test runner, a lightweight and efficient tool for writing and executing tests without relying on external dependencies
- Test doubles: Understanding stubs, mocks, and spies to replace real dependencies and simulate behaviors in a controlled environment
- The synergy between test doubles and Dependency Injection (DI), a key strategy to enhance testability and modularity
- How to write integration tests (with the Node.js built-in test runner) and E2E tests

Testing is a vast and continually evolving field, with entire books devoted to mastering its intricacies. Although we will try to provide some foundational knowledge, our focus here is on the practical testing patterns that are most relevant to Node.js development. As you continue your journey, remember that testing not only improves the quality of your code but also boosts your confidence as a developer, empowering you to deliver software that truly stands the test of time. As such, it is a skill that we encourage you to keep exploring and refining even beyond the boundaries of this book to keep growing as a software engineer.

Let’s roll up our sleeves and dive into the world of testing in Node.js!

An introduction to software testing

Testing is one of the most debated topics among developers. Should you follow **Test-Driven Development (TDD)** or **Behavior-Driven Development (BDD)**? What exactly qualifies as a unit test? What level of code coverage is acceptable? Is code coverage really a relevant metric? Are E2E tests necessary? Should testing be done locally or in a remote environment? And most importantly, how can you ensure your testing environments reliably replicate production?

If you’ve ever grappled with these questions, you’re not alone. Testing can be a contentious subject, but it remains a fundamental part of software development.

To understand its importance, consider an alternative reality where systems are deployed without any testing. You write some code, push it to production, and move on. It might seem efficient... until you receive a call at 3 A.M. because something is broken. Now you’re having to get out of bed to diagnose and fix an issue that could have been caught earlier. Debugging under pressure, deploying fixes, and maybe even reverting code... all that eats up way more time than if you’d just written a few tests in the first place. That “shortcut” ends up being a really long way around.

The benefits of testing go beyond catching bugs early. One of the most significant advantages of testing is fast feedback. Many developers work across complex systems, frequently shifting between different parts of the codebase. Each context switch requires time to regain focus. The longer it takes to verify a change, the harder it is to stay productive. If deploying and then verifying a change takes 90 minutes, you're likely to get distracted, making it harder to jump back into your work. Well-structured automated test suites help solve this by providing immediate feedback, ensuring you can iterate quickly and efficiently in a laser-focused fashion.

Another, perhaps surprising, positive effect that tests can have in a codebase is maximizing collaboration. Having a well-tested codebase means that the development workflow is more uniform and predictable, which, in turn, makes it easy to reproduce issues and document test cases, easing the collaboration among different developers or teams and shortening the time it takes to deliver changes. This is especially important in open-source projects where people from the most diverse backgrounds and with the most disparate expectations come to contribute: tests are one of the key aspects that allow both contributors and maintainers to have the confidence that proposed changes are not going to affect the reliability of the software. In addition to this, well-written tests can also serve as documentation. For instance, someone joining a project could review the tests to understand its implemented capabilities, relevant errors, and edge cases. Similarly, adopting a well-tested open-source library allows one to grasp how it can be used and its supported use cases simply by reading the tests.

We have already mentioned some testing-specific terms. If you are approaching this topic for the first time, some of them might be confusing. So, before diving any further, let's provide some definitions.

Definitions

The definitions in this section are meant to help you to navigate the rest of this chapter and build a more structured mental model about the large world of software testing. As we have already said, we are not going to go particularly deep, but you should be able to use these definitions as a jump-off point if needed.

System under test

The **System Under Test (SUT)** is the specific component, module, function, or entire application that is being evaluated or tested during a particular test case. It's the focal point of the test: the part of the system whose behavior is being verified. The SUT can range in scope from a single line of code to a complex, multi-tiered application.

Arrange, Act, Assert

The **Arrange-Act-Assert (AAA)** pattern is a widely used way of structuring unit tests and other types of automated tests. It provides a clear and consistent approach to writing tests, making them easier to understand, maintain, and debug.

Essentially, you split your test code into three different parts:

- **Arrange:** You set up some pre-conditions for the SUT
- **Act:** You execute your system with those preconditions
- **Assert:** You verify the results to confirm that your system behaved as expected

Every test you ever write, regardless of the type, will likely follow these three high-level steps.

We will see this approach in action once we start working on some examples of unit tests.

Code coverage

Code coverage is a metric that measures the percentage of your codebase that is executed when you run your test suite. It provides insight into how thoroughly your tests exercise your code. Depending on the testing framework used, code coverage can be measured at different levels, including the following:

- **Line coverage:** Percentage of lines of code executed
- **Branch coverage:** Percentage of decision points (e.g., if statements, switch cases) executed
- **Statement coverage:** Percentage of executable statements executed (a single line can contain multiple statements)
- **Class coverage:** Percentage of classes covered by the test suite
- **Function coverage:** Percentage of functions invoked by the test suite

While code coverage can be a useful tool for identifying areas of your code that lack testing, it's crucial to understand its limitations. A high code coverage percentage does not automatically guarantee high-quality code or that your tests are meaningfully verifying the behavior of your system.

Code coverage only tells you what code is executed by your tests, not how it's tested. You can achieve 100% code coverage with tests that have zero assertions, meaning your tests are simply running the code without verifying that it behaves correctly. These are examples of trivial tests, and they are bad: they add no value to a codebase.

Instead of solely focusing on achieving a high level of code coverage, prioritize writing focused tests that thoroughly verify the behavior of individual components under different significant circumstances.

Code coverage should be viewed as a byproduct of these practices rather than the primary objective. It's a useful indicator, but it shouldn't be the sole metric driving your testing strategy or for measuring the quality of your tests.

Test doubles: Stubs, spies, and mocks

Test doubles are stand-in components used to isolate the SUT from its dependencies. Like a stunt double in a movie, they mimic real implementations but are tailored for controlled testing environments. Dependencies (e.g., databases, APIs, or services) are replaced with test doubles to ensure the show goes on smoothly, even when these dependencies aren't ready, reliable, or practical to use. Let's break down the three most common types of test doubles, their purposes, and how to use them effectively.

Stubs

Stubs are the foundation of test isolation. Imagine replacing a live credit card processor with a scripted actor who always delivers the same line: "Approved!" or "Declined!" That's a stub.

Stubs provide predetermined, static responses to method calls. They ignore context, no matter what input they receive; they stick to their script. For example, a weather API stub might always return "sunny" to test an app's UI rendering, bypassing real-world unpredictability.

Spies

Spies take stubs a step further. They're stubs with a memory: they retain stub-like behavior but record interactions for later inspection.

A spy tracks the following:

- How many times a method was called
- What arguments were passed each time
- The order of calls in a sequence

For instance, a spy on an email service could confirm that a “password reset” message was sent once after a user trigger. Unlike a stub, it doesn't just simulate behavior; it observes it.

Mocks

Mocks are spies with a rulebook. They combine stub-like responses with predefined expectations and actively enforce them.

Mocks act as strict validators: they define exactly how a method should be called (arguments and call count), and they fail the test if reality deviates from expectations.

For example, a mock authentication service might demand the following:

1. The `checkPermissions()` function must be called exactly once with “admin” rights
2. The `logAccess()` method must be invoked once with the current admin user ID

If either of these rules fails, the entire test fails.

Mocks ensure critical interactions adhere to contracts, such as verifying compliance with security protocols or API specifications.

Be aware that mocks tightly couple tests to implementation details. If you refactor the code (e.g., rename a method or adjust parameters), mock-based tests break, even if the system's overall behavior remains correct. Try not to over-rely on mocks.



When it comes to test doubles, the term *mock* is often used informally to represent the entire category, including stubs and spies. This is because mocks offer the most comprehensive set of features among these options. As a result, many testing frameworks provide a single abstraction called *mock* that can be used to implement various types of test doubles, including stubs and spies, which often overlap in their functionality.

Test-driven development

TDD is more than just a way to test your code; it's a development approach that shapes how you design and structure your application. The core idea is simple: before writing any implementation, you first define what the code should do by writing a test. This keeps the focus on the actual requirements and prevents unnecessary code from creeping in.

At the heart of TDD is a straightforward but effective cycle, often called **Red-Green-Refactor**:

- **Red** (write a failing test): Start by writing a test that describes the behavior you want to implement. Since the code doesn't exist yet, the test fails, hence, "red."
- **Green** (make the test pass): Write just enough code to get the test to pass. This step is about functionality, not perfection. Your goal is to move from red to green as quickly as possible.
- **Refactor** (improve the code): Now that the test is passing, you can clean up the code. This means improving structure, eliminating duplication, and making it more maintainable, all while ensuring the test keeps passing.

TDD is often praised for helping developers write more structured, testable, and maintainable code. But it's also a bit of a polarizing topic. Some developers swear by it, using it religiously on every project. Others find it too rigid or impractical for their workflow. Many take a middle ground, applying TDD selectively when it makes sense.

While TDD is an important methodology, this chapter isn't about promoting or debating it. We mention it because it's a common topic in testing discussions, but the techniques we'll explore apply whether you choose to use TDD or not.

Behavior-driven development

BDD extends the idea of executable specifications, focusing on describing and verifying a system's behavior in a way that's clear to both technical and non-technical stakeholders. At its core, BDD is about improving communication using tests as a bridge between developers, product teams, and even business stakeholders to ensure everyone has a shared understanding of the requirements.

To achieve this, BDD expresses tests in a structured, human-readable format, typically using a language called **Gherkin**. Gherkin allows you to define features and scenarios in plain language, making them accessible to people who may not have a technical background. This approach helps align development efforts with business goals, reducing misunderstandings and ensuring that software behaves as expected.

Just to give you a feeling of what the Gherkin language looks like, here's an example of a specification for a user authentication flow:

```
Feature: User Authentication
```

```
Scenario: Successful Login
```

```
    Given I am on the login page
```

```
    When I enter valid username "user123" and password "password123"
```

```
    And I click the "Login" button
```

```
    Then I should be redirected to the dashboard page
```

```
    And I should see a welcome message "Welcome, user123!"
```

```
Scenario: Failed Login with Invalid Password
```

```
    Given I am on the login page
```

```
When I enter username "user123" and invalid password "wrongpassword"  
And I click the "Login" button  
Then I should remain on the login page  
And I should see an error message "Invalid username or password"
```

BDD testing tools allow you to parse this specification file as a source and use it to implement all the various steps of your test scenarios: **Given** (setup), **When** (actions), and **Then** (assertions).



In JavaScript and Node.js, one of the most widely used solutions to implement BDD using the Gherkin specification language is Cucumber (nodejsdp.link/cucumber).

BDD can be particularly useful when working on projects where clear communication of business requirements is critical, especially in teams that include product owners, QA engineers, or other stakeholders who may not be comfortable reading traditional test code. However, like TDD, BDD is not a one-size-fits-all solution. It requires discipline and works best when there's a real need for collaboration between technical and non-technical team members.

While we won't be focusing on BDD frameworks in this book, we encourage you to explore them on your own. The principles behind BDD can still be applied to the examples in this book, and adopting a behavior-driven mindset can help you write tests that better capture real-world use cases.

Continuous integration

Continuous Integration (CI) is all about making sure your code plays nicely with everyone else's, early and often. Instead of waiting days or weeks to merge changes, developers push their code to a shared repository frequently, triggering automated builds and tests to catch issues before they become real problems.

Back in the day, teams would work in isolation for long stretches, only to face painful, messy integrations when they finally merged their code. Bugs would pile up, releases would get delayed, and fixing things felt like putting out fires. CI flips that script by encouraging small, incremental updates that get tested automatically, ensuring that new code blends smoothly into the project without breaking anything.

Most CI setups revolve around a version control system such as Git, with a CI service running automated tests whenever new changes land in the repository. Developers can even run local tests before committing, acting as an extra safeguard. The end goal? Fewer surprises, better software quality, and a much smoother path to delivering updates to users.

Continuous delivery and continuous deployment

Continuous Delivery (CD) and **Continuous Deployment** (also CD, just to keep things confusing) both aim to automate and streamline the release process, but they differ in how far they take automation.

With continuous delivery, every change that passes through the CI pipeline is automatically built, tested, and deployed to a staging or testing environment. From there, it's up to a human (typically a developer, QA engineer, or release manager) to decide when to push the update to production. This approach ensures that teams always have a deployment-ready build but maintain control over when new changes go live. It's great for teams that want to release frequently but still need manual checks or business approvals before shipping to customers.

Continuous deployment, on the other hand, removes that final manual step. If a change passes all automated tests, it goes straight to production; no human intervention is required. This means new features, bug fixes, and improvements can reach users within minutes of being merged. The key to making this work is having a robust suite of automated tests, including regression tests, to catch issues before they ever make it to production. Because every change is deployed immediately, teams eliminate the bottlenecks of manual approvals, release planning meetings, and long deployment cycles.

The choice between continuous delivery and continuous deployment often comes down to risk tolerance and business needs. If you're working in a highly regulated industry or have customers who require predictable release schedules, continuous delivery might be the better fit. If speed and agility are the priority (think SaaS applications, startups, or internal tools), continuous deployment can be a game-changer, allowing teams to iterate rapidly and deliver value to users as soon as possible.

Regardless of which approach you take, the underlying principles remain the same: automate as much as possible, catch issues early, and make deployments a non-event rather than a stressful, high-stakes process.



Continuous integration and continuous delivery (or deployment) are often discussed together, and it's therefore very common to refer to them as **CI/CD**.

Types of tests

Throughout this chapter, we've emphasized that testing is not a one-size-fits-all endeavor. While terms such as unit tests, integration tests, and E2E tests may already sound familiar, understanding their distinct roles is critical to building effective test suites. These different types of tests form a layered strategy to validate your application's reliability at every level, from individual functions to complex user workflows.

In this section, we'll break down the three pillars of testing: unit tests for isolating and validating individual components, integration tests for ensuring modules collaborate seamlessly, and E2E tests for simulating real-world user journeys.

We'll explore how these layers complement each other and why neglecting one risks invisible gaps in your coverage. Finally, we'll briefly examine specialized test types (such as smoke, performance, and contract testing) that address niche scenarios but remain valuable tools in a developer's arsenal.

Unit tests

Imagine walking into a bakery. The aroma of fresh bread, the flakiness of a croissant, the sweetness of a perfectly frosted cake – all these delights depend on the quality of their individual ingredients. Flour must be finely milled, butter properly churned, and sugar precisely measured. In software, functions are those foundational ingredients. Just as a single stale egg can ruin a cake, a flawed function can compromise an entire application. Unit tests are the types of tests that act as your quality control, rigorously validating each “ingredient” in isolation before it becomes part of a larger recipe.

Unit tests examine the smallest testable units of your code (functions, methods, or classes) in complete isolation. They answer one question: Does this single component behave as expected under controlled conditions?

If we imagine ourselves building an e-commerce platform, the Node.js backend might handle tasks such as calculating order totals, validating discounts, or formatting delivery dates. Unit tests would focus solely on individual functions, such as the following:

- A `calculateTotal()` function that sums item prices and applies tax
- A `validateCoupon()` function that checks whether a discount code is active

These tests ignore the broader context (databases, UIs, APIs, etc.) and instead focus solely on the logic within each function.

Here are some of the main qualities of unit tests:

- **Precision:** Unit tests are surgical. They pinpoint failures to specific lines of code, making debugging faster. If a test for `validateCoupon()` fails, you know exactly where to look; there is no need to sift through database queries or API calls.
- **Speed:** Without external dependencies, unit tests run in milliseconds. A suite of hundreds can execute in seconds, enabling rapid iteration.
- **Design discipline:** Writing testable units forces you to structure code modularly. Functions with clear inputs, outputs, and minimal side effects naturally emerge.
- **Living documentation:** Well-named tests (e.g., `applies_20_percent_discount_when_coupon_valid()`) document how components should behave, guiding future developers (or your forgetful future self).

When writing unit tests, you should find yourself asking questions such as: How can I isolate this logic? What edge cases could break this function? Does this test add value, or just check a checkbox?

Unit tests can add tremendous value to a codebase and give developers confidence, but they are not perfect. Their greatest strength (isolation) is also their limitation. While a single unit of code might work well in isolation, it might fail when integrated with other components. This is why unit tests alone are rarely sufficient, and where integration tests become essential.

Integration tests

In our bakery analogy, unit tests ensure that individual ingredients (e.g., flour, butter, yeast) are pristine. But a croissant's perfection hinges on how these ingredients interact: the dough must be laminated in precise layers, proofed at the right temperature, and baked to golden crispness. Integration tests are the baker's process checks, validating not just the quality of each ingredient but their synergy throughout the recipe.

Integration tests verify that components work together as intended, much like ensuring laminated dough layers rise uniformly and bake into a flaky texture. They answer the question: Do these parts collaborate correctly, even if they function perfectly alone?

Back to our e-commerce example. Unit tests validate isolated functions (e.g., `calculateTax()`), while integration tests ensure integration between components behaves correctly, as follows:

- The `createOrder()` function correctly deducts inventory from the database
- The payment module communicates with a third-party gateway without data mismatches
- Errors (e.g., expired credit cards) trigger proper rollbacks in the order workflow

While powerful, integration tests come with inherent complexity:

- **Setup overhead:** Unlike unit tests, they require spinning up databases, APIs, or third-party systems. For example, testing a payment flow might need a live database, a mock payment gateway, and a seeded inventory.
- **State management:** To ensure tests are independent and reliable, the system must reset to a known state before each test. This often involves cleaning up databases (e.g., truncating tables), restoring data from snapshots, reinitializing external services, etc.
- **Reproducibility:** Differences in environments (OS versions, database vendors, etc.) can cause “works on my machine” failures. Containers (e.g., Docker) mitigate this by packaging dependencies, but they add orchestration complexity.
- **Speed trade-offs:** Even with containers, spinning up and tearing down systems for every test is slow. A suite of integration tests might take minutes versus seconds for unit tests.

Integration tests are a great tool, but one you should use sparingly. Prioritize testing critical user workflows that directly impact core functionality. Validate essential functionalities such as checkout processes and authentication systems to ensure they operate reliably under expected conditions. Reserve edge case validation for unit tests, which are better suited for granular scenarios. Aim for a concise suite of integration tests (typically dozens, not hundreds) to maintain efficiency. A well-structured, focused set of tests will safeguard key interactions without overburdening your development cycle.

End-to-end tests

In our bakery analogy, unit tests validate ingredients, integration tests ensure the layers of a croissant rise harmoniously, and E2E tests are the moment the croissant is served to a customer. Does it arrive warm and flaky? Does the first bite deliver the promised buttery crunch? E2E tests simulate the full user journey, validating not just individual components or their interactions but the entire experience as a customer would encounter it.

E2E tests verify that your application behaves correctly from start to finish, mimicking real-world user actions. They answer the ultimate question: Does the system deliver value to users exactly as intended?

Take the following example:

- A user signs up, adds items to a cart, checks out, and receives a confirmation email.
- An API client submits a payment request and receives a validated transaction ID.



In testing terminology, **black-box** tests treat the application as an opaque system, unaware of its internals, while **white-box** tests leverage knowledge of its implementation details. However, this isn't a binary choice, but more like selecting a shade on a grayscale. Tests rarely fit neatly into one category; their placement depends on how much they rely (or don't rely) on the system's inner workings. E2E tests typically lean toward the black-box end of this spectrum. They focus on user-facing behavior, simulating real-world scenarios without deep coupling to code or infrastructure. That said, even E2E tests might occasionally incorporate some awareness of the system (e.g., seeding test data).

E2E tests excel by replicating real user interactions across your entire application, validating complete workflows, such as a customer ordering a croissant online, from browsing a menu to receiving a confirmation email. Unlike unit or integration tests, they don't just check isolated parts; they ensure every component works seamlessly together as it would in production. Their true strength lies in exposing hidden flaws (payment gateway mismatches, UI breaks from CSS changes, or database latency under real-world load) that technical tests often miss. By running in production-like environments, E2E can even catch environment issues such as misconfigured APIs or missing environment variables. They bridge the gap between theoretical correctness and real-world reliability, ensuring your system delivers exactly as users expect.

As you might imagine, all this power comes with a cost. There are, in fact, a few important challenges to deal with when writing E2E tests:

- **Complex setup:** E2E tests require full environments (databases, APIs, UIs, etc.), mirroring production. Tools such as Docker and Kubernetes help standardize setups, but orchestration remains time-consuming. Sometimes, it's preferred to run E2E tests in a remote environment that mimics production, which means that these environments need to be set up and kept up to date generally as part of CI/CD pipelines. These are non-trivial pieces of infrastructure to take care of. You also need to simulate the user environment, which often means using complex frameworks for browser automation or even devices (i.e., mobile devices) automation.
- **Brittleness:** A single UI change (e.g., updating the text or an attribute of a button element) can break tests. Teams must balance stability with adaptability. As much as we like to make E2E tests agnostic from the actual implementation, the reality is that we are still simulating a user journey through code, so every change to that user flow (even a subtle one) might break our simulation.
- **Ownership ambiguity:** In large teams, who "owns" tests spanning multiple services? Without clear governance, E2E tests become outdated or neglected.

- **Speed versus depth:** E2E tests are slow (minutes to hours) and expensive. Running hundreds of them is impractical.

E2E tests are the ultimate assurance that your application doesn't just work, it delivers. While their complexity and brittleness mean they can't replace unit or integration tests, their unique value lies in guarding your most critical user journeys (checkout flows, signups, and payment processing) where failures directly impact trust and revenue. The key is selectivity: a handful of well-crafted E2E tests, focused on high-stakes workflows and run in production-like environments, strike the optimal balance between effort and confidence.

Other types of tests

While unit, integration, and E2E tests are the pillars of most testing strategies, and the ones you'll encounter most frequently, they're far from the only tools in a developer's arsenal. Testing is ultimately about increasing confidence, and different challenges demand different approaches. Although this chapter focuses primarily on unit, integration, and E2E tests, let's briefly explore other common testing types to equip you with a broader perspective:

- **Performance testing:** Assesses how the system behaves under stress. Can your API handle 10,000 concurrent users without crashing?
- **Usability testing:** Evaluates user experience. Is the checkout flow intuitive, or do users abandon it in frustration?
- **Security testing:** Identifies vulnerabilities. Can attackers bypass authentication or inject malicious code?
- **Regression testing:** Ensures new changes don't break existing features. Does the latest update preserve login functionality?
- **Fuzz testing:** Bombards the system with chaotic or malformed inputs (corrupt files, scrambled network packets) to uncover crashes or unexpected behavior. Think of it as stress-testing a bridge with random seismic shocks.
- **Property testing:** Defines invariants (e.g., "Encoding and decoding a message should return the original input") and generates valid-but-diverse inputs to verify them. It's like a scientist testing hypotheses under controlled conditions.
- **Mutation testing:** A meta-testing technique that evaluates your test suite's effectiveness. Mutation testing tools intentionally introduce small bugs ("mutants") into the code. If your tests fail to detect these mutants, they're not thorough enough.

There are many other types of testing, such as compatibility testing, acceptance testing, exploratory testing, golden testing, and contract testing, but these are the most common.

While this chapter is focused on core techniques, remember that testing is a spectrum, not a checklist. Each type of test addresses unique risks, whether fortifying security, refining usability, or ensuring resilience under chaos. By understanding these options, you can craft a testing strategy as nuanced as the systems you build.

The testing pyramid

If unit, integration, and E2E tests are the ingredients of your testing strategy, the **testing pyramid** is the recipe that ensures they combine harmoniously. It answers a critical question: How do I balance these test types to build a suite that is fast, reliable, and maintainable?



The concept of the testing pyramid was popularized by Mike Cohn in his book *Succeeding with Agile*. While Mike's original test automation pyramid used slightly different terminology to categorize tests, the principles and ideas behind it remain timeless. In this section, we present a more modern version of the pyramid, updated to reflect terms more commonly used in today's software development landscape. The core philosophy, emphasizing granularity, speed, and balance, stays unchanged.

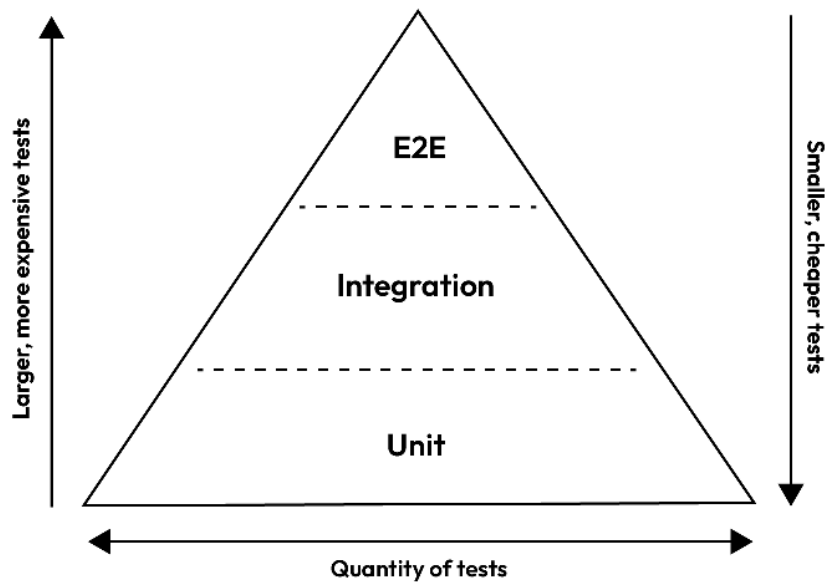


Figure 10.1 – A modern representation of the testing pyramid

The testing pyramid, as shown in *Figure 10.1*, helps teams strike a smart balance in their testing strategy. It's all about avoiding the trap of leaning too hard on one type of test, such as E2E tests, which, while powerful, can be brittle and time-consuming to keep up with. Instead, the pyramid guides you to catch most bugs early and cheaply at the “ground level” with unit tests. These fast, hyper-focused checks let you squash issues when they're small and simple to fix. At the same time, it doesn't ignore the big picture: a handful of well-chosen E2E tests at the top act as a final safety net, making sure everything clicks together the way real users expect.

This approach isn't just practical; it's efficient. By running tons of quick unit tests and saving the heavier E2E tests for only the most critical user journeys, you save time and effort while still covering your bases. The pyramid keeps your team moving fast, with unit tests giving instant feedback on tiny changes, and E2E tests stepping in to guard the most important workflows.

In summary, the testing pyramid suggests the following:

- **Many unit tests** to quickly verify individual pieces of code and catch breaks early
- **Fewer integration tests** to ensure key connections between components work smoothly
- **A handful of E2E tests** to validate critical user flows before shipping changes

This layered strategy keeps your code reliable without slowing you down.



While the testing pyramid is a valuable model, it isn't without its critics. Some argue that it undervalues what matters most to the business: the user experience. This critique has given rise to alternatives such as the **inverted testing pyramid**, which advocates for more E2E tests and fewer integration or unit tests.

In the JavaScript ecosystem, Kent C. Dodds popularized a fresh perspective called the **testing trophy** (nodejsdp.link/testing-trophy). This model prioritizes **Return on Investment (ROI)**: asking which tests deliver the most value per effort. Its mantra, *"Write tests. Not too many. Mostly integration,"* shifts focus toward a heavier emphasis on integration tests, with smaller but balanced suites of unit and E2E tests.

Writing tests with Node.js

We've extensively covered the wide world of testing, discussed the different types of tests, and highlighted guiding principles that help shape our testing mindset. Now, it's finally time to roll up our sleeves and write some code!

Our first unit test

Let's start with a simple example using an e-commerce function. We'll build a function called `calculateBasketTotal()` that takes a basket object as input. The basket contains an array of `items`, where each item has a `name`, `unitPrice`, and `quantity`. A little spoiler – our initial implementation contains a subtle bug that our test will later expose:

```
// calculateBasketTotal.js
export function calculateBasketTotal(basket) {
  let total = 0
  for (const item of basket.items) {
    total += item.unitPrice
  }
  return total
}
```

For our initial test, we are going to write a simple script that uses Node.js's built-in assert library to validate our code:

```
// test.js
import { equal } from 'node:assert/strict'
import { calculateBasketTotal } from './calculateBasketTotal.js'

// arrange
const basket = {
  items: [
    { name: 'Croissant', unitPrice: 2, quantity: 2 },
    { name: 'Olive bread', unitPrice: 3, quantity: 1 },
  ],
}

// act
const result = calculateBasketTotal(basket)

// assert
const expectedTotal = 7 // (2 * 2) + (3 * 1) = 7
equal(
  result,
  expectedTotal,
  `Expected total to be ${expectedTotal}, but got ${result}`
)
console.log('Test passed!')
```

The general behavior of this code should be relatively easy to understand, but let's highlight a few important details:

- We organize our tests into three distinct phases following the AAA methodology. First, the *Arrange* phase sets up the testing environment, in this case, initializing a basket with some items. Next, the *Act* phase executes the SUT (the `calculateBasketTotal()` function) against the prepared basket. Finally, the *Assert* phase validates whether the actual result matches the expected outcome, ensuring the logic behaves as intended. This structure keeps tests focused, readable, and aligned with the tested behavior.
- For assertions, we use Node.js's built-in `equal()` function from the `node:assert/strict` module. This function compares two values and throws an error if they differ, halting the test script entirely (since errors aren't caught). If the two values are equivalent, nothing happens. In other words: if our script exits successfully, our test passed; otherwise, it failed. The third parameter we pass to the `equal()` function is optional and represents a custom error message. It is good practice to provide a descriptive one to make it easier to understand what failed when an error is thrown.

- The `equal()` function from the `node:assert/strict` module considers the two given values equal if they are two primitive values (undefined, null, numbers, bigIntegers, strings, and symbols) with the same content. If the two values are objects, they need to be referencing exactly the same object for the assertion to pass. If they are two objects that are structurally equivalent (different objects in memory but with the same fields and the same values for every field), they won't be considered equal. If you need to compare two objects for structural equivalence, you can use the `deepEqual()` function. In short: use `equal()` for primitives or strict object identity checks; use `deepEqual()` to compare objects by their contents.



When importing `equal()` from `node:assert/strict`, you might wonder what the `strict` submodule does. In short, it enforces strict assertion mode, altering the behavior of assertion methods to avoid implicit type coercion, a common source of subtle bugs. For example, under `strict`, `deepEqual()` automatically behaves like `deepStrictEqual()`, meaning it checks both value and type (e.g., `5 !== '5'`). Non-strict assertions (like the original `deepEqual()`) are deprecated because they allow type juggling, which can hide potential bugs. By using `node:assert/strict`, you ensure all assertions follow strict rules, which is considered a best practice. An added benefit of strict mode is improved debugging: when assertions fail for objects, the error messages include a diff highlighting discrepancies between expected and actual values. This is especially helpful when troubleshooting complex data structures, as it pinpoints mismatches and reduces the need for manual inspection.

To run this test script, we can simply execute the following command in the terminal:

```
node test.js
```

When you run this file, you will see an error. The error indicates that the calculated total does not match our expectation:

```
AssertionError [ERR_ASSERTION]: Expected total to be 7, but got 5
5 !== 7
```

We told you there was a bug in our `calculateBasketTotal()` function! Let's have a second look at our function... Can you see the bug?

Yes, we forgot to multiply by the item quantity! Let's fix that:

```
// calculateBasketTotal.js
export function calculateBasketTotal(basket) {
  let total = 0
  for (const item of basket.items) {
    total += item.unitPrice * item.quantity // <- FIX
  }
  return total
}
```

After updating the code, if we run the test again, the test should pass and print **Test passed!** indicating that the bug has been fixed. Congrats, you have written your first test and fixed a bug too! Phew... this was a dangerous one. Imagine the monetary damage it could have caused in production. Thankfully, we caught it early with this test!

The Node.js test runner

Our first test was a simple Node.js script with a single assertion. While this works for very basic cases, it doesn't scale well as your test suite grows. Manually managing and running tests becomes tedious in larger projects.

A proper **testing framework** (or test runner) solves these common challenges by providing several useful features, such as the following:

- **Organizing tests:** Group related tests into files (usually one test file per source file) and keep tests isolated from one another.
- **Simplifying execution:** Run all tests (or a subset) with a single command. No need to run your own custom scripts one by one. During development, you can even run your tests in *watch mode*, which means that tests are automatically rerun if the source code changes.
- **Clear and standard results:** Show which tests passed, failed, or are still running with real-time feedback (e.g., color-coded results). Many frameworks can produce standard reporting outputs that can be used to integrate with other systems, such as reporting tools, such as Allure (nodejsdp.link/allure), for detailed analytics.
- **Mocking tools:** Simulate databases, APIs, third-party modules, or other dependencies easily.
- **Code coverage:** Track how much of your code is being executed during tests.
- **Spotting slow tests:** Identify tests that take too long to run.
- **Other advanced features:** Other features include snapshot testing (comparing outputs to saved “golden” versions), setup/teardown hooks, parametrized tests, and reusable test data (fixtures).

Popular frameworks such as **Mocha** (nodejsdp.link/mocha), **Jasmine** (nodejsdp.link/jasmine), **Jest** (nodejsdp.link/jest-mock), **Ava** (nodejsdp.link/ava), and **Vitest** (nodejsdp.link/vitest) offer most of these features and more.

Since version 18, Node.js includes an official test runner in the standard library (`node:test`). This built-in framework is powerful and modern, giving you everything you need to write tests directly in Node.js without relying on external dependencies.

While this chapter focuses on the built-in test runner, the concepts you'll learn (such as organizing tests, using assertions, mocks, etc.) can be applied with very little effort to more fully featured libraries such as Ava or Vitest, too, so don't be afraid to give these a try!

Our first test with the Node.js test runner

Here's what our test script would look like if we used the Node.js test runner:

```
// calculateBasketTotal.test.js // 1
import { equal } from 'node:assert/strict'
import { test } from 'node:test' // 2
import { calculateBasketTotal } from './calculateBasketTotal.js'

test('Calculates basket total', () => { // 3
  const basket = {
    items: [
      { name: 'Croissant', unitPrice: 2, quantity: 2 },
      { name: 'Olive bread', unitPrice: 3, quantity: 1 },
    ],
  }

  const result = calculateBasketTotal(basket)

  const expectedTotal = 7
  equal(
    result,
    expectedTotal,
    `Expected total to be ${expectedTotal}, but got ${result}`
  )
})
```

This code isn't very different from our custom test script, but let's break down some important changes and discuss the structure we need to adopt when using the Node.js test runner:

1. The file name follows the convention `<moduleName>.test.js` (i.e., `calculateBasketTotal.test.js`). By default, the Node.js test runner executes all files ending with `.test.js` when running the test suite. This convention standardizes test file naming. Test files can reside anywhere in your project. They don't need to be grouped in a `test` folder (though you could do that if you like). It's common to place test files alongside their corresponding modules (e.g., `module.js` and `module.test.js` in the same directory).
2. We import the `test()` function from the `node:test` module. This function is the entry point for defining executable tests.
3. The actual test logic is now wrapped in an anonymous function passed to the runner `test()` function (we'll refer to this as the **function under test**, or **FUT**, from now on). The first argument provides a descriptive name for the test (`'Calculates basket total'`), while the FUT contains the test implementation. The test runner executes the FUT and reports its outcome automatically. Multiple tests can be added in the same file by defining additional `test()` blocks.

One final detail to note is that we removed the line `console.log('Test passed!')` from our test code. This is not needed anymore since the test runner will tell us whether the test failed or succeeded.

To run the test, execute this command in your terminal:

```
node --test
```

This produces output such as the following:

```
✓ Calculates basket total (0.422583ms)
i tests 1
i suites 0
i pass 1
i fail 0
i cancelled 0
i skipped 0
i todo 0
i duration_ms 46.738458
```

While running the test, the Node.js framework provides real-time, interactive output in your terminal. As tests execute, results update progressively: passing tests display a checkmark with their execution time, while failures show a red cross and detailed error messages. The final summary includes stats such as total tests, passes, failures, and total duration.

And there you have it: your first test using the Node.js test runner! While this example covers the basics, the framework offers far more powerful tools to streamline complex scenarios. In the next sections, we will deep dive into some of its most important capabilities.

Organizing tests

We have already seen the `test()` function in action, but let's now dig a little bit deeper.

We mentioned that we can add multiple `test()` functions in a single test file, so let's start by trying to do exactly that. Let's add a new test for our `calculateBasketTotal()` function:

```
// calculateBasketTotal.test.js
// ...
test('Calculates basket total', () => {
  // ...
})

test('Calculates basket total with no items', () => {
  const basket = {
    items: [],
  }
})
```

```
const result = calculateBasketTotal(basket)

const expectedTotal = 0
equal(
  result,
  expectedTotal,
  `Expected total to be ${expectedTotal}, but got ${result}`
)
})
```

With this new test, we are trying to answer the question: How does our function behave with an empty basket? We expect the total to be 0, and this test should tell us if our current implementation is correct. We can run the following again:

```
node --test
```

We should get the following output:

```
✓ Calculates basket total (0.332958ms)
✓ Calculates basket total with no items (0.057791ms)
i tests 2
i suites 0
i pass 2
i fail 0
i cancelled 0
i skipped 0
i todo 0
i duration_ms 44.303791
```

Great, all green!

Note how the test runner reports the result of both tests using the descriptive names we provided when using the `test()` function.

The function we pass to `test()` to define the logic of a test (the FUT) can be defined in different ways:

1. A synchronous function that is considered failing if it throws an exception and is considered passing otherwise. This is exactly what we have done so far.
2. A function that returns a Promise that is considered failing if the Promise rejects and is considered passing if the Promise fulfills. This means that the FUT can also be an async function.
3. A function that receives a callback function (generally named `done()`). If the callback receives any *truthy* value (generally an `Error` object) as its first argument, the test is considered failing. If a *falsy* value is passed as the first argument to the callback, the test is considered passing. If the test function receives a callback function and also returns a Promise, the test will fail with a descriptive error saying **passed a callback but also returned a Promise**.

Here are some dummy code examples that represent how you can practically implement these three options:

```
// 1
test('passing sync test', t => {})
test('failing sync test', t => {
  throw new Error('fail')
})

// 2
test('passing async test with promise', t => Promise.resolve())
test('failing async test with promise', t => Promise.reject(new Error('fail')))

test('passing async test with async', async t => {})
test('failing async test with async', async t => {
  throw new Error('fail')
})

// 3
test('passing async test with callback', (t, done) => done())
test('failing async test with callback', (t, done) => done(new Error('fail')))

test('invalid test: both cb and promise (async)', async (t, done) => {})
```

Did you notice the `t` argument we used in every FUT defined in the preceding example? This is a `TestContext` object that gets automatically passed by the test runner into our FUT. It can be used to perform actions related to the current test. Examples include reading information about the current test (e.g., the test name), adding additional diagnostic information, or creating subtests.

Subtests

The idea of creating subtests is particularly interesting and can be useful under a couple of circumstances:

- Organize tests under a hierarchical structure
- Define test cases programmatically based on an array of inputs or programmatically generated test data

Here's a simple example of how we can define subtests:

```
import { test } from 'node:test'

test('Top level test', t => {
  t.test('Subtest 1', t => {
    // ...
  })
})
```



```

    })

    t.test('Subtest 2', t => {
      // ...
    })
  })
})

```

If we run this test, we will see the following output:

```

▶ Top level test
  ✓ Subtest 1 (0.291375ms)
  ✓ Subtest 2 (0.047292ms)
✓ Top level test (0.722416ms)
i tests 3
i suites 0
i pass 3
i fail 0
i cancelled 0
i skipped 0
i todo 0
i duration_ms 55.523959

```

This structure is particularly useful for grouping related test cases within large test files, improving readability and organization.



If you're using a version of Node.js earlier than 24, make sure to explicitly await subtests inside an async parent test. Otherwise, the parent might finish before its children, canceling unfinished subtests and leading to errors such as the following:

```
'test did not finish before its parent and was cancelled'
```

Starting with Node.js 24, subtests are awaited automatically, so this is no longer necessary.

Subtest concurrency

In the previous example, our subtests are running sequentially: *Subtest 1* is executed first, and only when it finishes is *Subtest 2* started. This is OK for simple and fast tests, but in the reality of production projects, this approach might mean that all your subtests take a long time to complete.

A much better approach is to enable concurrency between subtests. Here's how we can do that:

```

import { test } from 'node:test'

test('Top level test', { concurrency: true }, t => { // 1
  t.test('Subtest 1', _t => { // 2
    // ...
  })
})

```

```

    })

    t.test('Subtest 2', _t => { // 2
      // ...
    })
  })
})

```

If we compare this example with the previous one, there are a couple of small but important differences here:

1. When we call the `test()` function, we pass a second parameter, just before we pass our FUT. This parameter is optional, and it allows us to configure the test. One of the configuration options is concurrency. This option allows us to enable subtest concurrency. It can be set to a number to explicitly specify a maximum concurrency, or to `true` to tell the Node.js test runner to pick an optimal level of concurrency based on the current system. Setting it to `false` is equivalent to setting it to 1.
2. We don't need to explicitly await each test anymore. When running tests concurrently, the Node.js test runner will keep track of the running tests and won't cancel them. This also means that our top-level FUT doesn't need to be async.

Pattern: Enable subtest concurrency

Every time you use subtests, you most likely want to enable concurrency. This will keep your tests fast and your code simpler, since you won't have to worry about awaiting each subtest.



Note that when you have multiple test files, the Node.js test runner executes those files concurrently. You can control this behavior with the `--test-concurrency` (e.g., `--test-concurrency=1` to disable concurrency) flag when running `node --test`. This setting can come in handy in a couple of situations. If you happen to have tests that are interdependent and need to run in a specific order (not ideal, but sometimes unavoidable), disabling concurrency ensures they're executed sequentially. On the other hand, if you're running your tests on a powerful CI/CD machine with plenty of CPU cores, increasing the concurrency level can help you get faster feedback by fully leveraging available resources.

Parametrized test cases

Now, let's see how we can use subtests to create *parameterized* test cases: tests that run the same logic multiple times with different inputs. This approach is sometimes also referred to as creating *programmatically* test cases, since the individual cases are generated through code rather than written out manually. To do that, let's refactor our previous `calculateBasketTotal()` test suite:

```

import { equal } from 'node:assert/strict'
import { test } from 'node:test'
import { calculateBasketTotal } from './calculateBasketTotal.js'

test('Calculates basket total', { concurrency: true }, t => {

```

```

const cases = [ // 1
  {
    name: 'Empty basket',
    basket: { items: [] },
    expectedTotal: 0,
  },
  {
    name: 'One croissant',
    basket: { items: [{ name: 'Croissant', unitPrice: 2, quantity: 1 }] },
    expectedTotal: 2,
  },
  {
    name: 'Two croissants and one olive bread',
    basket: {
      items: [
        { name: 'Croissant', unitPrice: 2, quantity: 2 },
        { name: 'Olive bread', unitPrice: 3, quantity: 1 },
      ],
    },
    expectedTotal: 7,
  },
]

for (const { name, basket, expectedTotal } of cases) { // 2
  t.test(name, () => {
    const result = calculateBasketTotal(basket)
    equal(
      result,
      expectedTotal,
      `Expected total to be ${expectedTotal}, but got ${result}`
    )
  })
}
})

```

There are two main parts to discuss in this example:

1. The first thing we do in our top-level test (Calculates `basket total`) is to define a series of cases in an array. Every case is an object with a `name`, a `basket` (the input), and an `expectedTotal` (the expected output).
2. Then we loop over all the cases, and for every case, we *generate* a subtest by calling `t.test()`.

As you can see, this approach allows us to easily add new test cases without having to repeat the actual test logic multiple times. Did you see that we added an additional test case (One croissant) from our previous version of this test? If you come up with other cases that you want to test, you can simply add them to the cases array.

If we run this test, the output will look like this:

```
▶ Calculates basket total
  ✓ Empty basket (0.281334ms)
  ✓ One croissant (0.078083ms)
  ✓ Two croissants and one olive bread (0.100333ms)
✓ Calculates basket total (1.018709ms)
i tests 4
i suites 0
i pass 4
i fail 0
i cancelled 0
i skipped 0
i todo 0
i duration_ms 41.448042
```

The execution of the generated tests is nicely reported following the hierarchical structure we defined by using subtests.

Test suites

There's another way to organize your tests in a nice hierarchy within each test file. This is by using the `suite()` function, as illustrated in the following example:

```
import { test, suite } from 'node:test'

suite('Top level suite', { concurrency: true }, () => {
  test('Subtest 1', () => {})
  test('Subtest 2', () => {})
})
```

In this example, we are defining the tests and organizing them into a suite.

Note that both the `suite()` and `test()` functions have aliases. You can swap `suite()` with `describe()` and `test()` with `it()`; therefore, the following code is exactly equivalent to the preceding one:

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

describe('Top level suite', { concurrency: true }, () => {
  it('Test 1', () => {})
  it('Test 2', () => {})
})
```

These different aliases are provided to allow you to write tests using naming practices that are common in other testing frameworks. For example, Jasmine users typically use the `describe/it` nomenclature, while Vitest users typically use `describe/test`. Feel free to pick whichever naming resonates the most with you and your team.

Test runner tips and tricks

In this section, we introduce a rapid-fire sequence of additional tips and tricks that you should be aware of to make the most of the Node.js test runner.

Watch mode

Rapid feedback is essential when writing tests. Manually switching windows to rerun `node --test` after every change disrupts your flow and slows progress.

One simple yet powerful trick you can use to maximize your focus and productivity is to enable **watch mode** using the `--watch` flag, as follows:

```
node --test --watch
```

Here's what happens when you enable watch mode:

- The test suite runs once initially.
- The process stays alive, monitoring your test files and any imported modules.
- On saving changes to these files, the tests automatically re-execute.

This creates a tight feedback loop: write code, save, and instantly see results without leaving your editor. No more manual restarts and no context switching, just continuous validation as you iterate on your code.

Executing a subset of all your tests

As your codebase grows, so does your test suite. Waiting for hundreds of tests to complete during active development slows your feedback loop, especially when you only need to validate specific functionality. Node.js's test runner offers targeted execution to keep you productive.

Default test discovery

When running `node --test` without arguments, the runner automatically discovers files matching these glob patterns:

- `**/*.test.{cjs,mjs,js}`
- `**/*-test.{cjs,mjs,js}`
- `**/*_test.{cjs,mjs,js}`
- `**/test-*. {cjs,mjs,js}`
- `**/test.{cjs,mjs,js}`
- `**/test/**/*.{cjs,mjs,js}`

These glob patterns cover the most common test file naming conventions, including the following:

- `.test.js` suffix files (the convention we have used so far in this chapter)
- `-test.js` and `_test.js` suffix files
- Files in `/test` directories
- `test-*.js` prefix files

Targeted test execution with custom glob patterns

For focused development, you can specify custom glob patterns to run only relevant tests for your current tasks. Take the following example:

```
node --test "shoppingCart/**/*.test.js" "checkout/**/*.test.js"
```

This command does the following:

1. Runs all files with the suffix `.test.js` in `shoppingCart/` and its subdirectories
2. Runs all files with the suffix `.test.js` in `checkout/` and its subdirectories
3. Ignores other tests outside these paths

There are a few scenarios where you want to use custom glob patterns:

- **Feature work:** Test only the module you're modifying (e.g., `payment/**/*.test.js`)
- **Custom conventions:** If your team uses non-standard test file names (e.g., `*.spec.js`)
- **Debug sessions:** Isolate failing tests during a troubleshooting session

By combining targeted execution with watch mode, you can create a laser-focused development environment that responds instantly to changes.

Filter tests by test name

Another thing you can do is to filter test execution based on test names or test suite names. This can be done using the `--test-name-pattern` and `--test-skip-pattern` flags. These flags are quite flexible, and their behavior is best described with some examples.

Let's assume we are working with the following test file:

```
suite('Top level suite 1', { concurrency: true }, () => {  
  test('Test 1', () => {})  
  test('Test 2', () => {})  
})  
  
suite('Top level suite 2', { concurrency: true }, () => {  
  test('Test 1', () => {})  
  test('Test 2', () => {})  
})
```

Now, let's see some examples of how we can use these two flags to run different subsets of tests:

- `node --test`: Runs all tests
- `node --test --test-name-pattern="suite 2"`: Runs only Test 1 and Test 2 from suite 2
- `node --test --test-name-pattern="suite 2 Test 2"`: Runs only Test 2 from suite 2
- `node --test --test-name-pattern="Test 2"`: Runs Test 2 from both suite 1 and suite 2
- `node --test --test-skip-pattern="Test 2"`: Skips Test 2 from both suite 1 and suite 2
- `node --test --test-skip-pattern="suite 1"`: Skips all the tests in suite 1

Note how we can easily match on both suite names and test names, and we don't even have to provide the entire name of a suite or a test (see how we used "suite 1" and we didn't have to specify "Top level suite 1"). A simple case-sensitive string match will do the trick!

Filter tests with skip and only

Sometimes it is easier to skip tests or test suites directly in code rather than having to figure out which combination of file name patterns or test name patterns to use. You can do that by setting the skip option to true for a suite or a test. Take the following example:

```
suite('Top level suite', { concurrency: true, skip: true }, () => {  
  test('Test 1', () => {})  
  test('Test 2', () => {})  
})
```

This will skip this entire suite when running all your tests. If instead we only want to skip a test, for example, Test 2, we can do something like this:

```
suite('Top level suite', { concurrency: true }, () => {  
  test('Test 1', () => {})  
  test('Test 2', { skip: true }, () => {})  
})
```

This can be convenient, especially if you are dealing with a flaky test or a suite that you want to disable momentarily.

A similar option is `only`. When you set `only` to `true` on some tests or suites, the test runner will execute only these tests and filter out everything else. Be aware that for this option to work, you also need to pass the `--test-only` flag when running the tests.



A `todo` option exists too. This option provides a structured way to track unimplemented tests, acting as a graceful skip with intentionality. Unlike `skip` (which implies temporary disabling), `todo` explicitly marks tests as planned work.

Both `todo` and `skip` accept strings (as an alternative to `true`) to document because a test is pending. The given string will be reported in line with the test results by the test runner, providing context when someone reads the test results.

If you prefer not to use the test configuration option, you can also use the `TestContext` object to mark a test as `skip` or `todo`:

```
test('A skipped test', t => {
  t.skip(true)
})

test('A todo test', t => {
  t.todo(true)
})
```

Finally, there are a few other shortcuts you can use; you can call the `.skip()`, `.todo()`, and `.only()` methods directly on suite and test:

```
suite.skip('Skipped suite', () => { /* ... */ })
suite.only('Exclusive suite', () => { /* ... */ })
suite.todo('To do suite', () => { /* ... */ })

suite('Top level suite', () => {
  test.skip('Skipped test', () => { /* ... */ })
  test.only('Exclusive test', () => { /* ... */ })
  test.todo('To do test', () => { /* ... */ })
})
```

You have lots of syntactic options to mark tests as skipped, exclusive execution, or to-do. There isn't really a recommended one, so just pick the one that seems most natural to you and be consistent.

Test reporters

A **test reporter** determines how Node.js formats and displays your test results. While the default reporter works for most cases, different scenarios might demand different output formats: from human-readable summaries during development to machine-parsable reports for CI systems.

Test reporters can be specified with the `--test-reporter` flag. At the time of writing, there are five different built-in reporters:

- **spec**: The default one. We have already seen its output in action. It's a human-readable output that showcases the test tree using indentation and the various test results with colors and status symbols (✓/✗). It's ideal for development workflows.
- **tap**: The **TAP**, or **Test Anything Protocol** (nodejsdp.link/tap), output is a widely supported format that provides structured output that is both human-readable and machine-readable. It can be integrated with a variety of CI pipelines and other third-party tools.
- **dot**: Prints a dot (.) for passing tests and an X for failures in a compact stream. It is useful for large test suites where you just want progress feedback.
- **junit**: Produces JUnit XML files compatible with services such as Jenkins or CircleCI. Requires `--test-reporter-destination` to specify the output file.
- **lcov**: Generates coverage reports in LCOV format when used with `--experimental-test-coverage`. We'll discuss this one in more detail in the next section dedicated to coverage.

Here's an example of how you can use one of these test reporters:

```
node --test --test-reporter=dot
```

You can also use multiple test reporters to serve different needs simultaneously:

```
node --test \  
  --test-reporter=spec \  
  --test-reporter=junit \  
  --test-reporter-destination=console \  
  --test-reporter-destination=results.xml
```

This configuration prints human-friendly spec output to the console and writes CI-compatible JUnit results to `results.xml`.



If you have very specific needs, you can also create your own custom reporters. Check out the documentation if you are curious to learn more: nodejsdp.link/custom-reporter.

Collecting code coverage

As you write more tests, you might be wondering whether there are parts of your code that are not exercised during the tests. This can be entire modules, conditional branches, or just specific lines that you might have forgotten to test. Code coverage reports can help you identify these cases and pinpoint gaps in your current testing suite.

Node.js allows you to collect coverage information when executed with the `--experimental-test-coverage` flag, as in the following example:

```
node --test --experimental-test-coverage
```



As you can tell from the name of this flag, this feature is experimental at the time of writing. Chances are that when you are reading this, this flag will have been stabilized and renamed to `--test-coverage`. You can verify the current syntax in the official documentation (nodejsdp.link/coverage).

If you want to limit the coverage report to specific folders or files, you can do that with the `--test-coverage-include` and `--test-coverage-exclude` options, as in the following examples:

```
# only collect files for *.js files in src
node --test --experimental-test-coverage --test-coverage-include="src/*.js"

# exclude all *.js files in someModule
node --test --experimental-test-coverage --test-coverage-exclude="someModule/*.js"
```

Code coverage can also be disabled for sections of a file with special in-code comments. Take the following example:

```
/* node:coverage disable */
if (falsyCondition) {
  console.error('this should never happen')
}
/* node:coverage enable */
```

The comments `/* node:coverage disable */` and `/* node:coverage enable */` allow you to delimit entire sections of your code as something to be ignored for the purposes of coverage reporting. You can also ignore a single line or a number of sequential lines with `/* node:coverage disable next [N] */`, as in the following examples:

```
/* node:coverage ignore next */
if (falsyCondition) { console.log('this is never executed') }

/* node:coverage ignore next 3 */
if (falsyCondition) {
  console.log('this is never executed')
}
```



Be careful if you use code formatters such as Standard, ESLint, Prettier, or Biome. When you run them, they can reorganize the structure of your code. For example, they might turn a one-liner if statement into an equivalent expression spread across multiple lines. When they do this, they (most likely) won't update your coverage comments accordingly, and this might lead to inaccurate coverage reports. For this reason, we generally consider it safer to use the `/* node:coverage disable */` and `/* node:coverage enable */` delimiters.

Visualizing coverage in the console and the browser

The default test reporter displays the test coverage report as a table directly in the console. A simple coverage report might look like this:

```
i start of coverage report
i -----
i file | line % | branch % | funcs % | uncovered lines
i -----
i calculateBasketTotal.js | 71.43 | 66.67 | 100.00 | 4-5
i -----
i all files | 71.43 | 66.67 | 100.00 |
i -----
```

With small codebases, you can quickly open the file, check those lines, and decide whether to add missing tests and improve your coverage.

However, as your project grows, mapping these line numbers back to complex logic spread across multiple files becomes tedious. Is line 5 part of an error handler? A rarely triggered edge case? Scrolling through files to find these answers will slow down your workflow.

This is where the built-in **lcov** reporter helps. You can generate lcov reports with the following command:

```
node --test --experimental-test-coverage --test-reporter=lcov --test-reporter-destination=lcov.info
```

This generates an `lcov.info` file in your current working directory.



lcov is a standard format (nodejsdp.link/lcov) with a rich ecosystem of tools and integrations. Modern code editors have support for lcov (either directly or through dedicated plugins). With this support, coverage information is shown directly in your source code using color codes: green for covered lines, yellow and red for gaps. There are also tools (such as nodejsdp.link/lcov-viewer-cli) that can take your lcov files and convert them into a rich, browsable HTML report.

An alternative approach is to use a third-party tool called **c8** (nodejsdp.link/c8). Once you install c8 in your project, you can generate a rich HTML report with the following command:

```
npx c8 -r html node --test --experimental-test-coverage
```

These HTML reports are invaluable for team reviews or tracking coverage trends over time.



The report will be saved in the coverage folder, so remember to add this folder to your `.gitignore` file and treat it as a temporary artifact.

Using the test runner with TypeScript

The Node.js test runner natively supports TypeScript test files. Your `.test.ts` files can import TypeScript source modules directly. No transpilation step is required.

By default (unless using `--no-experimental-strip-types`), the runner detects the following TypeScript file glob patterns:

- `**/test/**/*-test.{cts,mts,ts}`
- `**/test/**/*_test.{cts,mts,ts}`
- `**/test/**/*_test.{cts,mts,ts}`

This is a little inconsistent with the other JavaScript patterns we presented before, because, by default, it assumes your tests are placed within a `test` folder in the project structure.

Thankfully, as discussed before, we can provide our custom patterns. So, if we imagine rewriting our `calculateBasketTotal.js` example (and the related test file) in TypeScript, we can run all the tests with the following command:

```
node --test '**/*.test.ts'
```

The provided glob pattern matches any file ending with `.test.ts`. With this pattern, our test files don't need to be within a `test` folder.

If we also want to capture coverage information, this is how we can do it:

```
node --test --experimental-test-coverage --test-coverage-exclude='**/*.test.ts'
 '**/*.test.ts'
```

You might notice that we are repeating the glob pattern twice. The reason is that we don't want to see coverage information for the test files themselves, but only for our modules, so we must explicitly exclude test files from coverage collection.



A more detailed example is available in the repository associated with this book:
nodejsdp.link/test-ts.

Writing unit tests

We've already written some unit tests for our `calculateBasket.js` module as part of our introduction to testing. This simple example showcased synchronous API testing, which proved to be an ideal starting point. The reason it was so straightforward is that our `calculateBasketTotal` function is both synchronous and pure: it relies on no external dependencies and produces no side effects, simply taking a clear input and returning an output.

While this simplicity makes for great learning material, real-life JavaScript projects often diverge from this ideal. We'll explore the challenges that arise when testing asynchronous code and functions that rely on external dependencies or produce side effects. For instance, in many applications, we use callbacks, promises, or `async/await` to interact with external systems such as databases. These interfaces introduce complexities that can make testing more difficult.

Another reason real-life projects are more complex is that our code often relies on external libraries or services to function correctly. This can include database clients, APIs, or other third-party dependencies. In these cases, our tests must account for the interactions between our code and these external systems, which can be tricky to replicate in isolation.

In this section, we'll delve into the challenges of testing real-life scenarios and explore effective techniques and patterns to help you write robust unit tests for your JavaScript projects.

Testing asynchronous code

A practical example of asynchronous testing in action is the `TaskQueue` class we implemented and used in *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*, and *Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await*. In our web spider project, this class proved invaluable as it provided an abstraction that enabled the efficient execution of asynchronous operations by running them concurrently while respecting a specified maximum concurrency level.

Let's have another look at the code:

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

export class TaskQueue extends EventEmitter {
  constructor(concurrency) {
    super()
    this.concurrency = concurrency
    this.running = 0
    this.queue = []
  }

  pushTask(task) {
    this.queue.push(task)
    process.nextTick(this.next.bind(this))
    return this
  }

  next() {
    if (this.running === 0 && this.queue.length === 0) {
      return this.emit('empty')
    }
  }
}
```

```

    while (this.running < this.concurrency && this.queue.length > 0) {
      const task = this.queue.shift()
      task()
        .catch(err => {
          this.emit('taskError', err)
        })
        .finally(() => {
          this.running--
          this.next()
        })
      this.running++
    }
  }

  stats() {
    return {
      running: this.running,
      scheduled: this.queue.length,
    }
  }
}

```

By reviewing this code, we can come up with a high-level specification for what this class does:

- When we initialize an instance, we need to provide the desired maximum concurrency.
- We can then use the `pushTask()` method to add a new task (in the form of a function) to the queue.
- Tasks added to the queue should be executed to completion, respecting the maximum level of concurrency.
- If a task fails, a `taskError` event is emitted.
- When all the tasks are completed, an `empty` event is emitted.
- We can call the `stats()` method at any time to know how many tasks are running and how many are scheduled.

How do we know that this specification is implemented correctly? Let's write some tests!

This is the initial structure for our test suite:

```

import assert from 'node:assert/strict'
import { once } from 'node:events'
import { suite, test } from 'node:test'
import { setImmediate } from 'node:timers/promises'
import { TaskQueue } from './TaskQueue.js'

```

```
suite('TaskQueue', { concurrency: true, timeout: 500 }, () => {
  test.todo('All tasks are executed and empty is emitted', async () => {})
  test.todo('Respect the concurrency limit', async () => {})
  test.todo('Emits "taskError" on task failure', async () => {})
  test.todo('stats() returns correct counts', async () => {})
}))
```

In this initial skeleton, we're using `suite()` to group all our tests together. This helps us organize them nicely and run them in parallel, which is great for keeping test runs fast. We've also added a timeout of 500 ms to catch any test that might hang. This is a useful tactic when working with async code, since a missed `await` or a logic bug could otherwise leave the test hanging forever. Notice how we're using `.todo()` for each test. Remember that this tells the test runner that these are placeholders for now. If you run the suite, everything will pass, but you'll see the *TODO* labels in the output, reminding you of what's left to do. Alright, let's go ahead and write the first test!

```
test('All tasks are executed and empty is emitted', async () => {
  const queue = new TaskQueue(2)
  const task1status = Promise.withResolvers()
  let task1Completed = false
  const task2status = Promise.withResolvers()
  let task2Completed = false

  const task1 = async () => {
    await setImmediate()
    task1Completed = true
    task1status.resolve()
  }
  const task2 = async () => {
    await setImmediate()
    task2Completed = true
    task2status.resolve()
  }

  queue.pushTask(task1).pushTask(task2)
  await Promise.allSettled([task1status.promise, task2status.promise])

  assert.ok(task1Completed, 'Task 1 completed')
  assert.ok(task2Completed, 'Task 2 completed')
  await once(queue, 'empty')
})
```

In this first test, we're checking that all the tasks we add to the queue actually get executed and that the `empty` event is emitted once the queue finishes processing everything.

We start by creating a `TaskQueue` with a concurrency of 2, so it can run both tasks at the same time. For each task, we set up a little mechanism using `Promise.withResolvers()` to track when the task completes. Inside each task, we simulate asynchronous work using `setImmediate()`, mark the task as completed, and then resolve the corresponding tracking promise.

Once we push both tasks to the queue, we wait for both tracking promises to settle; this tells us the tasks actually ran. Then, we assert that both flags (`task1Completed` and `task2Completed`) are true. Finally, we wait for the empty event from the queue, which should only be emitted once all tasks are done; if not, the test will be considered failed within 500 ms. This gives us good confidence that our `TaskQueue` is working correctly from end to end.



`Promise.withResolvers()` is a handy utility that gives you direct access to a promise's `resolve()` and `reject()` functions. It's perfect in tests when you want to manually control or observe when a specific async operation completes, just like we're doing here to track each task's completion.

Let's now focus on the next test:

```
test('Respect the concurrency limit', async () => {
  const queue = new TaskQueue(4)
  let runningTasks = 0
  let maxRunningTasks = 0
  let completedTasks = 0
  const task = async () => {
    runningTasks++
    maxRunningTasks = Math.max(maxRunningTasks, runningTasks)
    await setImmediate()
    runningTasks--
    completedTasks++
  }

  queue
    .pushTask(task)
    .pushTask(task)
    .pushTask(task)
    .pushTask(task)
    .pushTask(task)
  await once(queue, 'empty')

  assert.equal(maxRunningTasks, 4)
  assert.equal(completedTasks, 5)
})
```


In this test, we want to make sure that the `TaskQueue` respects the concurrency limit we set. We configure it with a concurrency of 4, which means it should never run more than four tasks at the same time.

To check this, we track how many tasks are running at any given moment using a `runningTasks` counter. Each time a task starts, we increment it, and each time a task completes, we decrement it. We also keep track of the maximum number of concurrently running tasks using `maxRunningTasks`.

We then schedule five tasks in total. Since the concurrency limit is four, the queue should process the first four tasks concurrently, and only start the fifth after one of the first ones finishes.

After waiting for the empty event to signal that all tasks have finished, we assert two things: that all five tasks have completed and that `maxRunningTasks` is exactly 4. This confirms that the queue is correctly enforcing the concurrency limit.



In this test, we're asserting that the maximum number of concurrently running tasks is exactly 4, which aligns with the concurrency level we set. This works reliably in our case because the mocked task completes predictably using `setImmediate`, ensuring no task finishes before the next one is scheduled. In more realistic or less predictable scenarios (where task durations may vary or resolve immediately), it might be harder to guarantee that the concurrency limit is fully saturated at any given moment. In such cases, a more flexible approach would be to assert that the maximum concurrency observed is less than or equal to the configured limit, rather than assuming it will reach full utilization.

Let's now check that errors are reported correctly:

```
test('Emits "taskError" on task failure', async () => {
  const queue = new TaskQueue(2)
  const errors = []
  queue.on('taskError', error => {
    errors.push(error)
  })

  queue.pushTask(async () => {
    await setImmediate()
    throw new Error('error1')
  })
  queue.pushTask(async () => {
    await setImmediate()
    throw new Error('error2')
  })
  await once(queue, 'empty')

  assert.equal(errors.length, 2)
  assert.equal(errors[0].message, 'error1')
```

```
    assert.equal(errors[1].message, 'error2')
  })
```

This test verifies that the `TaskQueue` emits a `taskError` event whenever a task fails. We start by creating a queue with a concurrency of 2 and setting up an event listener to collect any errors that are emitted.

We then push two failing tasks into the queue. Each one uses `setImmediate()` to simulate async work and then throws an error with a different message.

After waiting for the empty event (which means all tasks have been processed), we check that we received exactly two errors and that the messages match what we expected. This confirms that the queue doesn't silently swallow errors and correctly notifies listeners when something goes wrong. It's also important to note that, by checking for two errors instead of just one, we're also verifying that the queue continues processing the remaining tasks even after a failure: something that's especially important for robustness in real-world async workflows.

Let's now look at our final test in this suite:

```
test.todo('stats() returns correct counts', async () => {
  const queue = new TaskQueue(1)
  const task = async () => {
    await setImmediate()
  }

  queue.pushTask(task).pushTask(task)
  await setImmediate()

  assert.deepEqual(queue.stats(), { running: 1, scheduled: 1 })
  await once(queue, 'empty')
  assert.deepEqual(queue.stats(), { running: 0, scheduled: 0 })
})
```

This test checks that the `stats()` method gives us an accurate snapshot of what's happening inside the queue at any point in time. We create a queue with a concurrency of 1, meaning it will only run one task at a time, and we push two identical async tasks into it.

Right after pushing the tasks, we await `setImmediate()` to give the queue a chance to start processing the first task. At that point, we expect one task to be running and one to still be waiting in the queue, so `stats()` should return `{ running: 1, scheduled: 1 }`.

Then, we wait for the empty event, which tells us that both tasks have finished. At that point, `stats()` should return `{ running: 0, scheduled: 0 }`, confirming that the internal counters are being properly updated throughout the lifecycle of the queue. This kind of visibility into the queue's internal state is super helpful when debugging or building more advanced logic on top of it.

That wraps up our example of testing asynchronous code! We've seen how to verify task execution, concurrency limits, error handling, and internal state tracking; all are essential when working with async flows. In the next section, we'll shift gears and explore how to use mocks in our unit tests to isolate behavior, control dependencies, and make our tests even more focused and reliable.

Mocking

In this section, we're going to dive into the world of **mocking**. We'll start by learning how to create spies using `mock.fn()`, which allows us to track function calls without changing their behavior. From there, we'll explore how mocks can help us manage dependencies more effectively, making it easier to test code in isolation. Along the way, we'll see how to mock things such as HTTP requests and even entire module imports. We'll also take a closer look at the differences between mocking imports and using DI, so we can make informed choices about which approach fits best depending on the situation.

Creating spies with `mock.fn()`

Let's revisit our earlier test that verified whether all tasks were executed and the 'empty' event was emitted. Here's an extract of the original version:

```
test('All tasks are executed and empty is emitted', async () => {
  const queue = new TaskQueue(2)
  const task1status = Promise.withResolvers()
  let task1Completed = false
  const task2status = Promise.withResolvers()
  let task2Completed = false
  const task1 = async () => {
    await setImmediate()
    task1Completed = true
    task1status.resolve()
  }
  const task2 = async () => {
    // ...
  }

  queue.pushTask(task1).pushTask(task2)
  await Promise.allSettled([task1status.promise, task2status.promise])

  assert.ok(task1Completed, 'Task 1 completed')
  assert.ok(task2Completed, 'Task 2 completed')
  await once(queue, 'empty')
})
```

This test works fine, but it requires us to manually track the state of each task (`task1Completed` and `task2Completed`) and use resolver promises just to detect whether a task has run.

Let's now rewrite this using Node.js's built-in mocking functionality to create spies (which we previously described as stubs that record how they were used):

```
import { suite, test, mock } from 'node:test'
// ...
test('All tasks are executed and empty is emitted (v2)', async () => {
  const queue = new TaskQueue(2)

  const task1 = mock.fn(async () => {
    await setImmediate()
  })
  const task2 = mock.fn(async () => {
    await setImmediate()
  })

  queue.pushTask(task1).pushTask(task2)
  await once(queue, 'empty')

  assert.equal(task1.mock.callCount(), 1)
  assert.equal(task2.mock.callCount(), 1)
})
```

With this version, we no longer need flags or resolver promises. Instead, we use `mock.fn()` (with `mock` imported from the built-in `node:test` module) to wrap each task in a spy that records every call made to it. After the queue finishes processing, we simply check how many times each task was called using the `mock.callCount()` function.

This not only makes the test easier to read and less verbose but also gives us more flexibility. For example, we could later inspect the arguments the task was called with or how long it took to execute, all without changing the actual implementation of the task.

When you call `mock.fn()`, Node creates a spy function: a special wrapper that behaves just like your original function but also tracks how it's used. This includes information such as the following:

- How many times it was called (`mock.callsCount()`)
- With what arguments it was called (`mock.calls[i].arguments`)
- What values it returned or what errors it threw (`mock.calls[i].result` and `mock.calls[i].error`)

By default, `mock.fn()` wraps the original function, preserving its behavior while adding these tracking capabilities. That means your logic still runs as usual, just with some handy observability layered on top.

However, you can also use `mock.fn()` to define a new function entirely, which is what we did in our example. In that case, you're not spying on an existing function, but instead using the mock itself as the implementation. This is especially useful when you're creating tasks or dependencies specifically for testing and want full control over their behavior.

This flexibility (wrapping real functions or defining test-specific ones) is one of the reasons `mock.fn()` is such a useful tool when writing unit tests.

Mocking HTTP requests with the built-in test mock

Mocks and spies are particularly powerful when you're testing code that interacts with external dependencies such as filesystems, network calls, or databases. In those cases, you often don't want to perform real I/O operations in your unit tests. Instead, you mock the dependency so that you can control its behavior and focus the test entirely on your SUT.

To see how this works in practice, let's start with an example (extracted and simplified from our web spider implementation): a function called `getInternalLinks()`. It takes a `pageUrl` and returns a set of internal links found in that page's HTML. It fetches the page, parses the HTML content, and extracts anchor (`<a>`) tags whose `href` values point to other pages on the same host (but with a different path). These links are collected and returned as a Set of absolute URLs. Let's have a look at the code:

```
// getPageLinks.js
import { Parser } from 'htmlparser2' // v9.1.0

export async function getInternalLinks(pageUrl) {
  const url = new URL(pageUrl) // 1
  const response = await fetch(pageUrl)
  if (!response.ok) {
    throw new Error(`Failed to fetch ${pageUrl}: ${response.statusText}`)
  }

  const contentType = response.headers.get('content-type') // 2
  if (contentType === null || !contentType.includes('text/html')) {
    throw new Error('The current URL is not a HTML page')
  }

  const body = await response.text() // 3
  const internalLinks = new Set()
  const parser = new Parser({
    onopentag(name, attribs) {
      if (name === 'a' && attribs.href) {
        const newUrl = new URL(attribs.href, url)
        if (
          newUrl.hostname === url.hostname &&
```

```

        newUrl.pathname !== url.pathname
      ) {
        internalLinks.add(newUrl.toString())
      }
    },
  })
  parser.end(body)
  return internalLinks // 4
}

```

Let's review some important details of this implementation:

1. We start by converting `pageUrl` into a URL object, and we use `fetch()` to download the content. If the response is not successful (`response.ok` is `false`), the code throws an error.
2. Before parsing the content, we check whether the response body is HTML by inspecting the `content-type` header. If it's not an HTML page, the code throws an error (this prevents trying to parse things such as images or JSON).
3. The HTML body is read as text and passed into an `htmlparser2` parser. The parser is configured to look for `<a>` tags with an `href` attribute. For each `<a href="...">`, we resolve the link relative to the original page URL (so relative paths work correctly), and check whether the link is on the same hostname and that it points to a different path than the current page. If both conditions are met, the link is considered internal and added to the `internalLinks` set.
4. Finally, the function returns a set of internal URLs (as strings). The reason why we are using a set to collect links is that sets ensure uniqueness, so the same internal link won't be added more than once, even if it appears multiple times in the HTML.

Now, how do we test this function? Here's a first functional, but naïve, approach:

```

// getPageLinks.test.js
import { suite, test } from 'node:test'
import assert from 'node:assert/strict'
import { getInternalLinks } from './getPageLinks.js'

suite('getPageLinks', { concurrency: true, timeout: 500 }, () => {
  test('It fetches all the internal links from a page', async () => {
    const links = await getInternalLinks('https://loige.co')
    assert.deepEqual(
      links,
      new Set([
        'https://loige.co/blog',
        'https://loige.co/speaking',
        'https://loige.co/about',

```

```

    })
  )
})
})

```

This test checks that the `getInternalLinks()` function correctly fetches and returns all internal links from the home page of a real website; in this case, it's Luciano's website: `loige.co`. It calls the function with the URL, waits for the result, and then asserts that the returned set of links matches the expected ones.

While this test might work now, it comes with a few serious downsides:

- **It depends on a real, live website:** If the content of `loige.co` changes (say, if a new link is added or a section is renamed), this test will fail, even though there's nothing wrong with our code. That makes it brittle and prone to false negatives.
- **The test is not predictable:** Since the page might return different content at different times (or even fail to load due to network issues), you can't rely on it to always behave the same way. Tests should ideally be deterministic: they should give the same result every time they run.
- **It's hard to test error scenarios:** For example, how would you test the case where the server returns a non-HTML response or a failed status code? You'd have to find or create real URLs that return those exact cases, which is impractical and difficult to maintain.

Instead of making real HTTP requests, we can mock the `fetch()` call to simulate specific scenarios. That way, we can provide a static HTML response that we control and verify that the function behaves correctly. This makes the test more stable, faster to execute, more focused, and easier to extend. Let's see how we can do that:

```

test('It fetches all the internal links from a page', async t => {
  const mockHtml = `
    <html>
      <body>
        <a href="https://loige.co/blog">Blog</a>
        <a href="/speaking">Speaking</a>
        <a href="/about">About</a>
        <a href="https://www.linkedin.com/in/lucianomammino/">My LinkedIn
          profile</a>
        <a href="/about">About</a>
      </body>
    </html>
  `

  t.mock.method(global, 'fetch', async _url => ({
    ok: true,
    status: 200,

```

```

    headers: {
      get: key =>
        key === 'content-type' ? 'text/html; charset=utf-8' : null,
    },
    text: async () => mockHtml,
  }))

  const links = await getInternalLinks('https://loige.co')

  assert.deepEqual(
    links,
    new Set([
      'https://loige.co/blog',
      'https://loige.co/speaking',
      'https://loige.co/about',
    ])
  )
})

```

- In this test, we're checking that `getInternalLinks()` correctly extracts internal links from an HTML page, but instead of making a real HTTP request, we simulate the response using Node's built-in mocking system. We define a small HTML snippet (`mockHtml`) that includes several links. Some are internal to the domain `loige.co` (both absolute and relative). One is an external link (to a LinkedIn page). One internal link appears twice (to test deduplication).
- We then use `t.mock.method()` to mock the global `fetch()` function, so that any call to `fetch()` within this test returns our controlled response. The mocked response has a successful status (200) and a proper content type, and returns our HTML string when `.text()` is called.
- After calling `getInternalLinks('https://loige.co')`, we assert that the function returns only the expected internal links and that they are deduplicated correctly.



By using `t.mock.method()` instead of the top-level `mock.method()`, we scope the mock to this specific test. This means the mock is automatically cleaned up once the test completes, so we don't need to manually restore the original `fetch()` after the test completes. This helps keep our tests isolated and avoids potential side effects between tests, which is especially important when multiple tests might rely on the same global function.

Mocking HTTP requests with Undici

There are other ways to mock HTTP requests. For example, **Undici**'s `MockAgent` is a recommended and robust way to mock HTTP requests when using `fetch`, especially since Undici (`nodejsdp.link/11`) is the underlying `fetch` implementation in Node.js. It's shipped with Node, but not currently exposed by Node itself, so it must be installed as an external dependency to be used in tests.

Here's how to rewrite our `getInternalLinks()` test using Undici's `MockAgent` and `setGlobalDispatcher`. This version follows Node.js documentation best practices and sets up a mock HTTP response without touching global `fetch` directly:

```
import { MockAgent, setGlobalDispatcher } from 'undici' // v7.6.0 // 1
// ...
test('It fetches all the internal links from a page (with undici)', async _t =>
{
  const agent = new MockAgent()
  agent.disableNetConnect() // 2
  setGlobalDispatcher(agent)

  const mockHtml = `...` // elided for brevity
  agent // 3
    .get('https://loige.co')
    .intercept({
      path: '/',
      method: 'GET',
    })
    .reply(200, mockHtml, {
      headers: {
        'content-type': 'text/html; charset=utf-8',
      },
    })

  const links = await getInternalLinks('https://loige.co')

  assert.deepEqual(
    links,
    new Set([
      'https://loige.co/blog',
      'https://loige.co/speaking',
      'https://loige.co/about',
    ])
  )
})
```

Here's a list of the most important changes:

1. In this implementation, we import `MockAgent` and `setGlobalDispatcher` from `Undici`. These are used to set up a fully mocked HTTP environment, replacing the default dispatcher used by Node's native `fetch()`. In Undici's lingo, an agent is the internal component that allows dispatching requests against multiple different origins. By mocking it, we are effectively taking control of how requests are made, and we can use this mechanism to avoid making real HTTP requests and simulate different responses.
2. `agent.disableNetConnect()` prevents unmocked requests from accidentally hitting the real network. This ensures test isolation.
3. We need to configure the agent for our specific test case. Here, we call `.get('https://loige.co')` and `.intercept({ path: '/', method: 'GET' })` to configure the mock agent to intercept GET requests to `/` on the specific website and reply with our controlled HTML content and headers.

Compared to the previous solution, this approach allows for more fine-grained control. For example, you can configure specific responses for different URLs and methods, something that can be very convenient in more realistic scenarios where you might have different requests you need to mock in a single test case.

The only small drawback is that creating an agent instance can be a bit of a verbose repetition if you are writing multiple tests in the same suite. In these cases, we can create a global agent at the suite level and use the test runner's built-in `beforeEach()` and `afterEach()` hooks to control initialization (and de-initialization) of the agent at the suite level. Here's a simplified example of how to do that:

```
import { afterEach, beforeEach, suite, test } from 'node:test'
import { MockAgent, getGlobalDispatcher, setGlobalDispatcher } from 'undici'

suite('example with undici and beforeEach + afterEach', () => {
  let agent
  const originalGlobalDispatcher = getGlobalDispatcher()

  beforeEach(() => {
    agent = new MockAgent()
    agent.disableNetConnect()
    setGlobalDispatcher(agent)
  })

  afterEach(() => {
    setGlobalDispatcher(originalGlobalDispatcher)
  })

  test('a test...', () => {
    /* ... */
  })
})
```

```
test('another test...', () => {
  /* ... */
})
})
```

In this example, `beforeEach()` allows us to execute code before every test runs. We use this convenient hook to create a new `MockAgent` instance, disable real network access with `.disableNetConnect()`, and replace the global dispatcher with our mock using `setGlobalDispatcher(agent)`. This ensures that each test gets a fresh, isolated mock environment and that no external HTTP calls are accidentally made. More importantly, we don't need to copy-paste this setup code inside every single test; we are just declaring it once for the entire suite. Similarly, `afterEach()` is a hook that runs after every test. We use this hook to restore the original global dispatcher and make sure that our mocking is reset at the end of every test (it's generally a good practice to restore the environment at the end of every test).



If you're not using the built-in fetch API (for example, if you're working with HTTP clients such as `axios` (nodejsdp.link/axios) or using Node's low-level `node:http` or `node:https` modules), you won't benefit from mocking the global Undici dispatcher. In those cases, you'll need to mock the specific modules your code relies on. A popular library for this is `nock` (nodejsdp.link/nock), which makes it easy to intercept HTTP requests, define custom responses, and even assert how many times an endpoint was hit or what headers were sent. It's especially handy when you need to simulate multiple endpoints or chain different behaviors in the same test.

Mocking Node.js core modules

When writing unit tests, it's common to mock dependencies to isolate the behavior of the module under test. This becomes especially important when dealing with built-in modules such as `node:fs` or `node:fs/promises`, where we want to avoid actual filesystem interactions and instead simulate various conditions (e.g., a missing directory, a successful write, or a permission error).

Let's walk through a relatively simple example using the `saveConfig()` function:

```
// saveConfig.js
import { access, mkdir, writeFile } from 'node:fs/promises'
import { dirname } from 'node:path'

export async function saveConfig(path, config) {
  const folder = dirname(path)

  try {
    await access(folder)
  } catch {
    await mkdir(folder, { recursive: true })
  }
}
```

```
const json = JSON.stringify(config, null, 2)
await writeFile(path, json, 'utf-8')
}
```

This function takes a file path and a config object, ensures that the directory exists (creating it if needed), and writes the config to disk in JSON format.

Now, how do we test this function? Ideally, we don't want to interact with the real filesystem. That would make testing much harder because we would need to ensure that the filesystem is in the correct state (e.g., the given directory exists or not) before running our tests. Moreover, our tests might end up conflicting with other tests or applications relying on the same filesystem. A better approach is to use Node's (currently experimental) module mocking API to replace the `node:fs/promises` module with a mock implementation. Here's how we do that:

```
// saveConfig.test.js
import assert from 'node:assert/strict'
import { mock, suite, test } from 'node:test'
import { setImmediate } from 'node:timers/promises'

suite('saveConfig', { concurrency: true, timeout: 500 }, () => {
  test('Creates folder (if needed)', async t => {
    const mockMkdir = mock.fn()
    const mockAccess = mock.fn(async _path => {
      await setImmediate()
      throw new Error('ENOENT')
    })
    t.mock.module('node:fs/promises', {
      cache: false,
      namedExports: {
        access: mockAccess,
        mkdir: mockMkdir,
        writeFile: mock.fn(),
      },
    })

    const { saveConfig } = await import('./saveConfig.js')
    await saveConfig('./path/to/configs/app.json', { port: 3000 })

    assert.equal(mockMkdir.mock.callCount(), 1)
  })
})
```

This test verifies the behavior of the `saveConfig()` function when the target directory doesn't exist and needs to be created. The test uses Node.js's built-in mocking system (`t.mock.module`) to replace the `node:fs/promises` module with a custom mock. Specifically, note the following:

- The `access` function is mocked to simulate a missing directory by throwing an `ENOENT` error
- The `mkdir` function is mocked to track whether it gets called
- `writeFile` is also mocked, but not asserted on in this test, since we are only focusing on testing whether the folder gets created or not for this particular test

One very important detail is that we import the module dynamically (using `await import('./saveConfig.js')`) after the mock is in place. This is a requirement for the mock to take effect. This is necessary because any module that imports `node:fs/promises` before the mock is applied will keep a reference to the real module. In other words, mocking only works if the module is loaded after the mock is registered. That's why a dynamic import (`await import`) is used instead of a regular top-level import (`import ... from`), which would execute too early.

We can now run this test with the following:

```
node --test --experimental-test-module-mocks
```



At the time of writing, mocking modules is an experimental feature, so `--experimental-test-module-mocks` is required. This might not be the case anymore as you are reading this. So, make sure to check out the relevant documentation ([nodejsdp.link/mock-modules](https://nodejs.org/docs/latest/api/test.html#experimental-test-module-mocks)) to see whether this flag is still required or whether the functionality has since evolved significantly.

After you run the preceding command, you should see a happy, green test. Great!

Now, let's add a second test to make sure that, when the directory exists already, our code doesn't try to re-create the given directory:

```
test('Does not create folder (if exists)', async t => {
  const mockMkdir = mock.fn()
  const mockAccess = mock.fn(async _path => {
    await setImmediate()
  })
  t.mock.module('node:fs/promises', {
    cache: false,
    namedExports: {
      access: mockAccess,
      mkdir: mockMkdir,
      writeFile: mock.fn(),
    },
  })
})
```

```
const { saveConfig } = await import('./saveConfig.js')
await saveConfig('./path/to/configs/app.json', { port: 3000 })

assert.equal(mockMkdir.mock.callCount(), 0)
})
```

The only differences between this test and the previous one are that now our access mock does not throw an exception (which simulates that the given directory exists), and we now assert that the `mkdir` mock was not called.

Now, if we execute the tests again, we will see a scary error that looks like this:

```
✗ Does not create folder (if exists) (0.814ms)
  Error [ERR_INVALID_STATE]: Invalid state: Cannot mock 'node:fs/promises.' The
  module is already mocked.
```

The problem here is that we can't really mix module mocking and concurrent tests safely, at least not if the tests are mocking the same module. Module mocks are effectively global state. One solution to this problem is to disable concurrency by setting `concurrency` to `false` in the suite options, but we will explore some alternative approaches later in this chapter.

Mocking other dependencies

So far, we've looked at how to mock Node.js built-in modules such as `node:fs/promises`. But in many real-world applications, your code doesn't just rely on the platform; it also depends on your own internal modules or third-party ones. These might include things such as database clients, logging utilities, payment gateways, and so on. To keep our unit tests isolated and focused, we want to avoid actually invoking those dependencies. Instead, we can mock them and simulate their behavior to control test conditions and assert how they're used. Let's look at a realistic example involving a database client.

Let's start with a file called `dbClient.js`. This module exports a simple `DbClient` class with a `query` method that would (in a real app) connect to a database and execute a query. In this example, it just throws an exception, and since we will have to mock it, this is good enough for demonstrating how to write unit tests in similar scenarios:

```
// dbClient.js
export class DbClient {
  async query(_sql, _params) {
    // In real life, this would talk to a database
    throw new Error('Not implemented')
  }
}
```

Next, we are going to assume we are working on a project that allows customers to pay for products or services using vouchers. In our business logic, we want to determine whether a user can pay for something using their available vouchers.

To do this, we query the database for all non-expired vouchers with a positive balance, sum them up, and compare that to the required amount:

```
// payments.js
import { DbClient } from './dbClient.js'

const db = new DbClient()

export async function canPayWithVouchers(userId, amount) {
  const vouchers = await db.query(
    `SELECT * FROM vouchers
     WHERE user_id = ? AND
     balance > 0 AND
     expiresAt > NOW()`,
    [userId]
  )

  const availableBalance = vouchers
    .reduce((acc, v) => acc + v.balance, 0)

  return availableBalance >= amount
}
```



Note that this example is intentionally simple. We're focusing on specific testing techniques, so the logic is a bit contrived. We're fetching all voucher records just to sum their balances, something that could be done more efficiently with a single `SUM()` SQL query. In scenarios like this, keeping all the data processing at the database level is generally the more practical choice. That said, in a more realistic application, there are plenty of valid reasons why we might still want to retrieve all the records and handle some of the logic in the application layer. For instance, we might need to prioritize which vouchers to use based on expiration or balance, enforce custom business rules that are difficult to express in SQL, simulate how a payment would be split across multiple vouchers, or log and audit which vouchers were evaluated. These are all situations where retaining full control over the data in the application makes sense. So, while in our current example we're just using a simple `reduce()` to sum the balances, imagine a more complex scenario where additional logic is required. The testing techniques we're exploring here will continue to apply, even as the logic grows more sophisticated.

Here's how we test the `canPayWithVouchers()` function by mocking the `DbClient` class defined in `./dbClient.js`. Instead of hitting a real database, we return predefined mock records that simulate real vouchers coming from a database:

```
// payments.test.js
import assert from 'node:assert/strict'
import { after, beforeEach, mock, suite, test } from 'node:test'
import { setImmediate } from 'node:timers/promises'

const sampleRecords = [ // 1
  {
    id: 1,
    userId: 'user1',
    balance: 10,
    expiresAt: new Date(Date.now() + 1000),
  },
  {
    id: 2,
    userId: 'user1',
    balance: 5,
    expiresAt: new Date(Date.now() + 1000),
  },
  {
    id: 3,
    userId: 'user1',
    balance: 3,
    expiresAt: new Date(Date.now() + 1000),
  },
]

const queryMock = mock.fn(async (_sql, _params) => { // 2
  await setImmediate()
  return sampleRecords
})

mock.module('./dbClient.js', { // 3
  cache: false,
  namedExports: {
    DbClient: class DbMock {
      query = queryMock
    },
  },
})
```



```

}))

const { canPayWithVouchers } = await import('./payments.js') // 4

suite('canPayWithVouchers', { concurrency: false, timeout: 500 }, () => {
  beforeEach(() => { // 5
    queryMock.mock.resetCalls()
  })

  after(() => {
    queryMock.mock.restore()
  })

  test('Returns true if balance is enough', async () => { // 6
    const result = await canPayWithVouchers('user1', 18)

    assert.equal(result, true)
    assert.equal(queryMock.mock.callCount(), 1)
  })

  test('Returns false if balance is not enough', async () => {
    const result = await canPayWithVouchers('user1', 19)

    assert.equal(result, false)
    assert.equal(queryMock.mock.callCount(), 1)
  })
})

```

Let's discuss, one by one, the main takeaways of this test:

1. We define `sampleRecords`, an array of records that simulate data that might be coming from a real database connection.
2. We define `queryMock`, a mock function that simulates making a query to the database and returning the set of sample records.
3. We use `mock.module()` to replace the entire `DbClient` class before we import the module under test. This is done outside our suite because, as we saw in the previous section, mocking modules is effectively changing global state, so this mock is going to be available for the execution of the entire test. Note how this mock replaces the original query method with our `queryMock`. This is the place where the magic happens, since we are effectively swapping the behavior of the dependency module with a custom mocked behavior.
4. We use a dynamic import (`await import('./payments.js')`) to ensure that the mock is applied before the module under test is evaluated.

5. Inside our suite, we use two hooks. The first one, `beforeEach()`, runs before every test is executed, and it makes sure the call count for our `queryMock` is reset. This makes it easy to make assertions and check whether the mock was called in each individual test. The second hook, `after()`, is executed when the suite is complete, and it makes sure to restore the mocked class. This is a good practice that keeps the execution environment clean after the test suite is completed, in case there are more test files to be executed in the same process.
6. Finally, we have two separate tests. The first one verifies that when we provide an amount **less than or equal to** the user's total voucher balance, the `canPayWithVouchers()` function returns `true`. The second test provides an amount **greater than** the available voucher balance, and in that case, we expect `canPayWithVouchers()` to return `false`.



In this suite, we once again set concurrency to `false` to ensure consistency when mocking modules. Node.js currently doesn't support mocking the same module multiple times concurrently in a single suite. Without this setting, you may encounter errors such as `ERR_INVALID_STATE` or see inconsistent behavior due to cached mocks being reused across tests. Disabling concurrency guarantees that each test gets a clean, isolated mock environment.

This example shows a practical way to mock a dependency module in our tests. In our case, we mocked an internal module (the database client), but the same approach would work just as well for external dependencies, such as a third-party database client installed from npm. The key idea behind mocking is to isolate the unit of code we want to test. Here, our focus was on the logic inside `canPayWithVouchers()`, not on how database queries are executed. By mocking the database client, we were able to simulate different conditions and fully control the environment around our unit, leading to fast, focused, and reliable tests.

Problems with mocking imports

While mocking imports is a powerful and widely used technique in unit testing, it does come with some important limitations and trade-offs to keep in mind.

One major drawback of mocking imports is that it introduces **tight coupling** between our implementation details and our tests. When we mock a module or a class, we are making strong assumptions about how that dependency is used internally. Even if a dependency changes in a way that's completely backward compatible (e.g., it keeps the same API), we may still need to update our mocks or test logic to reflect those changes. This can make refactoring more painful and cause tests to fail for reasons unrelated to actual bugs.

With tight coupling between code and tests, as our codebase grows, making changes (such as adding features, fixing bugs, or refactoring) becomes more painful, because there's a higher chance that even a small change will require updates to a large number of tests. This is not a desirable characteristic of a healthy codebase, as it slows down future development and discourages iteration. Ideally, we want a codebase that is easy to evolve, with tests that provide confidence in the correctness of our logic, not tests that break just because we changed an import or reorganized a module.

As we've seen, mocking module imports (especially with Node's built-in `mock.module()`) is essentially a global operation. Once we mock a module, that mock affects how the module behaves everywhere it's imported within the test process. This global nature introduces some challenges that we need to be mindful of. For starters, we have to be very careful about when we apply a mock. If we import the module under test before the mock is in place, the mock simply won't work because the real dependency will already be cached and used. That's why we've consistently used dynamic imports in our tests: to make sure our mocks are active before the module is loaded.

Another issue is that we can't mock the same module in multiple tests at the same time. If we try to do that, Node.js might throw an error or our tests might behave inconsistently (depending on the order of operations). This is why, in several of our examples, we've had to explicitly disable concurrency. While this avoids the problem, it also means we lose one of the nice performance benefits of running tests concurrently.

All of this adds some extra complexity to our test setup and increases the risk of hidden coupling or side effects between tests. Because mocking and restoring modules isn't free in terms of performance, our test suite may end up running a bit slower, especially as the number of mocks grows.

Mocking can be a powerful tool, but these limitations remind us that it's not always the simplest or cleanest solution, in particular with large codebases with tons of tests.

Mocking imports versus dependency injection

As we've seen in the previous section, mocking module imports can be effective, but it also comes with some significant limitations: tight coupling, global scope, complex setup, and constraints on test concurrency. Fortunately, there's a powerful and elegant pattern that helps us avoid many of these issues: **dependency injection**, or **DI**.

We've already introduced the DI design pattern back in *Chapter 7, Creational Design Patterns*. If you need a quick refresher, here's a condensed summary of how we defined it:



DI is a simple but powerful pattern in which the dependencies of a component are provided as inputs by an external entity, often referred to as the injector. This injector can be anything from a simple initialization script to a fully featured container that maps and wires dependencies across the application. The main benefit of this approach is improved decoupling, especially for modules that rely on stateful or replaceable components (such as database clients or loggers). Instead of creating or importing their dependencies internally, modules receive them from the outside, which makes them easier to reuse, configure, and test.

By refactoring our source and applying this pattern, we can eliminate the need to mock modules at the import level. Instead, we just pass mock dependencies directly to the function or class under test. This makes our code cleaner, our tests simpler, and our dependencies easier to manage.

Let's take the function we've been working with (`canPayWithVouchers()`) and see how we can refactor it to use DI. Then, we'll revisit its test and observe how much simpler and more robust the new setup becomes:

```
// payments.js
export async function canPayWithVouchers(db, userId, amount) {
  const vouchers = await db.query(
    `SELECT * FROM vouchers
     WHERE user_id = ? AND
     balance > 0 AND
     expiresAt > NOW()` ,
    [userId]
  )

  const availableBalance = vouchers.reduce((acc, v) => acc + v.balance, 0)

  return availableBalance >= amount
}
```

In this new version of `canPayWithVouchers()`, we've refactored the function to accept the `db` client as a parameter instead of creating it internally or importing it from another module. This is the essence of DI: rather than tightly coupling our logic to a specific implementation of a dependency, we make that dependency configurable from the outside.

This simple change has a big impact. It makes the function more flexible and reusable. We can now use it with any object that exposes a compatible query method. It also removes the need to mock module imports in our tests. Instead, we can pass a plain mock object directly to the function, keeping our tests cleaner, faster, and easier to reason about. Let's see this last point in action by rewriting our tests:

```
// payments.test.js
import assert from 'node:assert/strict'
import { suite, test } from 'node:test'
import { setImmediate } from 'node:timers/promises'
import { canPayWithVouchers } from './payments.js' // 1

const sampleRecords = [ /* ... elided for brevity */ ]

suite('canPayWithVouchers', { concurrency: true, timeout: 500 }, () => { // 2
  test('Returns true if balance is enough', async t => {
    const dbMock = { // 3
      query: t.mock.fn(async (_sql, _params) => {
        await setImmediate()
        return sampleRecords
      }),
    },
```

```
}

const result = await canPayWithVouchers(dbMock, 'user1', 18)

assert.equal(result, true)
assert.equal(dbMock.query.mock.callCount(), 1)
})

test('Returns false if balance is not enough', async t => {
  const dbMock = { // 3
    query: t.mock.fn(async (_sql, _params) => {
      await setImmediate()
      return sampleRecords
    }),
  }

  const result = await canPayWithVouchers(dbMock, 'user1', 19)

  assert.equal(result, false)
  assert.equal(dbMock.query.mock.callCount(), 1)
})
})
```

With the new implementation of `canPayWithVouchers()` using DI, our test setup becomes much simpler and more maintainable. Let's highlight some of the most important changes that this new version of the test suite brings:

1. First, we no longer need to mock module imports or worry about when we apply those mocks. This means we can safely import the module under test at the top of the file, just like any other regular module; no more dynamic imports or special ordering concerns.
2. Since we're no longer mocking global modules, we can also re-enable concurrency at the suite level. Each test is now fully isolated and safe to run in parallel with others, which improves the performance of the test suite.
3. We also benefit from a cleaner mocking strategy: in each test, we create a local mock database client using `t.mock.fn()`. This mock is scoped to the individual test, so it's automatically cleaned up when the test finishes. There's no need for `beforeEach` or `after` hooks to reset global mocks; each test starts with a clean slate by design.

Altogether, this version of the test is easier to read, faster to run, and less fragile, all thanks to the simplicity and flexibility that come with DI.

As we've seen, using DI can greatly simplify our tests, reduce coupling, and make our code more modular and flexible. By injecting dependencies rather than hardcoding them, we gain more control over how our code behaves in different contexts, making it easier to test, reuse, and evolve over time. It's a powerful pattern that, when used thoughtfully, can lead to cleaner designs and more maintainable test suites. We strongly encourage you to adopt this approach wherever it makes sense in your own codebase.

At this point, we have a solid understanding of unit testing and how to isolate and verify the behavior of individual components. But unit tests alone won't give us full confidence in the system as a whole. Just because each part works fine in isolation doesn't mean that all the pieces will work well together once integrated. So, how do we test that? That's exactly what we'll explore next.

Writing integration tests

In our previous example, we executed a SQL query through a database client, and, in our unit tests, we replaced that client with a mock. This allowed us to isolate the business logic and test it independently. But it also raises two important questions:

- How do we know that the SQL query itself is correct and actually returns what we expect?
- How can we be sure that our mocked data is a faithful representation of what the real database would return in a real scenario?

This is where integration tests come into play.

Integration tests allow us to move beyond isolated units of code and verify how different components (such as our business logic and our database) work together in practice. They help us test the real flow, using real inputs, real queries, and real infrastructure (like an actual SQLite database), giving us confidence that everything behaves correctly as a system.

In this section, we'll explore how to write effective integration tests, how they complement unit tests, and how to structure them to stay fast, reliable, and easy to maintain. Let's dive in.

Testing with a local database

Let's return to our previous example: `canPayWithVouchers()`. We've already written unit tests for this function, and we've confirmed that its logic behaves correctly when provided with mock data. But there's one critical part we haven't tested yet: the database integration itself.

Right now, we're trusting that our SQL query is correct and that our mocked data accurately reflects what the database would return. But in practice, we haven't tested either of those assumptions! What if our query has a subtle bug? What if the schema isn't what we expected? We can write an integration test here to give us confidence that this part of the application works as expected and that we won't run into unpleasant surprises when our code hits production.

Before we can do that, we need to choose a SQL database to use during our test. While any SQL database would work here, we'll use SQLite to keep things simple. If you've read *Chapter 7, Creational Design Patterns*, and *Chapter 9, Behavioral Design Patterns*, you've already seen SQLite in action, so this should feel familiar! SQLite is a fast, lightweight, embeddable database engine. It runs entirely in-process, which means we don't need to spin up a full server or manage connections to an external system. We can execute real SQL queries and store data locally using nothing more than a library.

In most production applications, you'd likely use something such as PostgreSQL or MySQL, but those require more setup. They're great for real deployments, but a bit too heavy for quick integration tests unless you introduce tools such as Docker to help manage and isolate those services. (We'll cover that approach in *Chapter 12, Scalability and Architectural Patterns*.)

For now, SQLite gives us the perfect mix of realistic behavior and low overhead. So, let's begin by implementing our `DbClient` class to support SQLite. Previously, this class was just a stub with a `query()` method that threw a **Not Implemented** error. Now, we'll make it fully functional and connect it to a real, in-memory database we can use in our tests:

```
// dbClient.js
import sqlite3 from 'sqlite3' // v5.1.7
import { open } from 'sqlite' // v5.1.1

export class DbClient {
  #dbPath
  #db

  constructor(dbPath) {
    this.#dbPath = dbPath
    this.#db = null
  }

  async #connect() {
    if (this.#db) {
      return this.#db
    }

    this.#db = await open({
      filename: this.#dbPath,
      driver: sqlite3.Database,
    })

    return this.#db
  }
}
```

```
async query(sql, params = {}) {
  const db = await this.#connect()
  return db.all(sql, params)
}

async close() {
  if (this.#db) {
    await this.#db.close()
    this.#db = null
  }
}
}
```

This code implements a functional `DbClient` class that provides a lightweight wrapper around a SQLite database using the `sqlite` and `sqlite3` libraries. It encapsulates connection handling and exposes a simple API for executing SQL queries and managing the database lifecycle.

Here are some important implementation details:

- The constructor accepts a `dbPath`, which is the path to the SQLite database file. This can be a regular file path or `':memory:'` for an in-memory database (data not persisted when the process ends, which is perfect for testing).
- The `#connect()` method lazily opens the database connection the first time it's needed and caches it for future use. This means a new connection isn't established with every query, which improves efficiency. It also allows users of the class to call the `query()` method directly, without needing to worry about explicitly managing the connection themselves. This is a great example of **lazy initialization**: a pattern that defers the creation of potentially expensive resources (such as a database connection) until they are actually required. Beyond improving performance, it also simplifies the interface, making the class easier and more intuitive to use.
- The `query()` method connects to the database (if not already connected) and executes the provided SQL query with optional parameters, returning all matching rows.
- The `close()` method safely closes the database connection and resets the internal state.

Now that we've implemented our minimal database client, we need a way to set up the database schema before running queries or tests. To keep things organized and reusable, let's add a small utility function that initializes the necessary tables. This function will ensure that the `users` and `vouchers` tables exist, creating them if needed. It relies on our `DbClient` and uses standard SQL to define the schema in a way that works seamlessly with SQLite:

```
// dbSetup.js
export async function createTables(db) {
  await db.query(`
    CREATE TABLE IF NOT EXISTS users (
      id TEXT PRIMARY KEY,
```



```

        name TEXT NOT NULL
    )
` )

await db.query(`
    CREATE TABLE IF NOT EXISTS vouchers (
        id TEXT PRIMARY KEY,
        userId TEXT NOT NULL,
        balance REAL NOT NULL,
        expiresAt TIMESTAMP NOT NULL
    )
` )
}

```



Both `CREATE TABLE` statements use the `IF NOT EXISTS` clause, which guarantees that the tables are only created if they don't already exist. This makes the function safe to call multiple times, even in a database that already contains data, ensuring a reliable and idempotent setup process.

Finally, let's refactor the `payments.js` file to improve clarity and testability. We'll extract the logic for retrieving active vouchers into a separate function. This allows us to test the SQL query in isolation and makes the balance-checking logic more focused:

```

// payments.js
export async function getActiveVouchers(db, userId) {
    const vouchers = await db.query(
        `SELECT * FROM vouchers
        WHERE userId = ? AND
        balance > 0 AND
        expiresAt > strftime('%FT%T:%fZ', 'now')`,
        [userId]
    )

    return vouchers
}

export async function canPayWithVouchers(db, userId, amount) {
    const vouchers = await getActiveVouchers(db, userId)
    const availableBalance = vouchers.reduce((acc, v) => acc + v.balance, 0)

    return availableBalance >= amount
}

```

We now have two focused functions: one handles data retrieval (`getActiveVouchers()`), while the other (`canPayWithVouchers()`) contains the business logic for determining whether the available balance is sufficient. Note that we also slightly changed our SQL query to adapt it to the specific SQLite syntax (which doesn't support `NOW()` to get the current timestamp).

Now that our database setup is complete, we're finally ready to write our first integration test. We'll do this in a new file called `payments.int.test.js`. To keep our tests clean and focused, let's start by defining a couple of utility functions that will help us insert test data quickly and consistently:

```
function addTestUser(db, id, name) {
  return db.query(
    `INSERT INTO users (id, name)
      VALUES (?, ?)`,
    [id, name]
  )
}

async function addTestVoucher(db, id, userId, balance, expiresAt) {
  const record = {
    id,
    userId,
    balance,
    expiresAt: expiresAt ?? new Date(Date.now() + 1000).toISOString(),
  }
  await db.query(
    `INSERT INTO vouchers (id, userId, balance, expiresAt)
      VALUES (?, ?, ?, ?)`,
    [record.id, record.userId, record.balance, record.expiresAt]
  )
  return record
}
```

The `addTestUser()` function inserts a new user record into the `users` table, while `addTestVoucher()` inserts a new voucher record into the `vouchers` table. Both functions take a database client instance as their first argument, followed by the relevant data needed for the new record. The `addTestVoucher()` function also sets a default expiration time if one isn't provided, ensuring that the voucher isn't expired by default.

Now we're ready to write an integration test for the `getActiveVouchers()` function. This test verifies that our SQL query correctly filters vouchers based on the conditions defined in our business logic. Specifically, vouchers must belong to the correct user, must have a positive balance, and must not have expired:

```
suite('activeVouchers', { concurrency: true, timeout: 500 }, () => {
  test('queries for active vouchers', async () => {
    const expected = [] // 1
    const db = new DbClient(':memory:') // 2
    await createTables(db) // 3
    await addTestUser(db, 'user1', 'Test User 1') // 4
    await addTestUser(db, 'user2', 'Test User 2')
    expected.push(await addTestVoucher(db, 'voucher1', 'user1', 10))
    expected.push(await addTestVoucher(db, 'voucher2', 'user1', 5))
    expected.push(await addTestVoucher(db, 'voucher3', 'user1', 3))

    // 5
    // expired
    await addTestVoucher(
      db,
      'voucher4',
      'user1',
      10,
      new Date(Date.now() - 1000).toISOString()
    )
    // different user
    await addTestVoucher(db, 'voucher5', 'user2', 10)
    // zero balance
    await addTestVoucher(db, 'voucher6', 'user1', 0)

    const activeVouchers = await getActiveVouchers(db, 'user1') // 6

    db.close()
    assert.deepEqual(activeVouchers, expected) // 7
  })
})
```

Here's what we are doing in this test:

1. We initialize an empty array called `expected`. As we add valid voucher records to the database (in the next steps), we'll also push them into this array so we can use it to assert the function's output later.

2. We instantiate a new `DbClient`, using SQLite's `:memory:` option. This gives us a fresh, isolated database that lives only in memory. It is ideal for testing, as it requires no cleanup and won't persist between runs.
3. We call `createTables()` to create the `users` and `vouchers` tables in our temporary database. This ensures the database has the correct structure before inserting any data.
4. We add two users to the `users` table. The first one (`user1`) is the subject of our test, while `user2` is included to verify that vouchers for unrelated users are ignored.
5. We then add three valid vouchers for `user1` and track them in the `expected` array. These vouchers all have a positive balance and a valid expiration date, so we expect them to be returned by the function. We also add three more vouchers that should not be returned by `getActiveVouchers()`: one is expired, one belongs to a different user (`user2`), and one has a balance of zero.
6. We invoke `getActiveVouchers()` for `user1`. This will run the real SQL query and return all vouchers that meet the criteria: active (not expired), positive balance, and matching user.
7. Finally, we compare the result of the function against the `expected` array using `assert.deepEqual()`. This checks that the correct vouchers were returned, and only those vouchers, proving that our filtering logic behaves exactly as intended.

If we run the tests (`node --test`), we should see a green tick, which gives us confidence that our SQL query is correct. But is the `canPayWithVouchers()` function still working? To find that out, let's write a dedicated test suite for it in the same file (`payments.int.test.js`):

```
suite('canPayWithVouchers', { concurrency: true, timeout: 500 }, () => {
  test('Returns true if balance is enough', async () => {
    const db = new DbClient(':memory:')
    await createTables(db)
    await addTestUser(db, 'user1', 'Test User 1')
    await addTestVoucher(db, 'voucher1', 'user1', 10)
    await addTestVoucher(db, 'voucher2', 'user1', 5)
    await addTestVoucher(db, 'voucher3', 'user1', 3)

    const result = await canPayWithVouchers(db, 'user1', 18)

    db.close()
    assert.equal(result, true)
  })

  test('Returns false if balance is not enough', async () => {
    const db = new DbClient(':memory:')
    await createTables(db)
    await addTestUser(db, 'user1', 'Test User 1')
    await addTestVoucher(db, 'voucher1', 'user1', 10)
```

```
    await addTestVoucher(db, 'voucher2', 'user1', 5)
    await addTestVoucher(db, 'voucher3', 'user1', 3)

    const result = await canPayWithVouchers(db, 'user1', 19)

    db.close()
    assert.equal(result, false)
  })
})
```

The `canPayWithVouchers` integration test is structurally very similar to our earlier unit test but introduces a more complete and realistic environment by connecting to a real in-memory SQLite database instead of using a mocked one. This change allows us to validate not only the logic of the function itself but also its interaction with real data, a real schema, and an actual SQL query.

In the unit test version, we focused solely on verifying the logic of `canPayWithVouchers()` in isolation. To do this, we injected a fake database client where the `query()` method was mocked. It always returned the same hardcoded set of voucher records, regardless of the query or parameters. This approach was useful to confirm that the summing logic in the function behaved correctly under predictable conditions, and to ensure that we weren't relying on any specific SQL implementation. However, it also meant that we weren't verifying whether the SQL query was written correctly, or whether the function would behave the same way when real filtering was applied.

In contrast, this new integration test version initializes a fresh in-memory SQLite database using our real `DbClient` implementation. Before each test, we create the database schema using `createTables()`, insert a test user, and then add vouchers that simulate a realistic state of the system. We vary the total available balance by modifying the input vouchers, and then call the actual `canPayWithVouchers()` function.

We also no longer need to check how many times `query()` was called or with what parameters: the test now validates observable behavior against real inputs and expected outcomes, which is the core goal of integration testing.

In short, this new version goes beyond assumptions and verifies that the actual implementation works as intended when all the pieces are put together. It complements the unit tests by covering the “glue” that binds components, ensuring that the final system behaves reliably under real-world conditions.



In our previous integration tests, we were able to use `{ concurrency: true }` without any issues because we were relying on in-memory SQLite databases. Each test creates its own independent, isolated database instance, so running them concurrently is safe and efficient. However, when working with databases that require a dedicated server, such as MySQL or PostgreSQL, achieving the same level of isolation is more complex. In those cases, it's common to start a shared test database before the test suite runs and then run tests sequentially, resetting the database state between tests to ensure isolation. Alternatively, more advanced setups may involve spinning up ephemeral containers (e.g., using Docker) for each test worker, which provides better isolation but adds more overhead and setup complexity. Alternatively, you can explore tools such as `testcontainers` for Node.js (nodejsdp.link/testcontainers), which allows you to programmatically spin up Docker containers from your test scripts. This approach can provide full test isolation even with server-based databases, at the cost of slightly increased setup time and complexity.

Testing a web application

In the previous section, we focused on writing integration tests for a piece of software that interacted with a database. We validated that the application logic and the SQL queries behaved correctly together, using a real SQLite instance.

Now, we're going to take integration testing one step further by introducing an additional layer: a web API.

In this section, we'll build a small web application and then write full integration tests for it, covering the web layer, the database, and the business logic in one cohesive flow.

The application we're building is a very simplified event booking system. It will support two main operations:

- Creating a new event, each with a name and a maximum number of available seats
- Creating reservations for users to attend an event

Each user will be allowed to reserve only one seat per event, and only if there are still seats available. All data (events and reservations) will be stored in a SQLite database.

While the scope of this app is intentionally limited, there's already a fair amount of coordination between layers: database constraints, business rules, and a web-facing API. It's a perfect opportunity to see how to write meaningful and effective integration tests that validate how all parts of the system work together.

For the web framework, we'll use **Fastify** (nodejsdp.link/fastify). It's a modern, high-performance framework that's easy to use and provides excellent support for testing, including utilities for injecting HTTP requests directly into your app without needing to run a real server. That said, the testing patterns we'll use are broadly applicable and can be easily adapted to other frameworks, such as Express, Koa, or Hapi.

Setting up the project

Let's start by building the application itself, and then we'll dive into testing it.

For the database interactions, we will be reusing the same `DbClient` class from our previous example. We also need to create the tables that are required for this app. Let's do this in the `dbSetup.js` file:

```
export async function createTables(db) {
  await db.query(`
    CREATE TABLE IF NOT EXISTS events (
      id TEXT PRIMARY KEY,
      name TEXT NOT NULL,
      totalSeats INTEGER NOT NULL
    )
  `)

  await db.query(`
    CREATE TABLE IF NOT EXISTS reservations (
      id TEXT PRIMARY KEY,
      eventId TEXT NOT NULL,
      userId TEXT NOT NULL,
      UNIQUE(eventId, userId)
    )
  `)
}
```

The events table stores information about each event, including a unique ID, a name, and the total number of available seats.

The reservations table stores user reservations, linking each one to an event and a user. The combination of `eventId` and `userId` must be unique, which ensures that each user can only reserve one seat per event.

We will use this function during application startup or in the test setup, to ensure the database is ready before inserting or querying data.

Let's now define the main entry point of our web application:

```
// app.js
import Fastify from 'fastify'
import { bookEventRoute } from './routes/bookEvent.js'
import { createEventRoute } from './routes/createEvent.js'

export async function createApp(db) {
  const app = Fastify()
  app.decorate('db', db)
```

```
    await app.register(bookEventRoute)
    await app.register(createEventRoute)

    return app
}
```

This is where we see Fastify in action. The `createApp()` function is a simple factory that sets up and configures our web server. It takes a database instance as input and uses it to build a fully configured Fastify application.

Inside the function, we first create a new Fastify app instance by calling `Fastify()`. We then use `app.decorate()` to make the `db` object (our database client) available throughout the application, particularly inside route handlers. This is Fastify's built-in way of sharing values or services across the app, functioning a bit like lightweight DI or configuration management.

Next, we import and register our route modules (`bookEventRoute` and `createEventRoute`), which we'll look at shortly. By registering these routes here, we're wiring together all the components of the app in one place.

The function returns the fully configured Fastify instance, ready to be started or injected for testing. This modular setup is ideal for both production use and integration testing.

Before diving into the route definitions, let's take a quick look at how we start the web server for development or production:

```
// server.js
import { createApp } from './app.js'
import { DbClient } from './dbClient.js'
import { createTables } from './dbSetup.js'

const db = new DbClient('data/db.sqlite')
await createTables(db)

const app = await createApp(db)
app.listen({ port: 3000 })
```

This script is the file you run to start the web server in a real deployment or during local development.

Here's what's happening step by step:

- We start by creating an instance of our database client, pointing to a persistent SQLite file (`data/db.sqlite`).
- We then call `createTables(db)` to ensure the database schema is initialized. This function is safe to call even if the tables already exist, thanks to the use of `CREATE TABLE IF NOT EXISTS`.

- After the database is ready, we call `createApp(db)` to create and configure our Fastify application, injecting the database instance into it.
- Finally, we call `app.listen()` to start the web server on port 3000.

This modular structure gives us flexibility: we can start the app from this file in production, or we can reuse the `createApp()` function in our test suite to spin up the same app in-memory for integration testing.

The business logic

We now define the core business logic of our booking system. This logic lives in a dedicated module called `booking.js`. It contains the essential rules for creating events and handling reservations, ensuring that business constraints such as seat limits are properly enforced:

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

export async function reserveSeat(db, eventId, userId) {
  const [event] = await db.query('SELECT * FROM events WHERE id = ?',
    [eventId])
  if (!event) {
    throw new Error('Event not found')
  }

  const existing = await db.query(
    'SELECT COUNT(*) AS count FROM reservations WHERE eventId = ?',
    [eventId]
  )

  if (existing[0].count >= event.totalSeats) {
    throw new Error('Event is fully booked')
  }

  const reservationId = randomUUID()

  await db.query(
    'INSERT INTO reservations (id, eventId, userId) VALUES (?, ?, ?)',
    [reservationId, eventId, userId]
  )

  return reservationId
}

export async function createEvent(db, name, totalSeats) {
```

```

const eventId = randomUUID()
await db.query(
  'INSERT INTO events (id, name, totalSeats) VALUES (?, ?, ?)',
  [
    eventId,
    name,
    totalSeats,
  ]
)
return eventId
}

```

At a high level, this is what this code does:

- `createEvent()` inserts a new record into the events table. It uses `randomUUID()` to generate a unique ID for the event and stores the event name and total seat count.
- `reserveSeat()` handles the logic for booking a seat. It performs several checks:
 - It verifies that the event exists.
 - It counts how many reservations already exist for the event.
 - If the event is fully booked (i.e., reservation count $\geq$ total seats), it throws an error.
 - Otherwise, it creates a new reservation with a generated UUID.



We intentionally kept the seat reservation logic simple for the sake of this example, but be aware that this implementation isn't safe under high concurrency. When multiple users try to book at the same time, the seat count check might return `true` for several requests before any of them complete the actual reservation. If the event is nearly full, this can result in overbooking, a classic race condition. To make the logic more robust, you could wrap the check and insert in a database **transaction** and use **row-level locking** to ensure that no conflicting operations happen until the reservation is finalized.

This module encapsulates all the key rules and constraints that govern how the system behaves. By isolating this logic in its own file, we keep the route handlers clean and maintainable, while also making this code easier to test independently or reuse in other contexts.

Writing the routes code

We are finally ready to work on the first route: the one that allows clients to create a new event. This will be used to register events with a name and a fixed number of available seats. Here's the code:

```

// routes/createEvent.js
import { createEvent } from '../booking.js'

export function createEventRoute(fastify) {
  fastify.post('/events', { // 1

```

```

    schema: { // 2
      body: {
        type: 'object',
        required: ['name', 'totalSeats'],
        properties: {
          name: { type: 'string' },
          totalSeats: { type: 'integer' },
        },
      },
    },
  },
  async handler(request, reply) { // 3
    const { name, totalSeats } = request.body
    const eventId = await createEvent(fastify.db, name, totalSeats)
    return reply.status(201).send({ success: true, eventId })
  },
})
}

```

Here's what this route does:

1. It defines a POST endpoint at /events, allowing clients to create a new event.
2. It uses Fastify's built-in JSON schema validation to ensure the request body contains a name (string) and totalSeats (integer). If either is missing or of the wrong type, Fastify will automatically return a 400 Bad Request.
3. In the route handler, we extract the validated fields from the request body and call the createEvent() function (from our business logic module), passing the database and event details. If the creation succeeds, we return a 201 (Created) response along with the new event's ID.

This route is simple, but it already combines important layers: request validation, business logic delegation, and database interaction; all essential elements we'll be testing in our integration tests.

Now, let's look at the route that handles seat reservations for events. This endpoint allows a user to reserve a spot for a specific event (provided the event exists and still has seats available):

```

// routes/bookEvent.js
import { reserveSeat } from '../booking.js'

export function bookEventRoute(fastify) {
  fastify.post('/events/:eventId/reservations', { // 1
    schema: { // 2
      params: {
        type: 'object',
        required: ['eventId'],
        properties: {

```

```
      eventId: { type: 'string' },
    },
  },
  body: {
    type: 'object',
    required: ['eventId'],
    properties: {
      eventId: { type: 'string' },
    },
  },
},
async handler(request, reply) { // 3
  const { eventId } = request.params
  const { userId } = request.body

  try {
    const reservationId = await reserveSeat(fastify.db, eventId,
      userId)
    return reply.status(201).send({ success: true, reservationId })
  } catch (err) {
    if (err.message === 'Event not found') {
      return reply.status(404).send({ error: 'Event not found' })
    }

    if (err.message === 'Event is fully booked') {
      return reply.status(403).send({ error: 'Event is fully booked' })
    }

    fastify.log.error(err)
    return reply.status(500).send({ error: 'Server error' })
  }
},
})
}
```

Here's what this route does:

1. It defines a POST endpoint at `/events/:eventId/reservations`, where `eventId` is a dynamic path parameter.
2. It uses Fastify's schema system to validate both the route parameters (`eventId`) and the request body (`userId`). If either is missing or has the wrong type, Fastify will reject the request with a 400 Bad Request.

3. In the handler, it calls the `reserveSeat()` function, our core business logic, to attempt the reservation. If the event doesn't exist, we return a **404 Not Found**. If the event exists but has no more available seats, we return a **403 Forbidden**. Any unexpected errors are logged and result in a **500 Internal Server** error. If all goes well, it returns a **201** response that includes the ID of the new reservation.

This route completes the code we need for this example. Time to jump into testing it!

Integration testing

Now that we've built a simple but complete event booking system with a database layer, business logic, and a Fastify-based API, we're ready to write integration tests to validate that all the layers work together as expected.

The following tests simulate real HTTP requests to our application, exercising the full stack: routes, logic, and database behavior. We use an in-memory SQLite database to keep the tests fast, isolated, and clean. Here's the high-level structure of our integration test file (`booking.int.test.js`):

```
import { suite, test } from 'node:test'
import assert from 'node:assert/strict'
import { DbClient } from './dbClient.js'
import { createTables } from './dbSetup.js'
import { createApp } from './app.js'

suite('Booking integration tests', { concurrency: true }, () => {
  test.todo('Reserving a seat works until full', async () => {
    // TODO
  })

  test.todo('Returns 404 if event does not exist', async () => {
    // TODO
  })
})
```

In this initial skeleton, we import everything we need: the test runner, assertions, our database client and schema setup, and the app factory function. We also create a suite with two tests (currently marked as `TODO`).

Let's write the code for the first test, focused on testing that reserving a seat works until the event is full:

```
test('Reserving a seat works until full', async () => {
  const db = new DbClient(':memory:') // 1
  await createTables(db)
  const app = await createApp(db)

  const createEventResponse = await app.inject({ // 2
```

```
    method: 'POST',
    url: '/events',
    payload: { name: 'Event 1', totalSeats: 2 },
  })
  assert.equal(createEventResponse.statusCode, 201) // 3
  const eventData = createEventResponse.json()
  const reserveUrl = `/events/${eventData.eventId}/reservations`

  const res1 = await app.inject({ // 4
    method: 'POST',
    url: reserveUrl,
    payload: { userId: 'u1' },
  })
  assert.equal(res1.statusCode, 201)

  const res2 = await app.inject({
    method: 'POST',
    url: reserveUrl,
    payload: { userId: 'u2' },
  })
  assert.equal(res2.statusCode, 201)

  const res3 = await app.inject({ // 5
    method: 'POST',
    url: reserveUrl,
    payload: { userId: 'u3' },
  })
  assert.equal(res3.statusCode, 403)
  assert.deepEqual(await res3.json(), { error: 'Event is fully booked' })

  await db.close() // 6
  await app.close()
})
```

Here's what happens in this test:

1. We start by creating a fresh in-memory database and initializing the schema. We then create a Fastify app instance using this database.
2. We create a new event with two total seats using a real HTTP request to our API.
3. We check that the event creation was successful (it returned a 201), extract the event ID from the response to use it when making reservations, and build the URL for sending reservation requests for this event.

4. We successfully reserve both available seats for two different users (by sending `res1` and `res2`). For each request, we check that the reservation was created (status 201).
5. When we send a third reservation, we expect it to fail, as the event is now fully booked. This is exactly the behavior we want to verify. We do this by looking at the returned status code (we expect a 403) and the specific error message returned in the response body.
6. Finally, we close both the database and the app to ensure a clean teardown.



The `inject()` method in Fastify allows you to simulate real HTTP requests directly against your Fastify application without starting an actual HTTP server. It's a powerful tool for integration testing because it exercises your entire request lifecycle, from routing and validation to business logic and response handling, while keeping the tests fast and fully in-process. This makes it easy to test your routes in isolation, without relying on external network calls.

This concludes our first test. Now, let's write the second one and verify that trying to reserve a seat for an event that doesn't exist results in an error:

```
test('Returns 404 if event does not exist', async () => {
  const db = new DbClient(':memory:')
  await createTables(db)
  const app = await createApp(db)

  const res = await app.inject({
    method: 'POST',
    url: '/events/unknown/reservations',
    payload: { userId: 'u1' },
  })

  assert.equal(res.statusCode, 404)
  assert.deepEqual(await res.json(), { error: 'Event not found' })

  await db.close()
  await app.close()
})
```

This test follows the same structure as the previous one in terms of setup, but this time, we check what happens when a reservation is attempted for a non-existent event. The system should respond with a 404 Not Found and return a specific error message in the response body.

In this example, we've seen how integration tests can bring all the moving parts of an application together (the HTTP layer, business logic, and database access) and help us verify that they behave correctly as a whole. Starting from a simple booking system, we created real HTTP requests using Fastify's `inject()` method and verified how those requests translated into concrete side effects, such as creating events and handling reservations.

These tests gave us confidence that our application logic is not only correct in isolation but also holds up when exercised through the same interfaces a real client would use. This is exactly the value of integration testing: bridging the gap between isolated units and real-world usage.

Of course, this is just the beginning. You could expand the test suite further, for instance, by adding assertions to verify that Fastify's schema validation rejects malformed requests, or by testing other edge cases, such as duplicate reservations. We'll leave those as an exercise for you.

In the next section, we'll take things even further and explore E2E testing, where we test the system from the outside, just as a real user or external client would interact with it.

Writing E2E tests

In the previous section, we built and tested an API for booking event seats. Our tests exercised multiple layers of the application: the HTTP routes, the business logic, and the database. That's why we confidently called them integration tests: they validated that the different components of the system worked correctly when integrated together.

But you might be wondering: why not call those tests E2E tests? After all, they covered a broad part of the system, and in many real-world projects, similar tests might even be labeled that way. Truthfully, that wouldn't be wrong. The boundary between integration and E2E testing is often blurry, and there's no single universal definition that fits every project. So, the important thing is to draw the line deliberately, and in this book, we draw that line at the user.

To know what constitutes a proper E2E test, we first need to understand: what is the product, and who is the user? E2E testing is about validating the product from the user's perspective, simulating actual user behavior, and checking that the application responds in the right way, visibly and externally. That means focusing less on how the system is implemented and more on how it behaves when used "from the outside."

If your users are developers who interact with an API, then yes, the API is the product, and the previous integration tests could be considered E2E tests. But in our case, we didn't fully adopt the user's perspective. We tested the API by reaching into the internals of our system: we directly verified the database state, made multiple assumptions about how data was handled, and skipped over many aspects of how a user would actually experience the system. That's closer to white-box testing, where we use internal knowledge of the implementation to validate the outcome. E2E testing, instead, is much more of a black-box approach: it tests the system as a user sees it, without any privileged access to internal details.

To explore E2E testing in depth, we'll now take our event booking example one step further. We'll take an existing full-stack application that includes a UI, an API, and a database. Then, we'll test it the way a real user would: by controlling a browser, simulating real user flows, and checking that the application gives the right feedback at every step.

No assumptions, no peeking behind the curtain... just the full experience, as seen from the outside!

Let's get into it.

The application structure

Since our focus in this section is on E2E testing, we're not going to spend time building a full-stack application from scratch. Instead, we'll work with a pre-built application that we, your friendly authors, have prepared specifically for this purpose.

This application is a simple website where registered users can browse and book a spot for events. While intentionally minimal in terms of design and features, it includes just enough of a real-world structure to make our tests meaningful. It consists of a web frontend, a backend API, and a database. This stack gives us all the essential layers we need to write realistic E2E tests.



The source code of this application is fully available on GitHub (nodejsdp.link/events-app). There, you can find detailed instructions on how to run the application (including using Docker). If you are curious, feel free to have a look at the code, although it is not required to understand this section.

The website includes a few key sections:

- A home page, where users can browse upcoming events
- A login page, where registered users can authenticate
- A registration page, where users who don't yet have a profile can sign up
- A dashboard, visible only to logged-in users, listing their upcoming bookings
- A booking page for each event, allowing logged-in users to reserve a seat (if any are available)

Let's see some mockups to better understand what the website and each section look like.

Home page

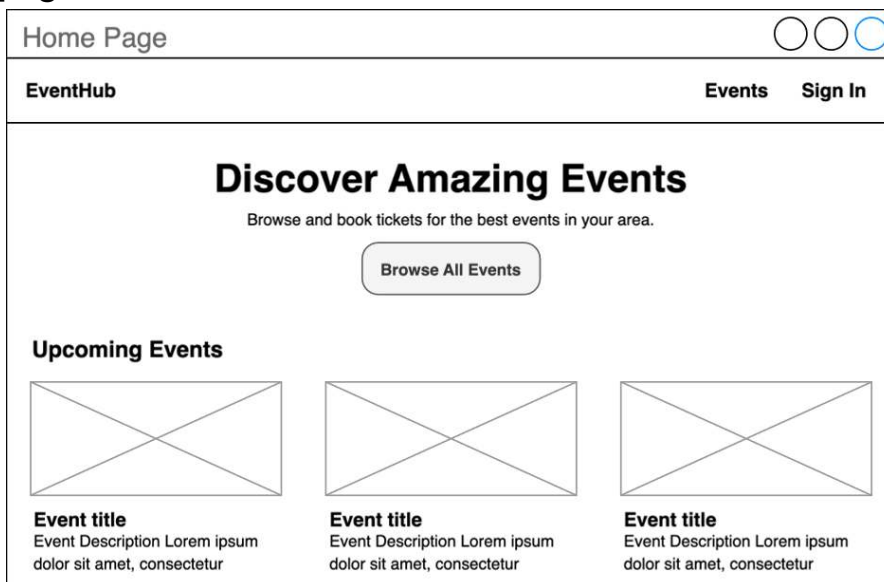


Figure 10.2 – The home page of our sample events website

This mockup shows the home page, the main entry point for most users. It lets visitors browse upcoming events and access the **Sign In** option from the navigation menu.

Sign-in and sign-up forms



Figure 10.3 – The sign-in and the sign-up pages of our sample events website

This mockup shows the **Sign In** and **Sign Up** pages of the application. These are the entry points for user authentication, allowing existing users to log in and new users to create an account.

Event Page

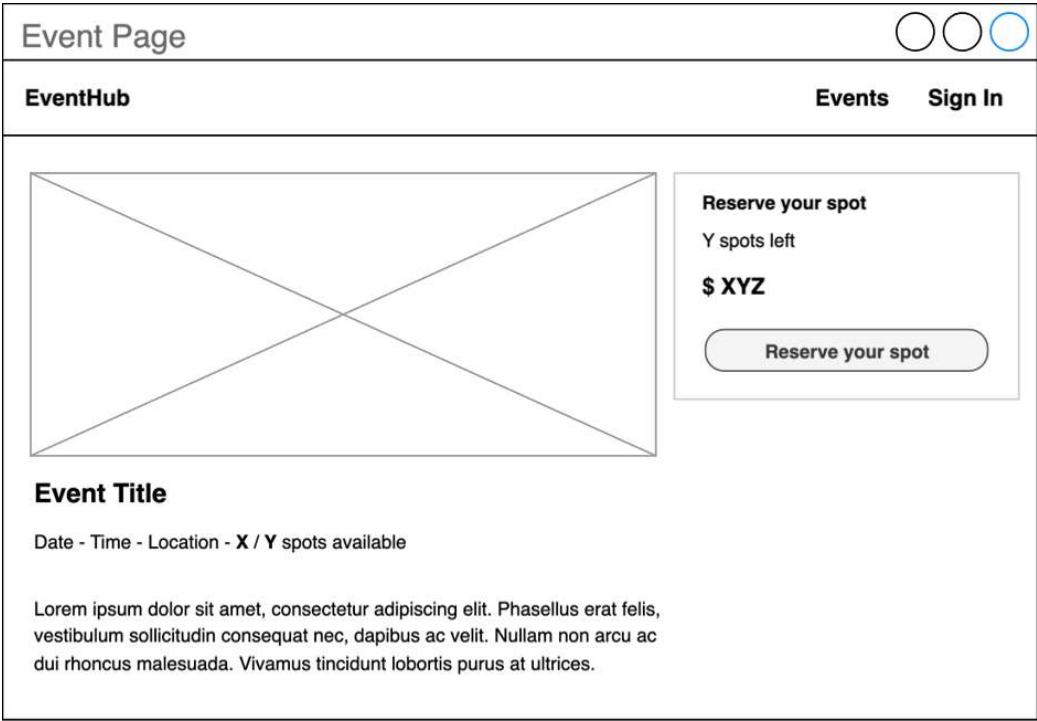


Figure 10.4 – The detailed event page of our sample events website

This mockup shows the event page, where users can view detailed information about a specific event. It includes the event title, date, time, location, availability, and description. On the right, users can see how many spots are left and click the **Reserve your spot** button to book a spot.

My reservations

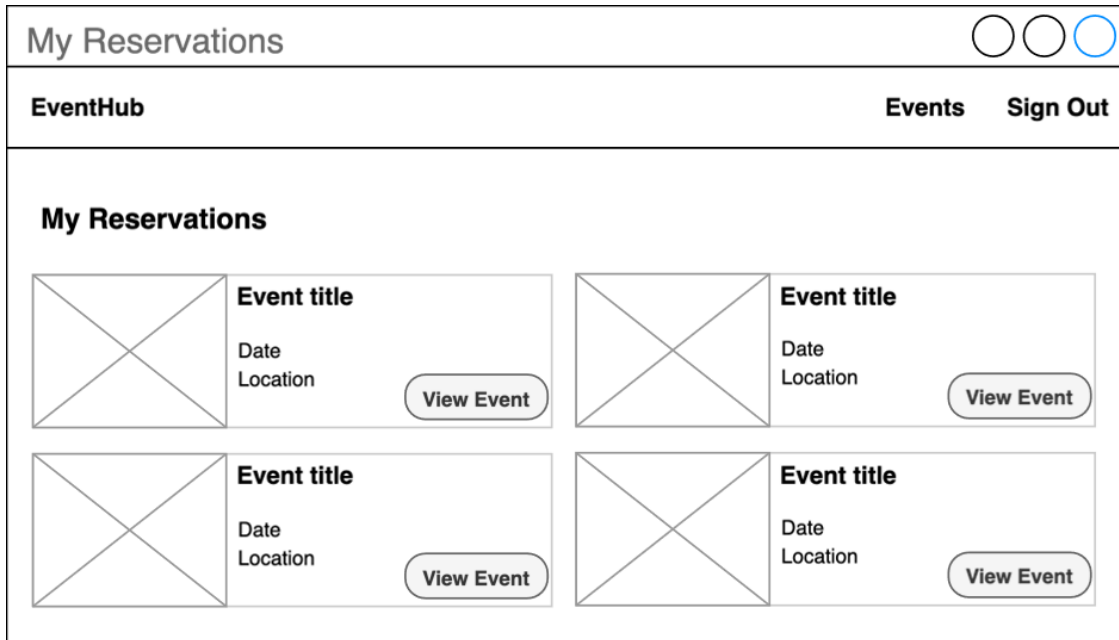


Figure 10.5 – The reservations page of our sample events website

This mockup shows the **My Reservations** page, where logged-in users can see a list of all the events they've booked. Each reservation is displayed with the event title, date, and location, along with a **View Event** button to access more details.

The user flow

Each of the sections we just described should start to make us think about potential user journeys and, therefore, opportunities for writing focused, scenario-driven tests. Our goal will be to simulate these flows programmatically, using a real browser, and validate that the application behaves correctly from the user's point of view.

Now that we have a clear idea of what this application does and how its UI is organized, let's put on our product manager hat for a moment and ask ourselves: "What's a realistic user flow that, if it works, would give us confidence that the product delivers on its promise?"

Here's a scenario that touches on almost every major part of the application:

1. A new, unregistered user lands on the home page, and they click the **Sign In** button in the navigation bar.
2. Since they don't have an account yet, they click on **Sign Up** to create one. Now they must fill in the sign-up form.
3. They enter their name.
4. They enter their email.
5. They enter their password.
6. Finally, they click the **Sign Up** button to submit the form.
7. Upon successful registration, they're automatically logged in and redirected to the home page.
8. From there, they click on one of the listed events and land on the event details page.
9. They click the **Reserve your spot** button to reserve a seat for the event, and they receive some form of feedback confirming their booking has been completed successfully.
10. Finally, they visit their dashboard (**My Reservations**).
10. Here, they expect to see the newly booked event listed.

Here's a more visual representation of this particular user flow:

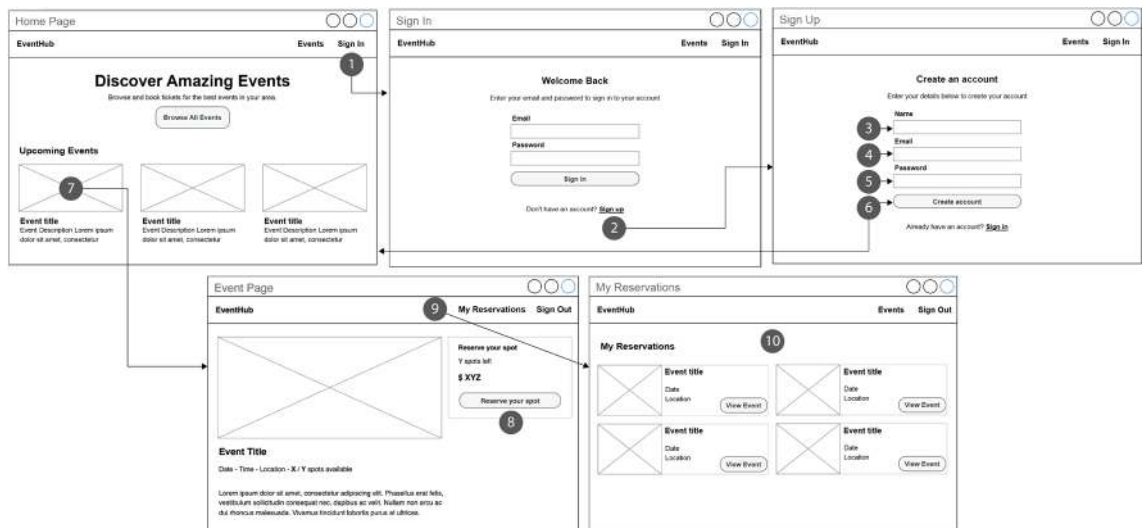


Figure 10.6 – An example user flow in our sample events website

Yes, it's a relatively long flow, but it exercises most of the core functionality of the application: navigation, registration and authentication; form submission; state transitions; and key interactions with the data layer. If this test passes, we can be reasonably confident that the system works correctly from a user's perspective and that the main pieces of the stack are functioning together as expected.

Browser automation

To properly test our application from the user's perspective, we need to go one step beyond traditional API or integration testing: we need to automate the browser itself. This is what E2E testing is all about: simulating real user behavior, as if someone were navigating our app in Chrome, Firefox, or Safari, clicking through forms, entering data, and receiving feedback from the interface.

To do that, we need tools that can control a browser programmatically. Over the years, the ecosystem of browser automation tools has grown significantly. Here are a few notable options:

- **Selenium** (nodejsdp.link/selenium): One of the oldest and most battle-tested tools in the space. It supports many languages and browsers but can feel heavyweight and slower to set up.
- **Puppeteer** (nodejsdp.link/puppeteer): A Node.js library originally developed by Google for controlling headless Chrome. It's lightweight and scriptable, great for basic automation.
- **Cypress** (nodejsdp.link/cypress): A popular testing framework focused on frontend testing. It provides a complete testing environment with an interactive UI, strong assertions, and a built-in test runner. However, it's somewhat opinionated and less suited for backend-heavy E2E flows.
- **Playwright** (nodejsdp.link/playwright): A newer player developed by Microsoft and inspired by Puppeteer, and the tool we'll be using in this section.

Some of these tools are primarily libraries. They let you script browser behavior, but you still must pair them with your own testing infrastructure. Others, such as Cypress and Playwright, offer much more than just automation. They include powerful testing frameworks, assertion libraries, debugging tools, and test runners, all designed to simplify the entire E2E testing experience.

That's exactly why we've chosen Playwright. **Playwright** is an open-source framework for web testing and automation, backed by Microsoft and actively maintained. It supports all major browser engines (Chromium, Firefox, and WebKit) through a single unified API, and it works across different OSs and architectures. One of its standout features is its multi-language support: you can use Playwright in JavaScript, TypeScript, Python, Java, and .NET.



While all language bindings are well supported, the JavaScript and TypeScript versions are the most actively maintained and feature-rich. New features typically land in the JavaScript/TypeScript ecosystem first, making it the best choice if you want early access to improvements and the full range of capabilities.

Another benefit? Installing Playwright automatically downloads the necessary browser binaries for your system (whether you're on macOS, Windows, or Linux) with a single command. You don't need to manage browser installations or versions manually: Playwright handles that for you.



While Playwright is our tool of choice in this chapter, we encourage you to experiment with the others as well. Each has its strengths and trade-offs, and the best choice often depends on the specific needs of your project or team.

That said, let's now see how we can use Playwright to write a real E2E test for the booking flow we described earlier.

Writing an E2E test with Playwright

Now that we've discussed why browser automation is essential for E2E testing and why we chose Playwright, it's time to get our hands dirty and start writing a real E2E test.

Setting up a new Playwright project

The first step is to initialize a new Playwright project. You can do this with a single command:

```
npm init playwright@^1.51.1
```



This is the version we used while writing this example. You can use the latest version instead, but keep in mind that some breaking changes may have been introduced after this book was published.

When you run the command, Playwright will launch an interactive setup tool that walks you through configuring your project. Here's what you can expect:

- **Choose between TypeScript and JavaScript:** The default is TypeScript, which is what Playwright recommends.
- **Name your tests folder:** The default is `tests`, but if your project already has a `tests` folder, it will suggest `e2e` instead.
- **Add a GitHub Actions workflow:** This creates a CI configuration file so you can easily run your Playwright tests in GitHub Actions.
- **Install Playwright browsers:** The default is `true`, and you should leave it as is. Playwright will download the appropriate browser binaries for your OS and CPU automatically.

Once the setup completes, Playwright installs its core libraries and generates a few different files in your project directory. These are the most important ones:

- The `playwright.config.ts` file is where you can configure how Playwright runs. You can customize things such as which browsers to test, how many parallel workers to use, whether tests should run in headless mode (meaning it won't render a browser window but simply run the tests in the background), and the global timeout for each test.

- The `tests-examples` folder contains a more detailed test of a simple to-do application. This is a great place to start to get familiar with the Playwright API.
- The `tests` folder (or `e2e`, or whatever you configured during the setup phase) with a simple example test.

If you're adding Playwright to an existing project, the setup tool will simply add the necessary dependencies to your `package.json`; otherwise, a new one will be generated for you.

Once everything is set up, you can run the test suite with the following:

```
npx playwright test
```

By default, Playwright will do the following:

- Run tests on all three supported browsers: Chromium, Firefox, and WebKit
- Run test files in parallel and strive for optimal utilization of CPU cores on your machine
- Run tests in headless mode (no browser UI will appear)
- Show test results and logs directly in the terminal

If you'd like to actually see your tests run in a real browser window, you can launch them in headed mode using the following:

```
npx playwright test --headed
```

This opens a browser and visually walks through every step defined in your test: clicks, form submissions, page transitions, everything. Once the test completes, Playwright automatically generates an interactive HTML report where you can inspect each test in detail, see what steps were executed, what passed, and what failed.

It's already quite convenient. But for an even more powerful testing and debugging experience, Playwright provides a dedicated UI mode. You can start it with the following:

```
npx playwright test --ui
```

This launches a full visual test runner built right into Playwright. It offers a developer-friendly interface where you can explore your test files, run individual tests or suites, and step through them using a time-travel debugger. You can pause execution, hover over each action to see what was happening in the browser at that exact point, and even pop out DOM snapshots for deeper inspection. You'll also be able to filter tests by status, name, tags, or project configuration, and watch changes live as you iterate on your code.

UI mode turns Playwright into more than just a testing tool: it becomes a complete workspace for building, debugging, and refining E2E tests with speed and precision. This is what we recommend you use while you are authoring your tests to make sure your tests are performing the right sequence of actions and that every action is completed as expected.

Understanding the Playwright API

At a high level, writing Playwright tests isn't that different from writing the unit or integration tests we've already explored using Node.js's built-in test runner. The general principle remains the same: you perform actions (in this case, through a browser), and then assert the resulting state to match your expectations.

However, working in a browser environment introduces a new layer of complexity. Simulating user actions means dealing with an inherently asynchronous and often unpredictable UI: pages re-render, transitions happen, network calls are triggered in the background, and loading times vary. A big challenge in browser testing is understanding when a page or element has reached a “steady” state before proceeding to the next action.

Take this simple example: a user clicks a link that navigates to a new page, and we want to perform another action on that page. How do we know the page has finished loading and is ready to interact with? In traditional test setups, this often leads to fragile tests filled with arbitrary timeouts or complex wait conditions.

Playwright solves this elegantly. In fact, Playwright is designed to automatically wait for the necessary conditions before performing each action. It performs a wide range of actionability checks (such as ensuring elements are visible, attached to the DOM, enabled, and stable) before interacting with them. There's no need to sprinkle manual waits before clicking a button or filling a field.

Similarly, Playwright's assertions are built to handle this kind of uncertainty. Rather than checking a condition immediately, Playwright treats assertions as expectations that should eventually become true. If the expectation isn't met within a default timeout, the test fails. This design removes much of the guesswork and flakiness often associated with E2E testing.

Let's look at a simple example to get a feel for how Playwright code looks in practice:

```
import { test, expect } from '@playwright/test'

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/')

  // Expect a title "to contain" a substring.
  await expect(page).toHaveTitle(/Playwright/)
})

test('get started link', async ({ page }) => {
  await page.goto('https://playwright.dev/')

  // Click the get started link.
  await page.getByRole('link', { name: 'Get started' }).click()

  // Expects page to have a heading with the name of Installation.
```



```
await expect(
  page.getByRole('heading', { name: 'Installation' })
).toBeVisible()
})
```

Here, we're defining two distinct test cases using the `test()` function, which comes from the Playwright testing library and is quite similar in structure to the `test()` function in the Node.js test runner.

Inside each test, we perform actions using the `page` object, which represents a browser tab. Each interaction returns a promise, so we use `await` to ensure each step completes before moving to the next. This is crucial because most Playwright actions involve multiple internal checks, such as waiting for a page to load or an element to become visible, before the action is considered successful.

For assertions, Playwright provides its own `expect()` function. Unlike Node's built-in `node:assert` module, which checks conditions directly, `expect()` requires you to wrap the value under test and gives you a fluent interface to define expectations. For example, `expect(page).toHaveTitle()` checks the page's title, and `expect(element).toBeVisible()` ensures an element is currently visible in the DOM. Assertions return a promise that will eventually either be fulfilled (the assertion passed) or rejected (the assertion failed), so we need to make sure we `await` it before moving on to the next step.

Let's now deep dive a bit more into some of the actions you can perform with the Playwright API.

Navigation

Most Playwright tests begin by navigating the browser to a specific URL using `page.goto()`. The following call is made:

```
await page.goto('https://playwright.dev/')
```

Then, Playwright automatically waits for the page to reach a fully loaded state before proceeding. This ensures that the test only continues once the page is ready for interaction. You can customize this behavior using options provided by `page.goto()` if you need to wait for a different load state or adjust the timeout.

The Locator API

Before we can interact with elements on a page (such as clicking a button or filling out a form), we need a reliable way to find them. That's where Playwright's Locator API comes into play.

Locators are powerful abstractions that allow you to target elements on the page at any given moment, regardless of when they appear or how they change during navigation or re-rendering. Instead of querying elements once and holding on to a reference, locators in Playwright always act on the latest state of the page, ensuring your tests remain stable even as the UI evolves dynamically.

For example, here's how you can create a locator and interact with it:

```
const getStarted = page.getByRole('link', { name: 'Get started' })
await getStarted.click()
```

In most cases, you'll write this in one line for convenience:

```
await page.getByRole('link', { name: 'Get started' }).click()
```

There's no need to add manual `waitFor` logic. Playwright automatically waits for the element to be visible, attached to the DOM, and ready for interaction before executing the action. This makes your tests more readable and far less fragile.

These are the most used built-in locators:

- `page.getByRole()` to locate by explicit and implicit accessibility attributes
- `page.getByText()` to locate by text content
- `page.getByLabel()` to locate a form control by the associated label's text
- `page.getByPlaceholder()` to locate an input by placeholder
- `page.getByAltText()` to locate an element, usually an image, by its text alternative
- `page.getByTitle()` to locate an element by its title attribute
- `page.getByTestId()` to locate an element based on its `data-testid` attribute (other attributes can be configured)

Once you've located an element, you can perform a wide range of actions. Here are a few of the most used methods:

- `locator.click()`: Click the element
- `locator.fill()`: Fill a text input or form field
- `locator.check()`: Check a checkbox
- `locator.uncheck()`: Uncheck a checkbox
- `locator.hover()`: Hover over the element
- `locator.focus()`: Programmatically focus the element
- `locator.press()`: Simulate a keyboard key press
- `locator.setInputFiles()`: Upload a file
- `locator.selectOption()`: Choose an option from a dropdown



Playwright offers many more actions and configuration options, so be sure to explore the official Locator API documentation (nodejsdp.link/playwright-locator) as you build more complex tests.

Thanks to locators, Playwright gives you a high-level and consistent way to describe interactions in your tests without worrying about low-level browser details or flaky timing issues.

Assertions

As we mentioned before, Playwright comes with a built-in assertion library that makes it easy to verify that your application behaves as expected. At the core of it is the `expect()` function, which you use to make assertions about values, elements, or page properties.

The basic syntax looks like this:

```
expect(something).toBeTruthy()
```

You pass a value to `expect()`, then call a matcher to express what you expect to be true. Playwright supports common matchers such as `toEqual()`, `toContain()`, and `toBeTruthy()`, which can be used for simple, synchronous assertions.

But what makes Playwright especially powerful for E2E testing is its support for asynchronous assertions. These aren't just one-off checks; they wait until the condition becomes true, up to a timeout. This is key to writing reliable, non-flaky tests, especially when dealing with dynamic pages where content may load or update over time. In Playwright lingo, these assertions are often referred to as *web-first assertions*.

For example, this assertion waits for the page title to contain the word Playwright:

```
await expect(page).toHaveTitle(/Playwright/)
```

Rather than immediately checking the title, Playwright polls the value and retries until the condition is met or the timeout is reached.

Here are some of the most commonly used async assertions you'll encounter in browser tests:

- `expect(locator).toBeVisible()`: Waits for an element to become visible
- `expect(locator).toBeEnabled()`: Checks that a control is enabled
- `expect(locator).toBeChecked()`: Verifies that a checkbox is selected
- `expect(locator).toHaveText()`: Confirms that an element contains specific text
- `expect(locator).toContainText()`: Matches part of the text content
- `expect(locator).toHaveAttribute()`: Ensures an element has a specific attribute
- `expect(locator).toHaveValue()`: Checks the value of an input field
- `expect(locator).toHaveCount()`: Verifies the number of matching elements
- `expect(page).toHaveTitle()`: Waits for a page to have a specific title
- `expect(page).toHaveURL()`: Confirms the current page URL

All these matchers are designed to be resilient to real-world timing issues. Instead of forcing you to manually wait for the UI to “settle,” Playwright's assertions do it for you, allowing your tests to focus on what needs to be true, not when it becomes true.

As your test suite grows, you'll find that Playwright's assertion library is one of its most powerful features. It makes your tests expressive, reliable, and, most importantly, closely aligned with real user behavior.

Understanding timeouts

As we mentioned in the previous sections, Playwright tries to make your life easier by helping you manage the asynchronous nature of E2E testing. Some operations have sensible default timeouts, while others, by default, will only fail when the overall test timeout is reached. This difference often causes confusion when getting started, so let's break down the default behavior and how you can customize it.

Not all operations behave the same way when it comes to timeouts:

- **Web-first assertions:** Certain assertions automatically wait for a condition to become true before proceeding. Examples include `expect(locator).toBeVisible()`, `expect(locator).toBeEnabled()`, and `expect(locator).toHaveText()`. These web-first assertions retry for up to five seconds by default, making tests more resilient to timing issues such as delayed rendering or animations. You can override this with the `timeout` option on the assertion.
- **Synchronous assertions:** Assertions that don't depend on the page state, such as `expect(value).toBe(42)`, are evaluated immediately and have no built-in retry behavior. If the expectation fails, the test fails right away.
- **Actions:** Actions such as `locator.click()` and `locator.fill()` follow the action timeout, which defaults to 0 (no per-action limit). They perform actionability checks (e.g., element is visible, enabled, stable, or not animating) before proceeding. If those checks never pass, the action can hang until the overall test timeout is reached, unless you set a timeout on the call or configure a global `actionTimeout`.
- **Navigations:** Operations such as `page.goto()` and `page.waitForURL()` use the navigation timeout, which also defaults to 0 in Playwright. You can set it per call with `timeout` or globally with `navigationTimeout`.

In short, only web-first assertions have a built-in five-second retry by default. All other operations will wait indefinitely unless you configure timeouts, which is why tests often fail only when the overall test timeout is reached. To fine-tune this behavior, you can set timeouts per call or adjust them globally with the `timeout`, `actionTimeout`, and `navigationTimeout` configuration options.



It's also worth noting that locators such as `page.getByRole()` are *lazy definitions*: they don't perform any action by themselves and therefore have no timeout. Timeouts only come into play once a locator is used in an action or an assertion.

Understanding these differences early on will save you a lot of debugging time and help you avoid flakiness caused by mismatched timeout expectations.

Testing our user flow

Now that we've covered the basics of Playwright and walked through a few examples, it's time to apply that knowledge to something real. Let's implement the full user journey we outlined earlier and turn it into a complete E2E test.

To recap, here's the flow we want to verify:

1. A new, unregistered user lands on the home page and clicks the **Sign In** link.
2. They click **Sign Up** to create an account and fill in the registration form.
3. They enter their name, email, and password.
4. They submit the form and are redirected to the home page.
5. From the home page, they click on an event to view its details page.
6. They click **Reserve your spot** to book a seat.
7. They see visual feedback that confirms the booking was successful.
8. They navigate to their **My Reservations** page.
9. They see the newly booked event listed there.

Let's see how this looks in code using Playwright.



This test assumes you already have the sample application running locally at `http://localhost:3000`. If not, make sure to start the app before running the test. Detailed instructions on how to do that are available in our code repo and in the specific test project README on GitHub (nodejsdp.link/events-app).

Starting the flow

We begin the test by navigating to the root of our local development server. This simulates a new user landing on the home page.

```
import { test, expect } from '@playwright/test'

test('A user can sign up and book an event', async ({ page }) => {
  await page.goto('http://localhost:3000')
  // ...
}
```

Navigating to Sign Up

From the home page, the user clicks on **Sign In** and then on **Sign up** to register a new account.

```
// ...
await page
  .getByRole('link', { name: 'Sign In' })
  .click()

await page.getByRole('link', { name: 'Sign up' }).click()
// ...
```

Filling the registration form

To avoid conflicts between test runs, we generate a unique name and email address using a timestamp seed. The form fields are filled out, and the **Create account** button is clicked.

```
// ...
const seed = Date.now().toString()
const name = `TestUser ${seed}`
const email = `test${seed}@example.com`
const password = `someRandomPassword${seed}`

await page.getByRole('textbox', { name: 'name' }).fill(name)
await page.getByRole('textbox', { name: 'email' }).fill(email)
await page.getByRole('textbox', { name: 'password' }).fill(password)

await page.getByRole('button', { name: 'Create account' }).click()
// ...
```

Booking an event

After account creation, the user clicks into a specific event. We read the initial number of available seats, click the **Reserve your spot** button, and wait for a successful reservation request to complete.

```
// ...
await page.getByRole('link', { name: 'Marathon City Run' }).click()

const availableCapacity = Number.parseInt(
  (await page.getByTestId('available-capacity')).textContent()) as string
)

await page.getByRole('button', { name: 'Reserve your spot' }).click()

// ...
```

Verifying the booking

Here, we verify that the page reflects a successful booking. We check for a Booked badge and confirm that the booking button is now saying **You have booked this event!** and that it is disabled. Finally, we ensure the number of available seats has gone down.

```
// ...
await expect(page.getByTestId('badge').first()).toHaveText('Booked')
const bookButton = await page.getByRole('button',
  { name: 'You have booked this event!' })
await expect(bookButton).toBeDisabled()
await expect(bookButton).toBeVisible()
```

```
const newAvailableCapacity = Number.parseInt(  
  (await page.getByTestId('available-capacity').textContent()) as string  
)  
expect(newAvailableCapacity).toBeLessThan(availableCapacity)  
// ...
```

Checking the dashboard

Lastly, the user navigates to the My Reservations section and confirms that the event they just booked appears on their dashboard.

```
// ...  
await page  
  .getByRole('link', { name: 'My Reservations' })  
  .click()  
  
await expect(  
  page.getByRole('heading', { name: 'My Reservations' })  
)  
  .toBeVisible()  
  
expect(  
  await page.getByRole('heading', { name: 'Marathon City Run' })  
)  
  .toBeVisible()  
  
// ...
```

This test simulates a complete, real-world user journey, from account creation to event reservation, and confirms that the application behaves correctly at every step. Thanks to Playwright's smart waiting mechanisms and expressive APIs, we can write a full E2E test like this in a clear, resilient, and highly readable way.

Final considerations

As we close out this example, it's important to recognize that even though our test was relatively straightforward, it still makes several assumptions that may not hold true in every environment.

For instance, our test expects to find an event called Marathon City Run already listed on the home page. This assumes that such an event exists and is consistently available. In real-world situations, especially in dynamic environments such as production, this might not be the case: new events may be added or removed by users, and data can change rapidly.

To avoid these issues, it's common to run E2E tests in a dedicated staging or QA environment. These environments are typically seeded with predictable test data before each test run. This allows the test suite to run against a known and stable dataset, reducing the chances of flakiness due to inconsistent or missing data.

Another simplification we made is in the sign-up flow. Our test user can register and start booking immediately. Most real-world applications will require email confirmation before activating a new account. This introduces new complexity into E2E testing: should we simulate the email flow using a real email inbox, intercept emails in a test service such as MailHog (nodejsdp.link/mailhog), or bypass the verification step altogether in the test environment? These decisions require coordination between developers, testers, and product teams.

You might also have noticed that events in our application have a price, but we don't process any payment during the booking. This was a deliberate choice to keep things simple, but in a real-world scenario, you'd likely need to integrate with a payment processor such as Stripe or PayPal. Testing payments in E2E flows can be challenging. One approach is to use sandbox environments provided by these services, which simulate real transactions without moving money. Another option is to mock or bypass the payment step entirely in your test environment to isolate the business logic from the third-party integration.



Some companies take things a step further. One of the authors (Luciano) previously worked at a company that ran E2E tests by making real payments to ensure absolute confidence that the payment flow worked in production-like conditions. They used a company credit card to process small transactions against their own merchant account. Since the money remained within the same organization, only minimal transaction fees were lost. However, this required careful tracking and coordination with the finance team to properly reconcile those transactions, especially for accounting and tax purposes.

The same goes for social login systems such as Sign in with Google or Login with GitHub. These can be difficult to test reliably in E2E scenarios since they involve redirects, external UI flows, and sometimes even CAPTCHA challenges. In test environments, it's common to stub or mock these login flows, or to offer a special “test-only” login route that bypasses third-party authentication altogether.

Another key consideration when writing resilient E2E tests is how we locate elements in the DOM. In our test, we mostly relied on accessible attributes such as `aria-label`, `role`, or visible text. This approach is more aligned with how users interact with the UI, but it also means our tests can break if labels or button text change. Occasionally, we used `data-testid` attributes (through the `page.getByTestId()` locator) as a more stable locator strategy. This works well when there's close collaboration between frontend developers and testers, and there's an agreement on which elements will expose test IDs. However, if the layout or naming conventions change, those IDs must be kept in sync to avoid broken tests.

All this highlights one fundamental truth about E2E testing: it's not easy.

E2E tests are powerful because they give us confidence that the entire system works together as expected (from the UI all the way to the database and back), but they also come with maintenance costs and complexity. Writing resilient E2E tests requires not just technical know-how but also cross-functional collaboration between development, QA, and product teams. Everyone needs to be aligned on how the application behaves, what data is needed for tests, and how to keep the testing environment stable.

Done well, E2E tests provide an invaluable safety net and allow you to confidently ship features knowing the most important user flows are covered. But like any good safety net, they require care, consistency, and coordination to be truly effective.

Summary

We opened this chapter by introducing the foundations of software testing: what it is, why it matters, and how to approach it. We defined essential concepts such as SUT, the Arrange-Act-Assert structure, and code coverage, and explored the role of test doubles such as stubs, spies, and mocks.

From there, we covered TDD and BDD, before placing testing in the context of modern workflows such as CI and continuous delivery. We then explored 11 principles for effective testing, offering a mental framework to guide us through real-world challenges such as flaky tests, test interdependence, and the limitations of coverage metrics.

With the theory in place, we moved into practice: starting with unit tests in Node.js using the built-in `node:test` runner. We tested both sync and async code and then focused on mocking, demonstrating how to isolate units from their dependencies, including HTTP calls, built-in modules, and custom internal modules. We then introduced DI as a cleaner alternative for decoupling and testability.

Next, we turned to integration testing, wiring up a real SQLite database to test our business logic and HTTP endpoints together. From there, we stepped into E2E testing by introducing Playwright, a powerful browser automation tool. We built a full-stack user flow (from sign-up to booking an event and checking reservations), testing the application as a user would, through the UI and without peeking inside.

Throughout, we emphasized writing tests that balance isolation with realism, and reliability with maintainability.

In the next chapter, we'll go beyond the fundamentals and into territory where only experienced Node.js developers tend to tread. We'll explore advanced patterns and techniques such as delayed initialization, request batching and caching, and cancellation techniques. These are the kinds of problems that arise in production-grade systems, and mastering them will take your skills to the next level. Ready to level up? Let's dive in.

Exercises

- **Exercise 10.1: Unit test for a utility function:** Create a function called `slugify(text)` that turns a string into a URL-friendly slug. For example, `Hello World!` should become `hello-world`. Write a unit test using Node.js's built-in test runner to verify the following:
 - The output is lowercase
 - Special characters are removed
 - Multiple spaces or dashes are collapsed into a single dash

- **Exercise 10.2: Test an asynchronous retry function:** Implement a function called `fetchWithRetry(asyncFn, maxRetries)` that retries the given asynchronous function up to `maxRetries` times if it throws. The `asyncFn` can be any asynchronous function (e.g., a `fetch` or database call). If `asyncFn` eventually succeeds, `fetchWithRetry()` should resolve to the same value. If it fails on every attempt, it should reject with the last error. Here are some suggestions:
 - Simulate an `asyncFn` that fails twice and succeeds on the third attempt.
 - Assert that it returns the expected value and that `asyncFn` was called three times (a great use case for `mock.fn()`).
 - Write another test where `asyncFn` always fails and confirm that the function rejects and is called exactly `maxRetries` times.
- **Exercise 10.3: Is it hot in here?** Write a function called `isHotIn(city)` that checks whether the weather in a given city is “hot.” The function should make an HTTP request using `fetch` to retrieve weather data, extract the temperature, and return `true` if it’s above a certain threshold (e.g., 30°C or 86°F, your choice!). You don’t need to call a real API; just simulate a response such as `{ temp: 31, unit: 'C' }` or `{ temp: 88, unit: 'F' }`. Then, write a test for `isHotIn(city)` without making a real network request. You can either mock the global `fetch()` function or refactor the function to accept a `weatherClient` dependency and inject a mock. In your test, do the following:
 - Verify that the `fetch` was called with the expected URL or parameters
 - Simulate both a “hot” and “not hot” response
 - Assert that `isHotIn()` returns the correct Boolean value in both cases
- **Exercise 10.4: Where’s my pizza?** Create a class called `PizzaTracker` that manages pizza orders using an SQLite database. The class should use a table named `orders` with the following columns:
 - `id` (string, primary key)
 - `customerName` (string)
 - `pizzaType` (string)
 - `status` (string, e.g. pending or delivered)
 - `eta` (integer, optional: represents estimated delivery time in minutes)

Implement the following methods:

- `placeOrder(id, customerName, pizzaType)`: Inserts a new order into the table with status pending and eta set to null
- `getOrders()`: Retrieves all orders from the table
- `markAsDelivered(id)`: Updates the status of a specific order to delivered
- `updateEta(id, eta)`: Updates the estimated delivery time for a given order

Now, write an integration test that does the following:

- Uses an in-memory SQLite database
 - Adds three different orders using `placeOrder()`
 - Updates the ETA for one of the orders and marks another one as delivered
 - Asserts that `getOrders()` returns all three orders
 - The status and eta fields reflect the updates correctly
- **Exercise 10.7: The spammers' heaven:** Build a minimal web application that serves a single HTML page with a newsletter sign-up form. You can use any HTTP framework you like (Express, Fastify, or even Node.js's built-in `createServer()` from `node:http`) as long as it serves an HTML page with the following content and behavior:
 - A form with an input field for the user's email
 - A submit button
 - When the form is submitted, the app should respond with a confirmation message such as: "Thanks for subscribing, `test@example.com`!"

Now, write an E2E test using Playwright that does the following:

- Launches the app locally
- Navigates to the sign-up page
- Fills in the email field with `test@example.com`
- Submits the form
- Asserts that the confirmation message is visible on the page

11

Advanced Recipes

If you have ever tried to cook a fancy dinner while friends keep showing up early, you already know the chaos of asynchronous programming. Some things need to bake, others need to chill, and somehow you must keep everything moving without burning the kitchen down. Sometimes, working with Node.js can feel the same. In this chapter, we take a problem-solution approach and, much like a trusted cookbook, offer you ready-to-use recipes to navigate common Node.js challenges.

You should not be surprised that most of the problems we explore here arise when we try to perform tasks asynchronously. In fact, as we have seen repeatedly in the previous chapters, tasks that are trivial in traditional synchronous programming can become more complicated when applied to asynchronous programming. A typical example is using a component that requires an asynchronous initialization step. In such cases, we face the inconvenience of having to delay any attempt to use the component until its initialization is complete. Later, we will show you how to handle this elegantly.

However, this chapter is not only about recipes related to asynchronous programming. You will also learn the best ways to run CPU-intensive tasks in Node.js.



To make this more concrete, here's a real-world example from Mario.

A few years ago, I was tasked with adding a feature to export an audit log to PDF. As you might guess, these logs could grow enormous, sometimes spanning thousands of pages. The plan was straightforward: generate a print-ready HTML page, then use Puppeteer (nodejsdp.link/puppeteer) to automatically convert it into a PDF. I assumed that since Puppeteer runs in an external process, even massive files would not affect the main application. But my first tests proved otherwise. The HTML template engine we used was synchronous, and for very large logs, it blocked the Node.js event loop for several seconds. Suddenly, the server would freeze just long enough to be noticeable—and I had not anticipated that.

The fix was to offload PDF generation to a separate Node.js process and use a pool of workers for print requests. This kept the main app responsive regardless of log size. That experience was a clear reminder of how easy it is to run into CPU-intensive operations in Node.js, and why it is so important to know how to handle them.

Here are the recipes you will learn in this chapter:

- Dealing with asynchronously initialized components
- Asynchronous request batching and caching
- Canceling asynchronous operations
- Running CPU-bound tasks

Let's get started.

Dealing with asynchronously initialized components

Throughout this book, we have stressed how important asynchronous programming is in JavaScript and Node.js, and we have learned a great deal about different ways and patterns to use asynchronous programming effectively. At this point, you might be wondering: why are there still so many synchronous APIs in Node.js, such as `readFileSync()` from `node:fs`?

One of the reasons synchronous APIs exist in the Node.js core modules and many npm packages is that they are convenient for handling initialization tasks. In simple programs, using synchronous APIs during initialization can simplify things significantly, and the typical drawbacks of synchronous code remain contained since they only affect the program once, at startup or when a particular component is initialized.

However, relying on synchronous APIs is not always possible. In some cases, a synchronous version may not be available, especially for components that need to perform network operations during their initialization phase, such as executing handshake protocols or retrieving configuration parameters. This is common with many database drivers and clients for middleware systems such as message queues, where most libraries are designed to offer only asynchronous APIs by nature.

In other cases, you might need to initialize something within an asynchronous code path (for example, in a route handler of a web application). In situations like these, using a synchronous API is not ideal because you risk blocking the event loop for an extended period, potentially making the entire web server unresponsive during the synchronous operation. Far from ideal!

The issue with asynchronously initialized components

Let's look at an example involving a module called `db`, which is used to interact with a remote database. The `db` module can accept requests only after it has completed the connection and handshake with the database server. Until this initialization phase is finished, no queries or other commands can be sent.

Note that, to keep things simple, this `db` module does not implement a real database connector but closely mimics the general structure and behavior of a realistic one. Here is the code for this sample module (`db.js`):

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

class Database {
  connected = false
```

```
#pendingConnection = null

async connect() {
  if (!this.connected) {
    if (this.#pendingConnection) {
      return this.#pendingConnection
    }
    // simulate the delay of the connection
    this.#pendingConnection = setTimeout(500)
    await this.#pendingConnection
    this.connected = true
    this.#pendingConnection = null
  }
}

async query(queryString) {
  if (!this.connected) {
    throw new Error('Not connected yet')
  }
  // simulate the delay of the query execution
  await setTimeout(100)
  console.log(`Query executed: ${queryString}`)
}

export const db = new Database()
```

This code defines a simple `Database` class that simulates the behavior of a database client. The `connect()` method simulates creating a connection to the database. It checks whether a connection needs to be established, and then it mimics a connection delay by waiting for 500 milliseconds. Once the connection is established, it sets the `connected` flag to `true`.



Note that, here, the private member `#pendingConnection` holds a promise representing an ongoing connection attempt. Before starting a new connection, we check whether this promise already exists. If it does, it means a connection is in progress, so instead of creating a new promise (and starting another connection), we simply reuse the existing one. This ensures that multiple calls to `connect()` before the first connection is established will *piggyback* on the same promise, preventing duplicate connections. This technique is often referred to as **promise piggybacking**, and it forms the foundation for another useful pattern called **request batching**. We will explore both in more detail later in this chapter.

The `query()` method simulates sending a query to the database. Before executing the query, it checks whether the connection has been established. If not, it throws an error to indicate that the database is not ready yet. If the connection is ready, it simulates a short delay (100 milliseconds) and then logs the query string to the console. At the end of the file, an instance of this `Database` class is created and exported as `db`, making it ready for use in other parts of the application (as a singleton).

This is a typical example of a component that requires asynchronous initialization. However, the current implementation is prone to misuse because application code might attempt to execute a query before the client has finished connecting.

There are a few ways this can happen:

- A user might forget to call the `connect()` method before executing a query.
- A user might call `connect()` but forget to await the call, effectively not waiting for the connection to be established.

Here is how these mistakes might look in code:

- Forgetting to call `connect()`:

```
import { db } from './db.js'
const users = await db.query('SELECT * FROM users')
```

- Forgetting to await the call to `connect()`:

```
import { db } from './db.js'
db.connect() // no await before this call!
const users = await db.query('SELECT * FROM users')
```

In both cases, executing the code will result in an error being thrown (Not connected yet).

We usually have two quick and easy solutions for situations like these, which we can call *local initialization check* and *delayed startup*. Let's analyze them in more detail.

Local initialization check

The first solution, perhaps the most straightforward, is to ensure that the module is initialized before any of its APIs are invoked. If it is not, we wait for its initialization before proceeding. This check needs to be performed every time we want to call an operation on the asynchronous module.

Here's what that looks like in code:

```
import { db } from './db.js'

async function getUsers() {
  if (!db.connected) {
    await db.connect()
  }
}
```

```
    await db.query('SELECT * FROM users')
  }
```

As you can see, before executing a query, we check whether the `db` instance is already connected. If it is not, we explicitly wait for the connection to complete by calling and awaiting `connect()`.



We could have avoided adding the `if` condition here, since our database client already checks internally whether it is connected before performing a connection. Adding the check at this point can be seen as a small performance optimization: if the client is already connected, we can skip calling and awaiting an asynchronous operation, thus avoiding an unnecessary deferral of the function's execution to a later cycle of the event loop.

In the example above, the responsibility for ensuring the connection is ready falls entirely on the consumer of the `db` module.

A common variation of this technique is to move the connection check inside the `query()` method itself. This shifts the burden from the consumer to the provider: the module would take care of ensuring it is initialized before executing any query, sparing the users from writing repetitive connection-checking code every time they interact with the database.

Delayed startup

The second quick (and somewhat dirty) solution to the problem of asynchronously initialized components is to delay the execution of any code that relies on the component until after its initialization routine has completed.

If we think about our previous database client example, one simple way to implement this idea is to create a factory function that returns a database client that is already connected. By doing this, we ensure that we obtain a connected client once and can then use it safely without having to worry about managing the connection state throughout the rest of our code.

Here's an example:

```
import { db } from './db.js'

async function getConnectedDb() {
  await db.connect()
  return db
}

async function getUsers(db) {
  await db.query('SELECT * FROM users')
}

const connectedDb = await getConnectedDb()
await getUsers(connectedDb)
```


As you can see, we introduced a factory function called `getConnectionDb()`, which ensures the database client is connected before returning it. We also slightly updated the `getUsers()` function from our previous example: this time, it takes a database instance (`db`) as an argument.

With this setup, we can first obtain a connected database instance (`connectedDb`) and then use it safely across the rest of the application. Ideally, the `getConnectionDb()` function could even be moved into the `db.js` module, since it acts as a common utility for working with the database.

The main disadvantage of this technique is that it requires us to know, ahead of time, which components or functions will depend on the asynchronously initialized resource. This makes the code more fragile and more prone to mistakes. This drawback becomes more tangible when our application has a more complex chain of dependencies. In such cases, we must ensure at every step that the database client is connected, rather than being able to assume we already have a connected client available as a dependency.

For example, suppose our application sends a welcome email when a new user signs up, and that feature internally queries the database to retrieve the user's email address and name. If we forget to pass the already-connected database client to the email-sending function, and that function tries to use the client directly without ensuring the connection has been established, it will fail. This happens because we did not anticipate that this new feature also relied on the database being connected.

One way to address this is to delay the startup of the entire application until all asynchronous services are initialized. This approach is simple and effective, but it can add significant delay to the application's startup time. Additionally, it does not account for cases where an asynchronously initialized component needs to be reinitialized after the initial startup (imagine the case where our database client loses the connection with the database and a new connection needs to be created). Pre-initializing all the resources also assumes that all these resources will always be used, which is often not true in large web applications with many features, different code paths, or CLI applications with multiple subcommands. Depending on how users interact with the application, they might never trigger certain code paths, meaning you would have paid the cost of initializing resources that are ultimately unused.

As we will see in the next section, there is a third alternative that allows us to transparently and efficiently delay operations until the asynchronous initialization step has completed.

Pre-initialization queues

Another approach to ensure that a component's services are invoked only after it is fully initialized is by combining a simple queue with the Command pattern.

The idea is to queue method invocations (specifically, those requiring the component to be initialized) while the component is still not ready and then execute them automatically once the initialization is complete.

Let's apply this technique to our sample db component:

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

class Database {
  connected = false
  #pendingConnection = null
  commandsQueue = []

  async connect() {
    if (!this.connected) {
      if (this.#pendingConnection) {
        return this.#pendingConnection
      }
      // simulate the delay of the connection
      this.#pendingConnection = setTimeout(500)
      await this.#pendingConnection
      this.connected = true
      this.#pendingConnection = null
      // once connected executes all the queued commands
      while (this.commandsQueue.length > 0) { // 2
        const command = this.commandsQueue.shift()
        command()
      }
    }
  }

  async query(queryString) {
    if (!this.connected) { // 1
      console.log(`Request queued: ${queryString}`)

      return new Promise((resolve, reject) => {
        const command = () => {
          this.query(queryString).then(resolve, reject)
        }
        this.commandsQueue.push(command)
      })
    }

    // simulate the delay of the query execution
    await setTimeout(100)
```

```
    console.log(`Query executed: ${queryString}`)
  }
}

export const db = new Database()
```

The technique here consists of two main parts:

1. If the component is not yet initialized (in our case, when `connected` is `false`), we create a command capturing the current invocation's parameters and push it into `commandsQueue`. When the command is later executed, it re-invokes the `query()` method and forwards the result back to the original caller through a promise. Notice how the `query()` method effectively implements two distinct code paths: one for when the component is not yet initialized (where the query is queued) and one for when it is initialized (where the query is executed). The early return inside the `if` statement cleanly separates these two behaviors.
2. Once the initialization is completed (after the database connection is established), we process `commandsQueue` by executing all the queued commands in order.

With this design, consumers of the `db` component no longer need to check whether it is initialized. All the queuing and deferred execution logic is handled internally, and the `db` instance can be used transparently as if it were always ready. Let's see a usage example:

```
import { setTimeout } from 'node:timers/promises'
import { db } from './db.js'

db.connect()

async function updateLastAccess() {
  await db.query(`INSERT (${Date.now()}) INTO "LastAccesses"`)
}

updateLastAccess()
await setTimeout(600)
updateLastAccess()
```

In this example, we are simulating the insertion of a timestamp representing the last access to a given page (imagine it as a simple web analytics table). The only thing we need to do is call `db.connect()`. We do not have to wait for the connection to be established before issuing queries to the database. Depending on the state of the client (whether the connection is established or not), the query will either be queued or executed immediately.

In this case, we are executing two queries: the first one is issued immediately (and will be queued), and the second one is issued after 600 milliseconds (which will be executed immediately, since our simulated connection delay is 500 milliseconds, and by that time the client will already be connected).



Note that with this strategy, if we forget to call `connect()`, any queries we execute will return promises that never resolve, unless `connect()` is eventually called later.

Running this code should produce the following output:

```
Request queued: INSERT (1745693136392) INTO "LastAccesses"  
<pause for ~600ms>  
Query executed: INSERT (1745693136392) INTO "LastAccesses"  
Query executed: INSERT (1745693136998) INTO "LastAccesses"
```

Using the State pattern

We can improve the modularity and reduce the boilerplate of the `Database` class even further by applying the State pattern, which we explored in *Chapter 9, Behavioral Design Patterns*.

In this approach, the component's behavior depends on its internal state, switching automatically between two states:

- **Initialized state:** This state handles all operations assuming the component is ready. Each method directly implements its business logic without worrying about initialization.
- **Queuing state:** This state is active before initialization completes. It implements the same methods, but their sole purpose is to queue the requested operations for later execution.

Let's first implement `InitializedState`, which contains the business logic for the ready state:

```
class InitializedState {  
  constructor(db) {  
    this.db = db  
  }  
  
  async query(queryString) {  
    // simulate the delay of the query execution  
    await setTimeout(100)  
    console.log(`Query executed: ${queryString}`)  
  }  
}
```

As you can see, `query()` simply simulates a query execution by introducing a small delay and then logging the executed query.

Next, we implement `QueuingState`, which handles requests received while the database is not yet connected:

```
const deactivate = Symbol('deactivate')

class QueuingState {
  constructor(db) {
    this.db = db
    this.commandsQueue = []
  }

  async query(queryString) { // 1
    console.log(`Request queued: ${queryString}`)
    return new Promise((resolve, reject) => {
      const command = () => {
        this.db.query(queryString).then(resolve, reject)
      }
      this.commandsQueue.push(command)
    })
  }

  [deactivate]() { // 2
    while (this.commandsQueue.length > 0) {
      const command = this.commandsQueue.shift()
      command()
    }
  }
}
```

There are a few important details to note here:

1. When `query()` is called before the connection is established, it queues a command instead of trying to execute immediately.
2. The `[deactivate]` method (named with a `Symbol` to avoid future naming conflicts and to keep the function private) flushes and executes all queued commands once the database is ready.

Now, let's reimplement the Database class to make use of these two states:

```
class Database {
  connected = false
  #pendingConnection = null

  constructor() {
    this.state = new QueuingState(this) // 1
  }

  async query(queryString) {
    return this.state.query(queryString) // 2
  }

  async connect() {
    if (!this.connected) {
      if (this.#pendingConnection) {
        return this.#pendingConnection
      }
      // simulate the delay of the connection
      this.#pendingConnection = setTimeout(500)
      await this.#pendingConnection
      this.connected = true
      this.#pendingConnection = null
      // once connected update the state
      const oldState = this.state // 3
      this.state = new InitializedState(this)
      oldState[deactivate]?.()
    }
  }
}

export const db = new Database()
```

Let's highlight the key aspects of this new implementation:

1. In the constructor, we initialize the component in `QueuingState` because the asynchronous initialization (the database connection) has not yet completed.
2. The `query()` method simply delegates the call to the current state's `query()` method, without any additional checks.
3. Once the connection is established, we switch the internal state from `QueuingState` to `InitializedState`, and deactivate the old state to execute any queued operations.

By applying the State pattern here, we significantly reduce repetitive code, encapsulate initialization concerns cleanly inside states, and allow the business logic (in `InitializedState`) to remain free from low-level connection checks.

This approach, however, only works when you have control over the component's implementation.

If you cannot modify the component directly, you would need to create an external wrapper or proxy to achieve similar behavior, but the principles and structure would remain much the same.

In the wild

The pattern we just presented is used by many database drivers and ORM libraries. The most notable is Mongoose (nodejsdp.link/mongoose), which is an ORM for **MongoDB**. With Mongoose, it's not necessary to wait for the database connection to open to be able to send queries. This is because each operation is queued and then executed later when the connection with the database is fully established, exactly as we've described in this section. This is clearly a must for any API that wants to provide a good **developer experience (DX)**.

Look at the code of Mongoose to see how every method in the native driver is proxied to add the pre-initialization queue. This also demonstrates an alternative way of implementing the recipe we presented in this section. You can find the relevant code fragment at nodejsdp.link/mongoose-init-queue.

Similarly, the `pg` package (nodejsdp.link/pg), which is a client for the PostgreSQL database, leverages pre-initialization queues, but in a slightly different fashion. `pg` queues every query, regardless of the initialization status of the database, and then immediately tries to execute all the commands in the queue. The relevant code line can be read here: nodejsdp.link/pg-queue.

Asynchronous request batching and caching

In high-load applications, caching is essential. It powers much of the modern web, helping to serve everything from static assets such as web pages, images, and stylesheets to dynamic data such as database query results. In this section, we will dive into how caching strategies apply to asynchronous operations and how, with the right techniques, a surge in requests can become an opportunity rather than a problem. We will explore powerful patterns such as asynchronous request batching, which can help you optimize performance and resource usage even further.

What's asynchronous request batching?

When dealing with asynchronous operations, the most basic level of caching can be achieved by **batching** together a set of invocations to the same API. The idea is very simple: if we invoke an asynchronous function while there is still another one pending, we can piggyback on the already running operation instead of creating a brand-new request.

We can appreciate this idea with a real-life analogy. In an assembly-line factory (for example, a car plant), people from production, maintenance, and procurement often need the same information: whether a spare part is available and how many units are in stock. Person 1 asks at the parts counter, and the clerk calls the nearby warehouse so a colleague can check the shelf and read the current count. Moments later, Person 2 from another department asks about that same part. Instead of placing a second call to the warehouse (which would trigger another run to the shelves), the clerk replies: “A colleague is already checking. Wait a moment and we’ll share the result.” The colleague completes the check once, returns with the number, and both people use that single reading. In short: one trip in progress can serve multiple requests that arrive during it; this keeps things efficient and avoids unnecessary duplicate work.

Request batching in a server works in a similar way: if multiple requests arrive for the same data while the first one is still being processed, they can all “piggyback” on the same operation instead of starting new, identical work.

Let’s take a look at the following diagram:

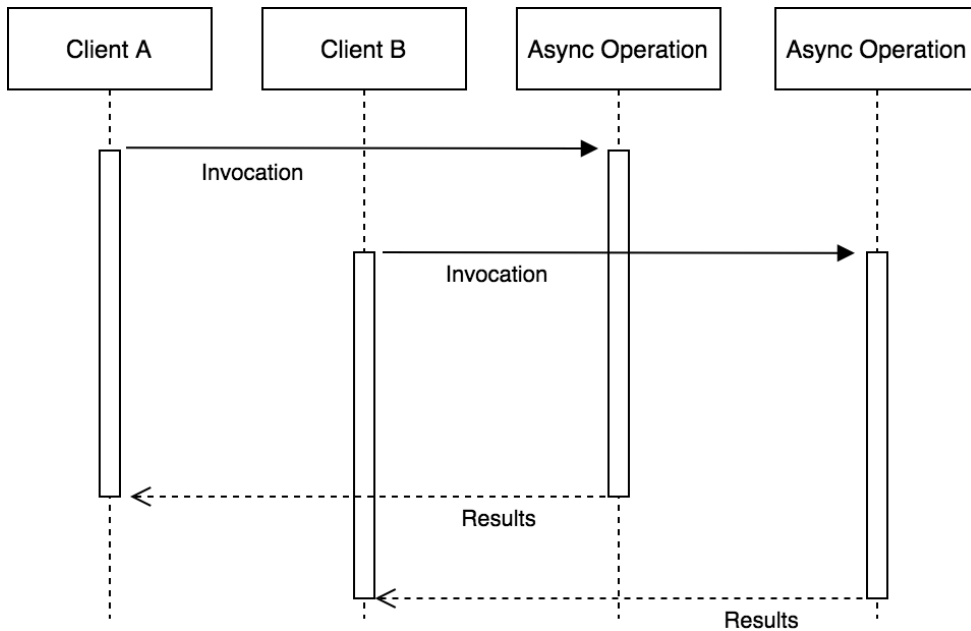


Figure 11.1 – Two asynchronous requests with no batching

The diagram shows two clients invoking the same asynchronous operation with *exactly the same input*. Naturally, without any batching mechanism in place, each client initiates its own separate operation, leading to two independent asynchronous processes that complete at different times.

If the operation is particularly expensive (for example, performing a heavy database query or rendering a complex page), we end up paying the full cost twice, even though the result will be identical for both requests. Doesn't that feel like a waste? And what if the system is under high load, and instead of two concurrent requests, we have dozens or even hundreds? Are you starting to see the optimization opportunity here?

Now, consider the following scenario:

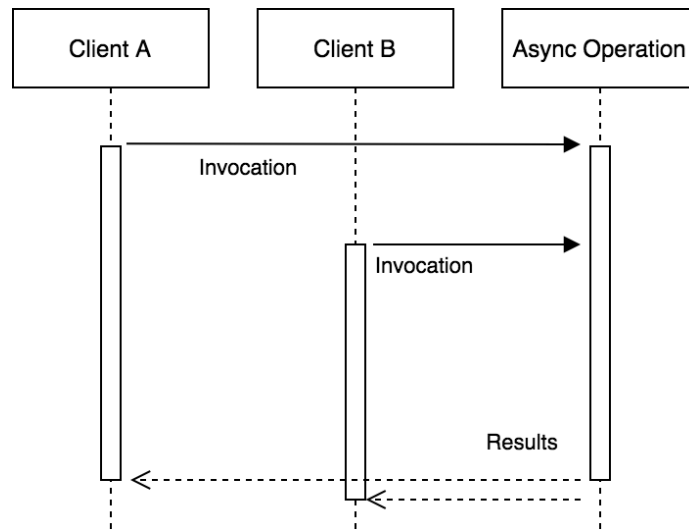


Figure 11.2 – Batching of two asynchronous requests

Figure 11.2 shows how two identical requests (invoking the same API with the same input) can be batched together, meaning they are both attached to the same running operation.

By doing this, when the asynchronous operation completes, both clients are notified, even though the operation itself was executed only once. This represents a simple, yet extremely powerful, way to optimize the load on an application without introducing the complexity of traditional caching mechanisms, which usually require careful memory management and invalidation strategies.

Optimal asynchronous request caching

Request batching becomes less effective when operations are fast enough or when matching requests are spread over a longer period.

Additionally, in many cases, we can safely assume that the result of two identical API invocations will not change frequently. In such situations, simple request batching alone will not deliver the best performance.

When this happens, a more aggressive caching mechanism becomes the best candidate for reducing the load on an application and improving its responsiveness.

The idea is straightforward: as soon as a request completes, we store its result in a cache, which could be an in-memory variable or an entry in a dedicated caching system like Redis. Then, the next time the API is invoked with the same input, the result can be retrieved immediately from the cache, avoiding the need to perform the operation again.

The concept of caching should already be familiar to most experienced developers. However, what makes caching particularly interesting in the context of asynchronous programming is that it can be combined with request batching to be truly optimal.

The reason is simple: while the cache is not yet populated, multiple concurrent requests for the same input can arrive. Without batching, each of these requests would trigger the same operation independently, and once they complete, they would all try to set the cache separately, leading to unnecessary duplicated work.

This is where request batching comes in. Let's see that in action in the following figure:

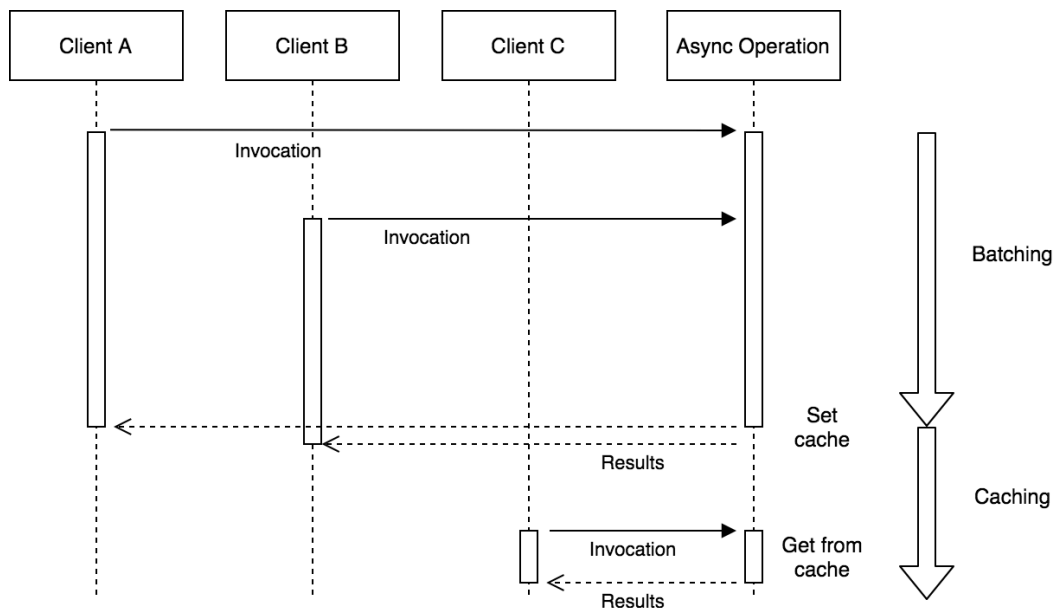


Figure 11.3 – Combined batching and caching

The preceding figure shows the two phases of an optimal asynchronous caching algorithm:

1. The first phase is identical to the batching pattern. Any request received while the cache is not set will be batched together (this is the case for **Client A** and **Client B** in the figure). When the request completes, the cache is set once.
2. When the cache is finally set, any subsequent request will be served directly from the cache. In the figure, this is the case for **Client C**.

Another crucial detail to remember is the *Zalgo* anti-pattern (as discussed in *Chapter 3, Callbacks and Events*). Since we are dealing with asynchronous APIs, we must be sure to always return the cached value asynchronously, even if accessing the cache involves only a synchronous operation, such as in the case in which the cached value is retrieved from an in-memory variable.

An API server without caching or batching

Before we start diving into this new challenge, let's implement a small demo server that we will use as a reference to measure the impact of the various techniques we are going to implement.

Let's consider an API server that manages the sales of an e-commerce company. In particular, we want to query our server for the sum of all the transactions of a particular type of merchandise. For this purpose, we are going to use a Level database through the `level` npm package (nodejsdp.link/level). The data model that we are going to use is a simple list of transactions stored in the sales database, which is organized in the following format:

```
transactionId {amount, product}
```

The key is represented by `transactionId`, and the value is a JSON object that contains the amount of the sale (`amount`) and the product type (`product`).

The data to process is quite basic, so let's implement a simple query over the database that we can use for our experiments. Let's say that we want to get the total amount of sales for a particular product. The routine would look as follows (file `totalSales.js`):

```
import { Level } from 'level'

const db = new Level('sales', { valueEncoding: 'json' })

export async function totalSales(product) {
  const now = Date.now()
  let sum = 0
  for await (const [_transactionId, transaction] of db.iterator()) {
    if (!product || transaction.product === product) {
      sum += transaction.amount
    }
  }
}
```

```
    console.log(`totalSales() took: ${Date.now() - now}ms`)  
  
    return sum  
  }  
}
```

The `totalSales()` function iterates over all the transactions of the sales database and calculates the sum of the amounts of a particular product. The algorithm is intentionally slow as we want to highlight the effect of batching and caching later.



In a real-world application, scanning an entire database table should be avoided. Instead, you should consider using an index (for example, to filter by product) or, even better, implement an event-driven pipeline that continuously calculates the total for each product on a dedicated table as new transactions are recorded.

We can now expose the `totalSales()` API through a simple HTTP server (the `server.js` file):

```
import { createServer } from 'node:http'  
import { totalSales } from './totalSales.js'  
  
createServer(async (req, res) => {  
  const url = new URL(req.url, 'http://localhost')  
  const product = url.searchParams.get('product')  
  console.log(`Processing query: ${url.search}`)  
  
  const sum = await totalSales(product)  
  
  res.setHeader('Content-Type', 'application/json')  
  res.writeHead(200)  
  res.end(  
    JSON.stringify({  
      product,  
      sum,  
    })  
  )  
}).listen(8000, () => console.log('Server started'))
```

To keep the setup lightweight and focused on the essential logic, we chose to use Node.js's built-in `node:http` module instead of a fully-fledged web framework such as Fastify or Express.

When a client sends a request, the server extracts the product query parameter from the URL and logs the incoming query to the console. It then calls the `totalSales()` function, passing the requested product, and waits for the result. Once the sum is calculated, the server responds with a JSON object containing both the product name and the total sales amount.

Before starting the server for the first time, we need to populate the database with some sample data. We can do this using the `populateDb.js` script, which you can find in the book's code repository under the folder dedicated to this section (`nodejsdp.link/batch-cache`). This script generates 100,000 random sales transactions in the database, ensuring that our queries take some time to crunch through the data. You can populate the database by running this:

```
node populateDb.js
```

Once the data is in place, we are ready to start the server:

```
node server.js
```

To query the server, simply open a browser window and navigate to:

```
http://localhost:8000?product=book
```

You should see an output like this:

```
{"product": "book", "sum": 1008193}
```



Keep in mind that your actual sum might be different, since the dataset is generated randomly.

This confirms that our implementation works. But how fast is it?

To better understand the performance of our server, we need more than a single request. For this, we will use **Autocannon** (`nodejsdp.link/autocannon`), an HTTP benchmarking tool written in Node.js, which we can install from npm:

```
npm i -g autocannon
```

Once Autocannon is installed, we can run a benchmark by executing this:

```
autocannon 'http://localhost:8000?product=book'
```

By default, Autocannon creates 10 concurrent connections, each one continuously sending requests, one after another, for about 10 seconds.

When the benchmark is complete, Autocannon prints a detailed report containing key performance metrics for the server under load. These include latency statistics (how long each request takes), request rate (requests per second), and data throughput (bytes per second). It also provides distribution statistics such as averages, percentiles, and maximum values, offering a complete picture of how the server behaves under concurrent access.

On my machine, this test resulted in **150 requests over 10 seconds**, averaging around **15 requests per second**.

This gives us a good baseline to start with. As we experiment with different optimizations, such as request batching and caching, we can run the benchmark again to see whether we are actually improving performance, and by how much.



Benchmarking is tricky because results vary with hardware, network, and load. Run your own tests and compare approaches under similar conditions instead of chasing absolute numbers. The goal is to show that techniques like batching and caching can significantly improve performance, with the exact gains depending on your server, traffic patterns, and cache duration.

Now, we are going to apply our optimizations and measure how much time we can save. We'll start by implementing both batching and caching by leveraging the properties of promises.

Batching and caching with promises

Promises are a great tool for implementing asynchronous batching and caching of requests. Let's see why.

If we recall what we learned about promises in *Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await*, there are two properties that can be exploited to our advantage in this circumstance:

- Multiple `then()` listeners can be attached to the same promise.
- The `then()` listener is guaranteed to be invoked (only once), and it works even if it's attached after the promise is already resolved. Moreover, `then()` is guaranteed to always be invoked asynchronously.

In short, the first property is exactly what we need for batching requests, while the second means that a promise is already a cache for the resolved value and offers a natural mechanism for returning a cached value in a consistent, asynchronous way. In other words, this means that batching and caching become extremely simple and concise with promises.

Batching requests

Let's now add a batching layer on top of our `totalSales` API.

The pattern we will use is very straightforward: if there is already an identical request in progress when the API is invoked, we will simply wait for that request to complete instead of launching a new one. As we will see, this can be implemented easily with promises.

The idea is simple: every time we launch a new request, we save the corresponding promise in a map, associating it with the request parameters (in our case, the product type). Then, for every subsequent request, we first check whether a promise for that product already exists. If it does, we return the existing promise; if not, we create and store a new one.

Now, let's see how this translates into code.

We will create a new module named `totalSalesBatch.js`, where we implement a batching layer on top of the original `totalSales()` API:

```
import { totalSales as totalSalesRaw } from './totalSales.js'

const runningRequests = new Map()

export function totalSales (product) {
  if (runningRequests.has(product)) { // 1
    console.log('Batching')
    return runningRequests.get(product)
  }

  const resultPromise = totalSalesRaw(product) // 2
  runningRequests.set(product, resultPromise)
  resultPromise.finally(() => { // 3
    runningRequests.delete(product)
  })

  return resultPromise
}
```

The `totalSales()` function of the `totalSalesBatch` module is effectively a proxy for the original `totalSales()` API, and it works as follows:

1. If a promise for the given product already exists, we just return it. This is where we *piggyback* on an already-running request.
2. If there is no request running for the given product, we execute the original `totalSales()` function, and we save the resulting promise into the `runningRequests` map.
3. Next, we make sure to remove the same promise from the `runningRequests` map as soon as the request completes.

The behavior of the new `totalSales()` function is identical to that of the original `totalSales()` API, with the difference that, now, multiple calls to the API using the same input are batched, potentially saving us time and resources.

Curious to know what the performance improvement compared to the raw, non-batched version of the `totalSales()` API is? Let's then replace the `totalSales` module used by the HTTP server with the one we just created (the `app.js` file):

```
// import { totalSales } from './totalSales.js'
import { totalSales } from './totalSalesBatch.js'

createServer(async (req, res) => {
  // ...
```

We can now restart the server and rerun our Autocannon benchmark.

On my machine, the server is now able to handle around **1,000 requests in 10 seconds**, averaging about **100 requests per second**.

This is roughly 6.5 times more requests per second compared to our previous implementation using the `totalSales()` API directly with no request batching. Not bad at all!

This result clearly demonstrates the significant performance boost we can achieve by simply adding a batching layer, without the complexity of managing a fully-fledged cache or the challenges of dealing with cache invalidation strategies.



The Request Batching pattern reaches its best potential in high-load applications and with slow APIs. This is because it's exactly in these circumstances that we can batch together a high number of requests.

Let's now see how we can implement both batching and caching using a slight variation of the technique we've just explored.

Batching and caching requests

Adding a caching layer to our batching API is straightforward, thanks to promises. All we must do is leave the promise in our request map, even after the request has completed.

Let's implement the `totalSalesCache.js` module straightaway:

```
import { totalSales as totalSalesRaw } from './totalSales.js'

const CACHE_TTL = 30 * 1000 // 30 seconds TTL
const cache = new Map()

export function totalSales (product) {
  if (cache.has(product)) {
    console.log('Cache hit')
    return cache.get(product)
  }

  const resultPromise = totalSalesRaw(product)
  cache.set(product, resultPromise)
  resultPromise.then(() => {
    setTimeout(() => {
      cache.delete(product)
    }, CACHE_TTL)
  }, err => {
    cache.delete(product)
  })
}
```



```
    throw err
  })

  return resultPromise
}
```

The relevant implementing with” code that enables with promises” caching is highlighted. All we must do is remove the promise from the cache after a certain time (CACHE_TTL) after the request has completed, or immediately if the request has failed. This is a very basic cache invalidation technique, but it works perfectly for our demonstration.

Now, we are ready to try the `totalSales()` caching wrapper we just created. To do that, we only need to update the `app.js` module, as follows:

```
// import { totalSales } from './totalSales.js'
// import { totalSales } from './totalSalesBatch.js'
import { totalSales } from './totalSalesCache.js'

createServer(async (req, res) => {
  // ...
```

Now, we can start the server again and benchmark it with Autocannon.

This time, on my machine, the server can handle around **376,000 requests in about 10 seconds!**

That is more than 350 times the number of requests handled in the batching example, and about 2,500 times more than our original version with no batching. An incredible performance boost for such a simple change!

Of course, these results depend heavily on many factors, such as the number of incoming requests and the delay between them. The advantages of caching over batching become much more substantial when the request volume is high and spread out over a longer period.



In real-world applications, more advanced cache invalidation and storage strategies may be needed:

To limit memory usage when caching many values, use policies like **least recently used (LRU)** or **first in first out (FIFO)**. In distributed deployments, in-memory caches can produce inconsistent results across instances. A shared store like Redis (nodejsdp.link/redis), Valkey (nodejsdp.link/valkey), or Memcached (nodejsdp.link/memcached) keeps data consistent and can be more performant. Manual invalidation (triggered when the underlying data changes) allows longer cache lifetimes while keeping results fresh, but is more complex to manage. As Phil Karlton famously said, “*There are only two hard things in Computer Science: cache invalidation and naming things.*”

Canceling asynchronous operations

Being able to stop a long-running operation is particularly useful if the operation has been canceled by the user or if it has become redundant. In multithreaded programming, we can just terminate the thread, but on a single-threaded platform such as Node.js, things can get a little bit more complicated.



In this section, we will focus on canceling asynchronous operations, not on canceling promises themselves, which is a different topic. It is worth noting that the promises/A+ specification does not define an API for canceling promises. However, if you need this functionality, you can use a third-party promise library such as Bluebird (see more at nodejsdp.link/bluebird-cancellation). Keep in mind that canceling a promise does not automatically cancel the underlying asynchronous operation. In fact, Bluebird offers an `onCancel` callback in its promise constructor, alongside `resolve` and `reject`, which can be used to explicitly cancel the underlying operation when the promise is canceled. And that is exactly what we will explore in this section.

A basic recipe for creating cancelable functions

In asynchronous programming, the basic principle for canceling the execution of a function is very simple: we check whether the operation has been canceled after every asynchronous call, and if that's the case, we prematurely quit the operation. Consider, for example, the following code:

```
import { asyncRoutine } from './asyncRoutine.js'
import { CancelError } from './cancelError.js'

async function cancelable(cancelObj) {
  const resA = await asyncRoutine('A')
  console.log(resA)
  if (cancelObj.cancelRequested) {
    throw new CancelError()
  }

  const resB = await asyncRoutine('B')
  console.log(resB)
  if (cancelObj.cancelRequested) {
    throw new CancelError()
  }

  const resC = await asyncRoutine('C')
  console.log(resC)
}
```

The `cancelable()` function receives, as input, an object (`cancelObj`) containing a single property called `cancelRequested`. In the function, we check the `cancelRequested` property after every asynchronous call, and if that's true, we throw a special `CancelError` exception to interrupt the execution of the function.

The `asyncRoutine()` function is just a demo function that prints a string to the console and returns another string after 100 ms. You will find its full implementation, along with that of `CancelError`, in the code repository for this book (nodejsdp.link/canceling-async-simple).

It's important to note that any code external to the `cancelable()` function will be able to set the `cancelRequested` property only after the `cancelable()` function gives back control to the event loop, which is usually when an asynchronous operation is awaited. This is why it's worth checking the `cancelRequested` property only after the completion of an asynchronous operation and not more often.



You might think that checking a `cancelRequested` flag after every `await` is inelegant and not DRY. That is true, but it reflects a core principle of asynchronous programming: the runtime will not stop an async function arbitrarily, only at `await` points where control returns to the event loop (see *Chapter 1, The Node.js Philosophy* section). JavaScript uses **cooperative multitasking**, where tasks yield control voluntarily (for example, with `await`), unlike **pre-emptive multitasking** in multi-threaded environments, where the OS can interrupt or terminate threads at any time. The key point is that in the async model, we are in control, and with that control comes responsibility: if we want cancellation, our code must regularly check for it and act accordingly.

The following code demonstrates how we can cancel the `cancelable()` function:

```
const cancelObj = { cancelRequested: false }

// schedule to set the cancel flag to true after 100ms
setTimeout(() => {
  cancelObj.cancelRequested = true
}, 100)

try {
  // starts the processing
  await cancelable(cancelObj)
} catch (err) {
  if (err instanceof CancelError) {
    console.log('Function canceled')
  } else {
    console.error(err)
  }
}
```

As we can see, all we must do to cancel the function is set the `cancelObj.cancelRequested` property to `true`. This will cause the function to stop and throw a `CancelError`.

Wrapping asynchronous invocations

Creating and using a basic asynchronous cancelable function is very easy, but there is a lot of boilerplate code involved. In fact, it involves so much extra code that it becomes hard to identify the actual business logic of the function.

We can reduce the boilerplate by including the cancellation logic inside a wrapping function, which we can use to invoke asynchronous routines.

Such a wrapper would look as follows (the `cancelWrapper.js` file):

```
import { CancelError } from './cancelError.js'

export function createCancelWrapper() {
  let cancelRequested = false

  function cancel() {
    cancelRequested = true
  }

  function callIfNotCanceled(func, ...args) {
    if (cancelRequested) {
      return Promise.reject(new CancelError())
    }
    return func(...args)
  }

  return { callIfNotCanceled, cancel }
}
```

The factory returns two functions:

- `callIfNotCanceled()`: A wrapper function that allows us to call an asynchronous function only if the overall asynchronous operation has not been canceled already
- `cancel()`: A function that triggers the cancellation

This setup lets us wrap multiple asynchronous invocations with the same wrapper function and then use a single `cancel()` call to cancel all of them at once.

The `callIfNotCanceled()` function takes, as input, a function to invoke (`func`) and a set of parameters to pass to the function (`args`). The wrapper simply checks whether a cancellation has been requested, and if positive, it will return a promise rejected with a `CancelError` object as the rejection reason; otherwise, it will invoke `func()` with the given arguments (`args`) and return its return value.

Let's now see how our wrapper factory can greatly improve the readability and modularity of our `cancelable()` function:

```
import { asyncRoutine } from './asyncRoutine.js'
import { createCancelWrapper } from './cancelWrapper.js'
import { CancelError } from './cancelError.js'

async function cancelable(callIfNotCanceled) {
  const resA = await callIfNotCanceled(asyncRoutine, 'A')
  console.log(resA)
  const resB = await callIfNotCanceled(asyncRoutine, 'B')
  console.log(resB)
  const resC = await callIfNotCanceled(asyncRoutine, 'C')
  console.log(resC)
}

const { callIfNotCanceled, cancel } = createCancelWrapper()
setTimeout(cancel, 100)

try {
  await cancelable(callIfNotCanceled)
} catch (err) {
  if (err instanceof CancelError) {
    console.log('Function canceled')
  } else {
    console.error(err)
  }
}
```

We can immediately see the benefits of using a wrapper function for implementing our cancellation logic. In fact, the `cancelable()` function is now much more concise and readable.

Cancelable async functions with AbortController

The previous solutions work, but they rely on a custom API, which limits interoperability with third-party code. For example, if you publish a library with a cancellable async function, who provides the cancellation object? Your library or the user's code? If each library defines its own mechanism, interoperability suffers. The solution is to use a *standard*, and that is where `AbortController` comes in (nodejsdp.link/abort-controller). `AbortController` is a standard interface in JavaScript (available in browsers and Node.js) that lets you abort one or more asynchronous operations as needed.



If you have ever canceled a `fetch()` request, you have already used `AbortController`, possibly without even realizing it.

To create an instance of `AbortController`, you simply call:

```
const ac = new AbortController()
```

Note that the `AbortController` class is available in the global scope (no explicit import required).

The `AbortController` exposes two important parts: the *controller* itself (which can trigger cancellation) and the associated *signal* (which is used by asynchronous operations to listen for cancellation requests).

To trigger a cancellation, you call the `abort()` method on the controller instance:

```
ac.abort()
```



When you call `ac.abort()`, you can optionally pass an error object as an argument. If you do not provide an error, a default one is created internally using the `DOMException` class with the property name set to `'AbortError'`.

This sends a signal to all operations that are listening to the associated `AbortSignal` object.

To connect an asynchronous function to the cancellation mechanism, you can pass the `signal` property of the controller (an `AbortSignal` instance) as an argument:

```
someAsyncFunction(ac.signal)
```

Inside the asynchronous function, you can check whether cancellation has been requested in two ways:

One way is by calling `abortSignal.throwIfAborted()`, which immediately throws an error if the signal has been aborted and interrupts the flow:

```
abortSignal.throwIfAborted()
```

This will throw either the custom error you passed when calling `ac.abort()` or a default cancellation error.

Alternatively, you can check the `abortSignal.aborted` property, a Boolean value that tells you whether the operation has been canceled, allowing for more customized handling:

```
if (abortSignal.aborted) {  
  // Handle cancellation manually (throw or return)  
}
```



In addition to the `throwIfAborted()` method and the `aborted` property, an `AbortSignal` also provides an event-driven interface. You can listen for the `'abort'` event by using `abortSignal.addEventListener()`. For example:

```
ac.signal.addEventListener('abort', (event) => {  
  console.log(event.type) // Prints 'abort'  
}, { once: true })
```

This allows you to react to cancellation events in a more flexible or decoupled way. The `{ once: true }` option ensures the listener is automatically removed after the event fires.

It is important to clearly understand the difference between the controller and the signal.

The `AbortController` is the part responsible for triggering the cancellation; it is typically owned by whoever initiates the asynchronous operation and wants the ability to cancel it later.

The `AbortSignal` is the part responsible for receiving and reacting to the cancellation request; it is passed to the asynchronous operation and checked internally to decide how to behave when cancellation occurs.

This separation makes it clear who has the authority to request a cancellation and who needs to listen for it and react accordingly.

Now that we are familiar with the `AbortController` and its API, let's see how we can re-implement our previous example to achieve a more standardized and interoperable approach:

```
import { asyncRoutine } from './asyncRoutine.js'  
  
async function cancelable(abortSignal) {  
  abortSignal.throwIfAborted()  
  const resA = await asyncRoutine('A')  
  console.log(resA)  
  
  abortSignal.throwIfAborted()  
  const resB = await asyncRoutine('B')  
  console.log(resB)  
  
  abortSignal.throwIfAborted()  
  const resC = await asyncRoutine('C')  
  console.log(resC)  
}  
  
const ac = new AbortController()  
setTimeout(() => ac.abort(), 100)
```

```
try {
  await cancelable(ac.signal)
} catch (err) {
  if (err.name === 'AbortError') {
    console.log('Function canceled')
  } else {
    console.error(err)
  }
}
```

This example puts into practice what we have discussed about `AbortController` and `AbortSignal`.

The `cancelable()` function accepts an `AbortSignal` and periodically checks whether a cancellation has been requested by calling `abortSignal.throwIfAborted()` before each asynchronous step.

Meanwhile, an `AbortController` instance (`ac`) is created, and after 100 milliseconds, `ac.abort()` is called to trigger cancellation.

When cancellation is requested, the next call to `abortSignal.throwIfAborted()` throws an exception, which is caught in the `try...catch` block. If the error is an `AbortError`, the program logs that the function was canceled. Otherwise, any other unexpected error is logged separately.



If you want to make the previous code a bit more compact and avoid a bit of repetition, you can create a `callIfNotAborted()` wrapper function. This function would take an `AbortSignal`, another function to call, and a list of arguments. It would first check whether the signal has been aborted (for example, by calling `abortSignal.throwIfAborted()`), and if not, it would invoke the provided function with the given arguments and return its result. This utility would allow us to save a line at each checkpoint, turning something like this:

```
abortSignal.throwIfAborted()
const resA = await asyncRoutine('A')
```

into something like this:

```
const resA = await callIfNotAborted(abortSignal,
  asyncRoutine, 'A')
```

This idea is very similar to the `callIfNotCanceled()` function we implemented earlier, but in this case, it is standardized by using an `AbortSignal` instead of a custom cancellation object.

Now that you have seen how the standard `AbortController` works, this is the approach you should prefer when implementing cancelable asynchronous operations in your own code. Spending some time earlier building our own cancellation mechanism was not wasted effort: it should have given you a deeper understanding of how cancellation works under the hood and why `AbortController` is designed the way it is. With this foundation, you should now be able to use `AbortController` more effectively and correctly in real-world scenarios.

Running CPU-bound tasks

The `totalSales()` API that we implemented in the *Asynchronous request batching and caching* section was (intentionally) expensive in terms of resources and took a few hundred milliseconds to run. Nonetheless, invoking the `totalSales()` function did not affect the ability of the application to process concurrent incoming requests. What we learned about the event loop in *Chapter 1, The Node.js Platform*, should explain this behavior: invoking an asynchronous operation always causes the stack to unwind back to the event loop, leaving it free to handle other requests.

But what happens when we run an expensive synchronous task that takes a long time to complete and that doesn't give back control to the event loop until it has finished? This kind of task is also known as **CPU-bound**, because its main characteristic is that it is heavy on CPU utilization rather than being heavy on I/O operations. Let's work immediately on an example to see how these types of tasks behave in Node.js.

Solving the subset sum problem

To kick off this section, let's choose a computationally expensive problem that's both simple to understand and ideal for experimentation. A perfect candidate is the **subset sum problem**, which asks whether a set (or multiset) of integers contains a non-empty subset whose sum equals zero. For example, given the input `[1, 2, -4, 5, -3]`, valid solutions include `[1, 2, -3]` and `[2, -4, 5, -3]`.

The brute-force approach to solving this problem involves checking every possible combination of subsets—an operation with a cost of $O(2^n)$. That means even a modest input of 20 integers would require checking over a million combinations (specifically, 1,048,576). That's exactly the kind of workload we need to put our ideas to the test.



The subset sum problem isn't just a theoretical curiosity. It plays a crucial role in solving real-world challenges across a variety of fields. In *cryptography*, it has been used to design public-key encryption schemes, taking advantage of its computational complexity to secure sensitive data. In *finance*, it helps optimize investment portfolios by selecting combinations of assets that meet specific return and risk criteria. In *resource allocation* and *project scheduling*, it helps efficiently distribute limited resources across competing tasks. In *bioinformatics*, researchers use it to identify genetic sequences or protein structures that match specific properties. Even in *puzzle games* and *AI-driven problem-solving*, subset sum algorithms are behind the scenes, identifying valid configurations or moves.

For our version of the problem, we'll go one step further: instead of just checking for subsets that sum to zero, we'll look for all subsets whose sum matches a given target value. This generalized variation not only broadens the scope but gives us the perfect sandbox to explore performance bottlenecks, cancellation, and control over long-running computations.

Now, let's work to implement such an algorithm. First, let's create a new module called `subsetSum.js`. We will start by creating a class called `SubsetSum`:

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

export class SubsetSum extends EventEmitter {
  constructor(sum, set) {
    super()
    this.sum = sum
    this.set = set
    this.totalSubsets = 0
  }
  //...
```

The `SubsetSum` class extends `EventEmitter`. This allows us to produce an event every time we find a new subset matching the sum received as input. As we will see, this will give us a lot of flexibility.



An alternative approach would be to use generators and yield a new solution each time one is found. This can be a clean and efficient way to produce results lazily, especially when you're interested in iterating over solutions one at a time rather than handling them through events.

Next, let's see how we can generate all the possible combinations of subsets:

```
_combine(set, subset) {
  for (let i = 0; i < set.length; i++) {
    const newSubset = subset.concat(set[i])
    this._combine(set.slice(i + 1), newSubset)
    this._processSubset(newSubset)
  }
}
```

We will not go into too much detail about the algorithm, but there are two important things to notice:

- The `_combine()` method is completely synchronous. It recursively generates every possible subset without ever giving back control to the event loop.
- Every time a new combination is generated, we provide it to the `_processSubset()` method for further processing.

The `_processSubset()` method is responsible for verifying that the sum of the elements of the given subset is equal to the number we are looking for:

```
_processSubset(subset) {
  console.log('Subset', ++this.totalSubsets, subset)
  const res = subset.reduce((prev, item) => (prev + item), 0)
  if (res === this.sum) {
    this.emit('match', subset)
  }
}
```

The `_processSubset()` method applies a `reduce` operation to the subset to calculate the sum of its elements. Then, it emits an event of the type `match` when the resulting sum is equal to the one we are interested in finding (`this.sum`).

Finally, the `start()` method puts all the preceding pieces together:

```
start() {
  this._combine(this.set, [])
  this.emit('end')
}
```

The `start()` method triggers the generation of all the combinations by invoking `_combine()`, and lastly, emits an end event, signaling that all the combinations were checked and any possible match has already been emitted. This is possible because `_combine()` is synchronous; therefore, the end event is emitted as soon as the function returns, which means that all the combinations have been calculated.

Next, we must expose the algorithm we just created over the network. As always, we can use a simple HTTP server for this task. We want to create an endpoint that accepts requests in the format `/subsetSum?data=<Array>&sum=<Integer>` that invokes the `SubsetSum` algorithm with the given array of integers and sum to match.

Let's implement this simple server in a module named `index.js`:

```
import { createServer } from 'node:http'
import { SubsetSum } from './subsetSum.js'

createServer((req, res) => {
  const url = new URL(req.url, 'http://localhost')
  if (url.pathname !== '/subsetSum') {
    res.writeHead(200)
    return res.end("I'm alive!\n")
  }

  const data = JSON.parse(url.searchParams.get('data'))
  const sum = JSON.parse(url.searchParams.get('sum'))
```

```
res.writeHead(200)
const subsetSum = new SubsetSum(sum, data)
subsetSum.on('match', match => {
  res.cork()
  res.write(`Match: ${JSON.stringify(match)}\n`)
  res.uncork()
})
subsetSum.on('end', () => res.end())
subsetSum.start()
}).listen(8000, () => console.log('Server started'))
```

Thanks to the fact that the `SubsetSum` object returns its results using events, we can stream the matching subsets as soon as they are generated by the algorithm, in real time. Another detail to mention is that our server responds with the text `I'm alive!` every time we hit a URL different from `/subsetSum`. We will use this for checking the responsiveness of our server, as we will see in a moment.



In the code above, we use `res.cork()` and `res.uncork()` to control how data is written to the HTTP response stream. Normally, writable streams in Node.js might buffer small writes internally before actually sending them out. By explicitly calling `res.cork()` before writing and `res.uncork()` immediately after, we make sure that each `match` is flushed to the client as soon as it is available ([nodejsdp.link/cork](https://nodejs.org/docs/latest/api/stream.html#streamrescork)). This trick allows us to see each new match appear in real time, which helps us appreciate how much time can pass between one result and another, especially when dealing with expensive computations where results may not come immediately.

We are now ready to try our subset sum algorithm. Curious to know how our server will handle it? Let's fire it up, then:

```
node index.js
```

As soon as the server starts, we are ready to send our first request. Let's try it with a multiset of 17 random numbers, which will result in the generation of 131,071 combinations, a nice amount to keep our server busy for a while:

```
curl -G http://localhost:8000/subsetSum --data-urlencode "data=[16, 19,1,1,-16,9,1,-5,-2,17,-15,-97,19,-16,-4,-5,15]" --data-urlencode "sum=0"
```

We should see the results coming from the server. But if we try the following command in another terminal while the first request is still running, we will spot a huge problem:

```
curl -G http://localhost:8000
```

We will immediately see that this last request hangs until the subset sum algorithm of the first request has finished: the server is unresponsive! This was expected. The Node.js event loop runs in a single thread, and if this thread is blocked by a long synchronous computation, it will be unable to execute even a single cycle to respond with a simple `I'm alive!`

We quickly understand that this behavior does not work for any kind of application meant to process multiple concurrent requests. But don't despair. In Node.js, we can tackle this type of situation in several ways. So, let's analyze the three most popular methods, which are interleaving with `setImmediate`, using external processes, and using worker threads.



Having an endpoint that can block the event loop for a prolonged period is not just a performance issue; it can also expose your application to **Denial of Service (DoS)** attacks. The goal of a DoS attack is to exhaust a system's resources and make it unavailable to legitimate users, often by exploiting vulnerabilities or flooding it with massive amounts of traffic (as in **DDoS**, or **Distributed Denial of Service** attacks). In the context of Node.js, an attacker could deliberately trigger heavy computations to block the event loop and prevent the server from responding to any other requests. Even without malicious intent, a server under normal high load could experience similar problems if long-running operations are not properly isolated. This is why it is critical to avoid blocking the event loop to build reliable, secure, and scalable Node.js applications.

Interleaving with `setImmediate`

Usually, a CPU-bound algorithm is built upon a set of steps. This can be a set of recursive invocations, a loop, or any variation/combination of these. So, a simple solution to our problem would be to give back the control to the event loop after each of these steps completes (or after a certain number of them). This way, any pending I/O can still be processed by the event loop in those intervals in which the long-running algorithm yields the CPU. A simple way to achieve this is to schedule the next step of the algorithm to run after any pending I/O requests. This sounds like the perfect use case for the `setImmediate()` function (we already introduced this API in *Chapter 3, Callbacks and Events*).

Interleaving the steps of the subset sum algorithm

Let's now see how this technique applies to our subset sum algorithm. All we must do is slightly modify the `subsetSum.js` module. For convenience, we are going to create a new module called `subsetSumDefer.js`, taking the code of the original `subsetSum` class as a starting point.

The first change we are going to make is to add a new method called `_combineInterleaved()`, which is the core of the technique we are implementing:

```
_combineInterleaved(set, subset) {
  this.runningCombine++
  setImmediate(() => {
    this._combine(set, subset)
    if (--this.runningCombine === 0) {
      this.emit('end')
    }
  })
}
```

As we can see, all we had to do was defer the invocation of the original (synchronous) `_combine()` method with `setImmediate()`. However, now, it becomes more difficult to know when the function has finished generating all the combinations, because the algorithm is not synchronous anymore.

To fix this, we must keep track of all the running instances of the `_combine()` method using a pattern very similar to the asynchronous parallel execution flow that we saw in *Chapter 4, Asynchronous Control Flow Patterns with Callbacks*. When all the instances of the `_combine()` method have finished running, we can emit the end event, notifying any listener that the process has completed.

To finish refactoring the subset sum algorithm, we need to make a couple more tweaks. First, we need to replace the recursive step in the `_combine()` method with its deferred counterpart:

```
_combine(set, subset) {
  for (let i = 0; i < set.length; i++) {
    const newSubset = subset.concat(set[i])
    this._combineInterleaved(set.slice(i + 1), newSubset)
    this._processSubset(newSubset)
  }
}
```

With the preceding change, we make sure that each step of the algorithm will be queued in the event loop using `setImmediate()` and, therefore, executed after any pending I/O request instead of being run synchronously.

The other small tweak is in the `start()` method:

```
start () {
  this.runningCombine = 0
  this._combineInterleaved(this.set, [])
}
```

In the preceding code, we initialized the number of running instances of the `_combine()` method to 0. We also replaced the call to `_combine()` with a call to `_combineInterleaved()` and removed the emission of the end event because this is now handled asynchronously in `_combineInterleaved()`.

With this last change, our subset sum algorithm should now be able to run its CPU-bound code in steps interleaved by intervals, where the event loop can run and process any other pending requests.

The last missing bit is updating the `index.js` module so that it can use the new version of the `SubsetSum` API. This is a simple change:

```
import { createServer } from 'node:http'
// import { SubsetSum } from './subsetSum.js'
import { SubsetSum } from './subsetSumDefer.js'

createServer((req, res) => {
  // ...
```

We are now ready to try this new version of the subset sum server. Start the server again and then try to send a request to calculate all the subsets matching a given sum:

```
curl -G http://localhost:8000/subsetSum --data-urlencode "data=[16, 19,1,1, -16,9,1,-5,-2,17,-15,-97,19,-16,-4,-5,15]" --data-urlencode "sum=0"
```

While the request is running, check again whether the server is responsive:

```
curl -G http://localhost:8000
```

Cool! The second request should return almost immediately, even while the SubsetSum task is still running, confirming that our technique is working well.

Considerations on the interleaving approach

As we saw, running a CPU-bound task while preserving the responsiveness of an application is not that complicated; it just requires the use of `setImmediate()` to schedule the next step of an algorithm to run after any pending I/O. However, this is not necessarily the best recipe in terms of efficiency. In fact, deferring a task introduces a small overhead that, multiplied by all the steps that an algorithm has to run, can have a significant impact on the overall running time. This is usually the last thing we want when running a CPU-bound task. A possible solution to mitigate this problem would be using `setImmediate()` only after a certain number of steps—instead of using it at every single step—but still, this would not solve the root of the problem.

Also, this technique doesn't work very well if each step takes a long time to run. In this case, in fact, the event loop would lose responsiveness, and the whole application would start lagging, which is undesirable in a production environment.

Bear in mind that this does not mean that the technique we have just seen should be avoided at all costs. In certain situations, in which the synchronous task is executed sporadically and doesn't take too long to run, using `setImmediate()` to interleave its execution is sometimes the simplest and most effective way to avoid blocking the event loop.



Remember that `process.nextTick()` cannot be used to interleave a long-running task. As we saw in *Chapter 3, Callbacks and Events*, `nextTick()` schedules a task to run before any pending I/O, and this will cause I/O starvation in case of repeated calls. You can verify this yourself by replacing `setImmediate()` with `process.nextTick()` in the previous example.

Using external processes

Deferring the steps of an algorithm is not the only option we have for running CPU-bound tasks. Another pattern for preventing the event loop from blocking is using **child processes**.

We already know that Node.js performs at its best when running I/O-intensive applications, such as web servers, where its asynchronous architecture helps optimize resource utilization. To maintain the responsiveness of an application, the best strategy is to avoid running expensive CPU-bound tasks in the main application context altogether. The goal is to remove heavy computation from the main event loop and move it elsewhere, for example, by offloading it to a separate process, so that the event loop remains free to quickly handle incoming requests. This has three main advantages:

- The synchronous task can run at full speed, without the need to interleave the steps of its execution.
- Working with processes in Node.js is simple, probably easier than modifying an algorithm to use `setImmediate()`, and allows us to easily use multiple processors without the need to scale the main application itself.
- If we really need maximum performance, the external process could be created in lower-level languages, such as good old C, or more modern compiled languages such as Go, Rust, or Zig. Always use the best tool for the job!

Node.js has an ample toolbelt of APIs for interacting with external processes. We can find all we need in the `node:child_process` module. Moreover, when the external process is just another Node.js program, connecting it to the main application is extremely easy and allows seamless communication with the local application. This magic happens thanks to the `child_process.fork()` function, which creates a new child Node.js process and automatically creates a communication channel with it, allowing us to exchange information using an interface very similar to the `EventEmitter`. Let's see how this works by refactoring our subset sum server again.

Delegating the subset sum task to an external process

The goal of refactoring the `SubsetSum` task is to create a separate child process responsible for handling the synchronous processing, leaving the event loop of the main server free to handle requests coming from the network. This is the recipe we are going to follow to make this possible:

1. We will create a new module named `processPool.js` that will allow us to create a pool of running processes. Starting a new process is expensive and requires time, so keeping them constantly running and ready to handle requests allows us to save time and CPU cycles. Also, the pool will help us limit the number of processes running at the same time to prevent exposing the application to DoS attacks.
2. Next, we will create a module called `subsetSumFork.js`, responsible for abstracting a `SubsetSum` task running in a child process. Its role will be communicating with the child process and forwarding the results of the task as if they were coming from the current application.
3. Finally, we need a **worker** (our child process), a new Node.js program with the only goal of running the subset sum algorithm and forwarding its results to the parent process.

Implementing a process pool

Let's start by building the `processPool.js` module piece by piece:

```
import { fork } from 'node:child_process'

export class ProcessPool {
  constructor(file, poolMax) {
    this.file = file
    this.poolMax = poolMax
    this.pool = []
    this.active = []
    this.waiting = []
  }
  //...
```

In the first part of the module, we import the `fork()` function from the `node:child_process` module, which we will use to create new processes. Then, we define the `ProcessPool` constructor, which accepts a `file` parameter representing the Node.js program to run, and the maximum number of running instances in the pool (`poolMax`). We then define three instance variables:

- `pool` is the set of running processes ready to be used.
- `active` contains the list of the processes currently being used.
- `waiting` contains a queue of callbacks for all those requests that could not be fulfilled immediately because of the lack of an available process.

The next piece of the `ProcessPool` class is the `acquire()` method, which is responsible for eventually returning a process ready to be used when one becomes available:

```
acquire() {
  return new Promise((resolve, reject) => {
    let worker
    if (this.pool.length > 0) { // 1
      worker = this.pool.pop()
      this.active.push(worker)
      return resolve(worker)
    }

    if (this.active.length >= this.poolMax) { // 2
      return this.waiting.push({ resolve, reject })
    }

    worker = fork(this.file) // 3
    worker.once('message', message => {
```

```

    if (message === 'ready') {
      this.active.push(worker)
      return resolve(worker)
    }
    worker.kill()
    reject(new Error('Improper process start'))
  })
  worker.once('exit', code => {
    console.log(`Worker exited with code ${code}`)
    this.active = this.active.filter(w => worker !== w)
    this.pool = this.pool.filter(w => worker !== w)
  })
})
}

```

The logic of `acquire()` is explained as follows:

1. If we have a process in the pool ready to be used, we move it to the active list and then use it to fulfill the outer promise with `resolve()`.
2. If there are no available processes in the pool and we have already reached the maximum number of running processes, we must wait for one to be available. We achieve this by queuing the `resolve()` and `reject()` callbacks of the outer promise, for later use.
3. If we haven't reached the maximum number of running processes yet, we create a new one using `child_process.fork()`. Then, we wait for the ready message coming from the newly launched process, which indicates that the process has started and is ready to accept new jobs. This message-based channel is automatically provided with all processes started with `child_process.fork()`.

The last method of the `ProcessPool` class is `release()`, whose purpose is to put a process back into the pool once we are done with it:

```

release(worker) {
  if (this.waiting.length > 0) { // 1
    const { resolve } = this.waiting.shift()
    return resolve(worker)
  }
  this.active = this.active.filter(w => worker !== w) // 2
  this.pool.push(worker)
}

```

This is how the `release()` method works:

1. If there is a request in the waiting list, we simply reassign the worker we are releasing by passing it to the `resolve()` callback at the head of the waiting queue.
2. Otherwise, we remove the worker that we are releasing from the active list and put it back into the pool.

As we can see, the processes are never stopped but just reassigned, allowing us to save time by not restarting them at each request. However, it's important to observe that this might not always be the best choice, and this greatly depends on the requirements of your application.

Other possible tweaks for reducing long-term memory usage and adding resilience to our process pool are:

- Terminate idle processes to free memory after a certain amount of inactivity.
- Add a mechanism to kill non-responsive processes or restart those that have crashed.

In this example, we will keep the implementation of our process pool simple and easy to understand.

Communicating with a child process

Now that our `ProcessPool` class is ready, we can use it to implement the `SubsetSumFork` class, whose role is to communicate with the worker and forward the results it produces. As we already mentioned, starting a process with `child_process.fork()` also gives us a simple message-based communication channel, so let's see how this works by implementing the `subsetSumFork.js` module:

```
import { EventEmitter } from 'node:events'
import { join } from 'node:path'
import { ProcessPool } from './processPool.js'

const workerFile = join(
  import.meta.dirname,
  'workers',
  'subsetSumProcessWorker.js'
)
const workers = new ProcessPool(workerFile, 2)

export class SubsetSum extends EventEmitter {
  constructor(sum, set) {
    super()
    this.sum = sum
    this.set = set
  }

  async start() {
    const worker = await workers.acquire() // 1
```

```

worker.send({ sum: this.sum, set: this.set })

const onMessage = msg => {
  if (msg.event === 'end') { // 3
    worker.removeListener('message', onMessage)
    workers.release(worker)
  }

  this.emit(msg.event, msg.data) // 4
}

worker.on('message', onMessage) // 2
}
}

```

The first thing to note is that we created a new `ProcessPool` object using the file `./workers/subsetSumProcessWorker.js` as the child worker. We also set the maximum capacity of the pool to 2.

Another point worth mentioning is that we tried to maintain the same public API of the original `SubsetSum` class. In fact, `SubsetSumFork` is an `EventEmitter` whose constructor accepts `sum` and `set`, while the `start()` method triggers the execution of the algorithm, which, this time, runs on a separate process. This is what happens when the `start()` method is invoked:

1. We try to acquire a new child process from the pool. When the operation completes, we immediately use the worker handle to send a message to the child process with the data of the job to run. The `send()` API is provided automatically by Node.js to all processes started with `child_process.fork()`. This is essentially the communication channel that we were talking about.
2. We then start listening for any message sent by the worker process using the `on()` method to attach a new listener (this is also a part of the communication channel provided by all processes started with `child_process.fork()`).
3. In the `onMessage` listener, we first check whether we received an end event, which means that the `SubsetSum` task has finished, in which case we remove the `onMessage` listener and release the worker, putting it back into the pool.
4. The worker produces messages in the format `{event, data}`, allowing us to seamlessly forward (re-emit) any event produced by the child process.

That's it for the `SubsetSumFork` wrapper. Let's now implement the worker (our child process).



It is good to know that the `send()` method available on a child process instance can also be used to propagate a socket handle from the main application to a child process (look at the documentation at nodejs.org/docs/latest/api/child_process/#child_process_send_socket). This is the technique used by the `cluster` module to distribute the load of an HTTP server across multiple processes. We will see this in more detail in the next chapter.

Implementing the worker

Let's now create the `workers/subsetSumProcessWorker.js` module, our worker process:

```
import { SubsetSum } from '../subsetSum.js'

process.on('message', msg => { // 1
  const subsetSum = new SubsetSum(msg.sum, msg.set)

  subsetSum.on('match', data => { // 2
    process.send({ event: 'match', data: data })
  })

  subsetSum.on('end', data => {
    process.send({ event: 'end', data: data })
  })

  subsetSum.start()
})

process.send('ready')
```

We can immediately see that we are reusing the original (and synchronous) `SubsetSum` as it is. Now that we are in a separate process, we don't have to worry about blocking the event loop anymore; all the HTTP requests will continue to be handled by the event loop of the main application without disruptions.

When the worker is started as a child process, this is what happens:

1. It immediately starts listening for messages coming from the parent process. This can be easily done with the `process.on()` function (a part of the communication API provided when the process is started with `child_process.fork()`). The only message we expect from the parent process is the one providing the input to a new `SubsetSum` task. As soon as such a message is received, we create a new instance of the `SubsetSum` class and register the listeners for the `match` and `end` events. Lastly, we start the computation with `subsetSum.start()`.
2. Every time the running algorithm finds a solution (`match`), we wrap it in an object having the format `{event, data}` and send it to the parent process. These messages are then handled in the `subsetSumFork.js` module, as we have seen in the previous section.

As we can see, we just had to wrap the algorithm we already built, without modifying its internals. This clearly shows that any portion of an application can be easily put in an external process by simply using the technique we have just seen.



When the child process is not a Node.js program, the simple communication channel we just described (`on()`, `send()`) is not available. In these situations, we can still establish an interface with the child process by implementing our own protocol on top of the standard input and standard output streams, which are exposed to the parent process. To find out more about all the capabilities of the `child_process` API, you can refer to the official Node.js documentation at [nodejsdp.link/child_process](https://nodejs.org/en/docs/api/child-process).

Considerations for the multi-process approach

As always, to try this new version of the subset sum algorithm, we must simply replace the module used by the HTTP server (the `index.js` file):

```
import { createServer } from 'node:http'
// import { SubsetSum } from './subsetSum.js'
// import { SubsetSum } from './subsetSumDefer.js'
import { SubsetSum } from './subsetSumFork.js'

createServer((req, res) => {
  //...
```

We can now start the server again and try to send a sample request:

```
curl -G http://localhost:8000/subsetSum --data-urlencode "data=[16, 19,1,1,-16,9,1,-5,-2,17,-15,-97,19,-16,-4,-5,15]" --data-urlencode "sum=0"
```

As for the interleaving approach that we saw previously, with this new version of the `SubsetSum` module, the event loop is not blocked while running the CPU-bound task. This can be confirmed by sending another concurrent request, as follows:

```
curl -G http://localhost:8000
```

The preceding command should immediately return the text `I'm alive!`.

Even more interestingly, we can try starting two `SubsetSum` tasks at the same time. If your system has more than one processor, you will see that each task runs on a different CPU core, taking full advantage of the hardware.

If we then try to run three `SubsetSum` tasks concurrently, the third task will not start immediately. This is not because the main process's event loop is blocked, but because we have set a concurrency limit of two processes for the `SubsetSum` task. As a result, the third request will be queued and will only start once one of the two running processes becomes available.



Node.js provides a way to get information about the available CPU cores on the current machine. You can do this by using the `node:os` module, and specifically the `cpus()` function, which returns an array of objects describing each logical CPU core. This information can be useful for setting an optimal default concurrency limit whenever you are working on tasks that involve spawning multiple processes, like we are doing with our current `SubsetSum` implementation. For example:

```
import { cpus } from 'node:os'
console.log(cpus().length) // prints the number of logical CPU
cores in the system
```

As we saw, the multi-process approach has many advantages compared to the interleaving approach. First, it doesn't introduce any computational penalty when running the algorithm. Second, it can take full advantage of a multi-processor machine.

Now, let's see an alternative approach that uses threads instead of processes.

Using worker threads

Since Node 10.5.0, we have a new mechanism for running CPU-intensive algorithms outside of the main event loop called **worker threads**. Worker threads can be seen as a lightweight alternative to `child_process.fork()` with some extra goodies. Compared to processes, worker threads have a smaller memory footprint and a faster startup time since they run within the main process but inside different threads.

Worker threads are essentially threads that, by default, don't share anything with the main application thread; they run within their own V8 instance, with an independent Node.js runtime and event loop. Communication with the main thread is possible thanks to message-based communication channels, the transfer of `ArrayBuffer` objects, and the use of `SharedArrayBuffer` objects whose synchronization is managed by the user (usually with the help of `Atomics`).



You can read more about `SharedArrayBuffer` and `Atomics` in the following article: nodejsdp.link/shared-array-buffer. Even though the article focuses on web workers, a lot of concepts are like Node.js's worker threads.

This extensive level of isolation of worker threads from the main thread preserves the integrity of the language. At the same time, the basic communication facilities and data-sharing capabilities are more than enough for 99% of use cases.

Now, let's use worker threads in our `SubsetSum` example.

Running the subset sum task in a worker thread

The worker threads API has a lot in common with that of `ChildProcess`, so the changes to our code will be minimal.

First, we need to create a new class called `ThreadPool`, which is our `ProcessPool` adapted to operate with worker threads instead of processes. The following code shows the differences between the new `ThreadPool` class and the `ProcessPool` class. There are only a few differences in the `acquire()` method, which are highlighted; the rest of the code is identical:

```
import { Worker } from 'node:worker_threads'

export class ThreadPool {
  // ...
  acquire() {
    return new Promise((resolve, reject) => {
      let worker
      if (this.pool.length > 0) {
        worker = this.pool.pop()
        this.active.push(worker)
        return resolve(worker)
      }

      if (this.active.length >= this.poolMax) {
        return this.waiting.push({ resolve, reject })
      }

      worker = new Worker(this.file)
      worker.once('online', () => {
        this.active.push(worker)
        resolve(worker)
      })
      worker.once('exit', code => {
        console.log(`Worker exited with code ${code}`)
        this.active = this.active.filter(w => worker !== w)
        this.pool = this.pool.filter(w => worker !== w)
      })
    })
  }
  // ...
}
```


Next, we need to adapt the worker and place it in a new file called `subsetSumThreadWorker.js`. The main difference from our old worker is that instead of using `process.send()` and `process.on()`, we'll have to use `parentPort.postMessage()` and `parentPort.on()`:

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

parentPort.on('message', msg => {
  const subsetSum = new SubsetSum(msg.sum, msg.set)

  subsetSum.on('match', data => {
    parentPort.postMessage({ event: 'match', data: data })
  })

  subsetSum.on('end', data => {
    parentPort.postMessage({ event: 'end', data: data })
  })

  subsetSum.start()
})
```

Similarly, the `subsetSumThreads.js` module is essentially the same as the `subsetSumFork.js` module except for a couple of lines of code, which are highlighted in the following code fragment:

```
import { EventEmitter } from 'node:events'
import { join } from 'node:path'
import { ThreadPool } from './threadPool.js'

const workerFile = join(
  import.meta.dirname,
  'workers',
  'subsetSumThreadWorker.js'
)

const workers = new ThreadPool(workerFile, 2)

export class SubsetSum extends EventEmitter {
  constructor(sum, set) {
    super()
    this.sum = sum
    this.set = set
  }

  async start() {
```

```

const worker = await workers.acquire()
worker.postMessage({ sum: this.sum, set: this.set })

const onMessage = msg => {
  if (msg.event === 'end') {
    worker.removeListener('message', onMessage)
    workers.release(worker)
  }

  this.emit(msg.event, msg.data)
}

worker.on('message', onMessage)
}
}

```

As we've seen, adapting an existing application to use worker threads instead of forked processes is a trivial operation. This is because the API of the two components are very similar, but also because a worker thread has a lot in common with a fully-fledged Node.js process.

Finally, we need to update the `index.js` module so that it can use the new `subsetSumThreads.js` module, as we've seen for the other implementations of the algorithm:

```

import { createServer } from 'node:http'
// import { SubsetSum } from './subsetSum.js'
// import { SubsetSum } from './subsetSumDefer.js'
// import { SubsetSum } from './subsetSumFork.js'
import { SubsetSum } from './subsetSumThreads.js'

createServer((req, res) => {
  // ...

```

Now, you can try the new version of the subset sum server using worker threads. As for the previous two implementations, the event loop of the main application is not blocked by the subset sum algorithm, as it runs in a separate thread.



The example we've seen uses only a small subset of all the capabilities offered by worker threads. For more advanced topics, such as transferring `ArrayBuffer` objects or `SharedArrayBuffer` objects, you can read the official API documentation at [nodejsdp.link/worker-threads](https://nodejs.org/api/worker_threads).

Running CPU-bound tasks in production

The examples we've seen so far should give you an idea of the tools at our disposal for running CPU-intensive operations in Node.js. However, components such as process pools and thread pools are complex pieces of machinery that require proper mechanisms to deal with timeouts, errors, and other types of failures, which, for brevity, we left out of our implementation. Therefore, unless you have special requirements, it's better to rely on more battle-tested libraries for production use. Two of those libraries are `workerpool` (nodejsdp.link/workerpool) and `piscina` (nodejsdp.link/piscina), which are based on the same concepts we've seen in this section. They allow us to coordinate the execution of CPU-intensive tasks using external processes or worker threads.

One last observation is that we must consider that if we have particularly complex algorithms to run or if the number of CPU-bound tasks exceeds the capacity of a single node, we may have to think about scaling out the computation across multiple nodes. This is a completely different problem, and we'll learn more about this in the next two chapters.

Summary

This chapter added some great new tools to our belt, and as you can see, our journey is getting more focused on advanced problems. Due to this, we have started to delve deeply into more complex solutions. This chapter gave us not only a set of recipes to reuse and customize for our needs but also some great demonstrations of how mastering a few principles and patterns can help us tackle the most complex problems in Node.js development.

The next two chapters represent the peak of our journey. After studying the various tactics of Node.js development, we are now ready to move on to the strategies and explore the architectural patterns for scaling and distributing our Node.js applications.

Exercises

- **11.1 Proxy with pre-initialization queues:** Using a JavaScript Proxy, create a wrapper for adding pre-initialization queues to any object. You should allow the consumer of the wrapper to decide which methods to augment and the name of the property/event that indicates if the component is initialized.
- **11.2 Batching and caching with callbacks:** Implement batching and caching for the `totalSales` API examples using only callbacks, streams, and events (without using promises or `async/await`). Hint: Pay attention to `Zalgo` when returning cached values!
- **11.3 Deep async cancelable:** Extend the `createAsyncCancelable()` function so that it's possible to invoke other cancelable functions from within the main cancelable function. Canceling the main operation should also cancel all nested operations. Hint: Allow to yield the result of an `asyncCancelable()` from within the generator function.

- **11.4 Compute farm:** Create an HTTP server with a POST endpoint that receives, as input, the code of a function (as a string) and an array of arguments, executes the function with the given arguments in a worker thread or in a separate process, and returns the result back to the client. Hint: You can use `eval()`, `vm.runInContext()`, or neither of the two. Note: Whatever code you produce for this exercise, please be aware that allowing users to run arbitrary code in a production setting can pose serious security risks, and you should never do it unless you know exactly what the implications are.

12

Scalability and Architectural Patterns

Node.js began as a small, non-blocking web server built with JavaScript and C++, but its creation was driven by a bigger goal: **scalability**. Ryan Dahl wanted a platform that could handle many concurrent users without exhausting server resources. The idea came from trying to implement a file upload progress bar in early 2009 (nodejsdp.link/first-commit), which was difficult with the tools available. Existing platforms like Ruby on Rails struggled under high concurrency, and experiments with C, Haskell, and Python each came with trade-offs. Then JavaScript, powered by Google's new V8 engine, offered a fresh start for server-side development. Ryan built a platform where non-blocking, asynchronous behavior was the default, using a single thread and event loop to manage thousands of connections efficiently. From there, Node.js grew into a complete platform for building fast, event-driven applications, with scalability as a core design principle. But just because Node.js can scale doesn't mean every app will. Achieving performance and reliability requires understanding the platform, making thoughtful design choices, and following best practices.

This chapter takes a higher-level view of scalability: not just handling more users, but building systems that are reliable, maintainable, and resilient under load. We will cover:

- Why scalability should be part of your design process from the very beginning
- What the scale cube is, and how it helps frame different strategies for scaling
- How to scale a Node.js application by running multiple instances
- How to use load balancers to distribute traffic efficiently
- What a service registry is, and why it matters in dynamic environments
- How to scale using container orchestration tools such as Kubernetes
- How to break down a monolithic application into microservices
- How to connect many services using simple and proven architectural patterns

Understanding these concepts is not just about building better apps. It is about leveling up as a developer. Let's dive in.

An introduction to application scaling

Scalability is the ability of a system to grow and adapt to changing conditions. It is not only about technical capacity but also about supporting the growth of the business and the organization behind it. If you are building a product expected to reach millions of users, you will face questions like: How will the system handle increasing demand without slowing down or crashing? How will it store large volumes of data and manage I/O efficiently? As your team grows, how will you organize work so different parts of the codebase can evolve independently? Even smaller projects face scalability challenges, just in different forms, and ignoring them can harm both the project and the company.

Of course, you do not need to optimize everything up front. The pragmatic path is to understand the available technical options and their trade-offs so you can decide when they are worth applying. Pragmatism is only possible with a broad knowledge of those technical options and a clear view of your business goals.

In this chapter, we will explore patterns and architectures for scaling Node.js applications. With these tools and an understanding of your business context, you can design systems that adapt, perform reliably, and keep users satisfied.

Scaling Node.js applications

Most of the work in a Node.js application runs on a single thread. As discussed in *Chapter 1, The Node.js Platform*, this is not a limitation but a strength. With its non-blocking I/O model, Node.js can handle hundreds or thousands of concurrent I/O-heavy requests per second, even on modest hardware.

Still, a single thread has limits. To handle higher traffic or demanding workloads, you need to scale by running multiple processes and, eventually, distributing them across machines. Scaling not only increases capacity but also improves **reliability** and **fault tolerance**: if one instance fails, others can take over.

Scalability also applies to application structure. As systems grow in features, teams, and responsibilities, designing them as distributed, modular components keeps them maintainable and adaptable. JavaScript's flexibility encourages small, focused modules, and with discipline and tools like TypeScript, its quirks can be turned into strengths.

Mastering when and how to scale Node.js applications is key to building reliable, high-performance systems. In the next sections, we will explore the patterns, tools, and principles that make this possible.

The three dimensions of scalability

When we talk about scalability, the first fundamental principle to understand is **load distribution**. This refers to the practice of splitting the workload of an application across multiple processes and machines. There are many ways to achieve this, and the book, *The Art of Scalability*, by Martin L. Abbott and Michael T. Fisher, introduces a helpful model to categorize them.

This model is called the **scale cube**, and it describes scalability across three distinct dimensions:

- X-axis: Cloning
- Y-axis: Decomposing by service/functionality
- Z-axis: Splitting by data partition

These three dimensions can be represented as a cube, as shown in *Figure 12.1*:

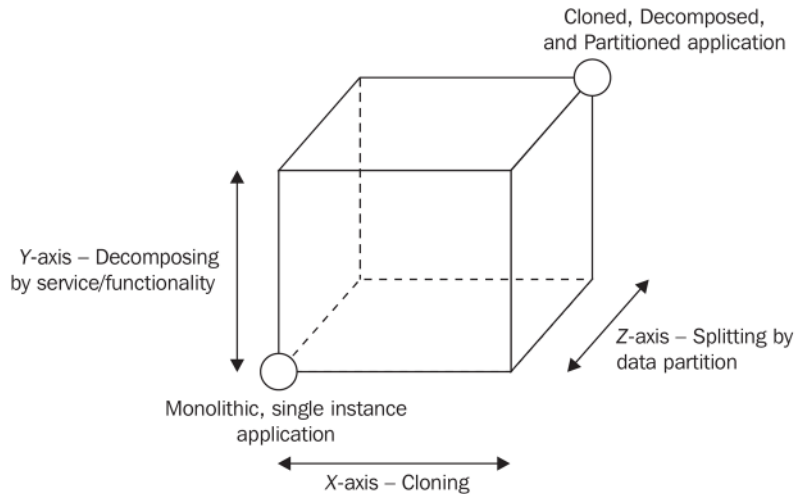


Figure 12.1: The scale cube

The bottom-left corner of the cube represents a typical **monolithic application**. This is an application that contains all its functionality in a single codebase and runs as a single instance. This setup is common for applications in early development stages or those with small workloads.

From this starting point, there are three primary strategies for scaling. These are visualized as movement along the cube's three axes: X, Y, and Z.

X-axis: cloning

Scaling along the X-axis is the most straightforward evolution of a monolithic application. It involves duplicating the same application and running it in multiple instances. Each instance handles a portion of the total workload. This approach is usually simple to implement, cost-effective in terms of development effort, and highly effective in practice. For example, cloning the application across four instances allows each one to manage roughly one-quarter of the total traffic.

Y-axis: decomposing by service or functionality

Scaling along the Y-axis means *decomposing* the application into separate services based on functionality or use case. Each service becomes an independent application with its own codebase, and it may also have its own database and user interface. For example, you might separate the administrative dashboard from the public-facing part of your product, or isolate user authentication into a dedicated service. The way you divide functionality depends on factors such as business requirements, data boundaries, and user needs.

This form of scaling has a significant impact, not just on architecture, but also on how the system is developed, deployed, and maintained. This is where the concept of **microservices** usually comes into play, as it is closely tied to fine-grained Y-axis decomposition.

Z-axis: splitting by data partition

The Z-axis introduces a more advanced scaling strategy. Here, the application is designed so that each instance is responsible for only a portion of the total data. This technique, often called **data partitioning**, is commonly used at the database level but can also be applied at the application level in specific scenarios.

For example, you could divide users based on country (*list partitioning*), assign partitions by the starting letter of their name (*range partitioning*), or use a hash function to determine which partition a user belongs to (*hash partitioning*). Each application instance would then operate only on its assigned partition.

This strategy requires a lookup mechanism to determine which instance is responsible for any given piece of data. While powerful, Z-axis scaling is typically reserved for highly distributed systems or special cases, such as when building custom data persistence layers or working with databases that lack native partitioning support. It is also used in systems operating at a massive scale, such as those found at large tech companies.

Due to its complexity, Z-axis scaling should usually be considered only after the opportunities of the X and Y axes have been fully explored.

Combining strategies across the scale cube

The three dimensions of the Scale Cube are not mutually exclusive. Many real-world systems scale along all of them over time. The goal is not to pick the “best” axis but to find the right balance for your stage of growth, current needs, and constraints. Scaling is an ongoing process. A common journey starts with a monolithic architecture in the early stages, when speed and simplicity help achieve **product-market fit**. As usage grows, you can scale horizontally along the X-axis by running multiple instances of the same application, increasing capacity with minimal complexity. When team size and product scope create coordination bottlenecks, scaling along the Y-axis by decomposing into independent services allows teams to work in parallel and reduces the complexity of each service. If large data volumes become a challenge, the Z-axis, partitioning data so different instances handle different portions, can address performance or storage limits.

Scaling in any dimension is a spectrum. You might lightly decompose into a few services or adopt dozens of microservices, shard only some data or fully partition it. The right approach depends on your context, your team's capabilities, and the trade-offs involved.

In the next sections, we will focus on the two most common techniques for scaling Node.js applications: **cloning** and **decomposing** by functionality or service.

Cloning and load balancing

Before we dive into specific techniques, it's important to introduce the concepts of **vertical scaling** and **horizontal scaling**.

Vertical scaling means upgrading the hardware that runs your application. This could involve adding more memory, increasing CPU capacity, or improving disk performance. In traditional systems, this is often the first approach, because it allows the application to stay exactly as it is while relying on more powerful machines to handle the increased load. It's like trying to move more goods with a single vehicle. You might start with a small station wagon, then upgrade to a van, and eventually use a full-size truck. But there's only so far you can go. Sooner or later, your vehicle becomes too expensive, too heavy, or too large to manage efficiently. That's when adding more vehicles becomes the better option.

This is where horizontal scaling comes in. Instead of pushing a single machine to its limits, you run multiple instances of your application, either on the same machine using separate processes or across multiple machines. This is the approach we referred to earlier when discussing cloning and scaling along the X-axis of the scale cube. It's like switching from one giant truck to a fleet of vehicles, each handling a portion of the load.

In traditional, multithreaded web servers, horizontal scaling is typically introduced only after vertical scaling has been maxed out or becomes too expensive. These servers are designed to take full advantage of all available cores and memory by running multiple threads.

Node.js takes a different approach to concurrency. Because the code runs on a single thread by default, a single process will typically use only one CPU core. This means that, out of the box, you might not fully take advantage of all the power available in a modern multi-core server.



While the JavaScript code you write in Node.js runs on a single thread (the *main thread*), the node process itself uses multiple threads behind the scenes. For example, Node.js creates a thread pool (managed by libuv) for tasks such as filesystem operations, compression, and cryptography, along with threads for the garbage collector and the event loop. When network I/O is involved, additional threads are used for DNS lookups. This internal multithreading is transparent to most developers but is key to Node.js's ability to handle many types of operations efficiently.

We could see this as a limitation, or we could take it as a gentle push to start thinking about scalability from the very beginning. Since Node.js encourages scaling through multiple processes, developers naturally adopt patterns that promote **resilience**, **redundancy**, and modular architecture early in the development process. For example, you can clone your application across several processes or containers, not necessarily because you've hit a performance ceiling, but to increase availability and fault tolerance. If one process fails, others can continue handling requests without interruption.

This intrinsic characteristic of Node.js also encourages better habits. If you design your application to run in multiple instances from the start, you naturally avoid relying on local resources like memory or disk storage that cannot be shared across instances. For example, storing session data in memory might work fine when your application is running as a single instance, but once the app is cloned or deployed across multiple processes or machines, this approach breaks down. Since memory is not shared between processes or machines, a request might land on one instance where the user is authenticated, but the next request could hit a different instance that has no knowledge of the user's session. As a result, the user may appear to be logged out unexpectedly.

With this in mind, let's now look at the most basic mechanism for scaling Node.js applications across multiple processes: the `node:cluster` module.

The cluster module

In Node.js, the simplest pattern to distribute the load of an application across multiple processes running on a single machine is by using the `node:cluster` module, which is part of the core libraries. These processes are often referred to as *instances* of the application, and they are all spawned from the same codebase. The `node:cluster` module makes it easy to fork these processes and automatically balances incoming connections across them, as shown in *Figure 12.2*:

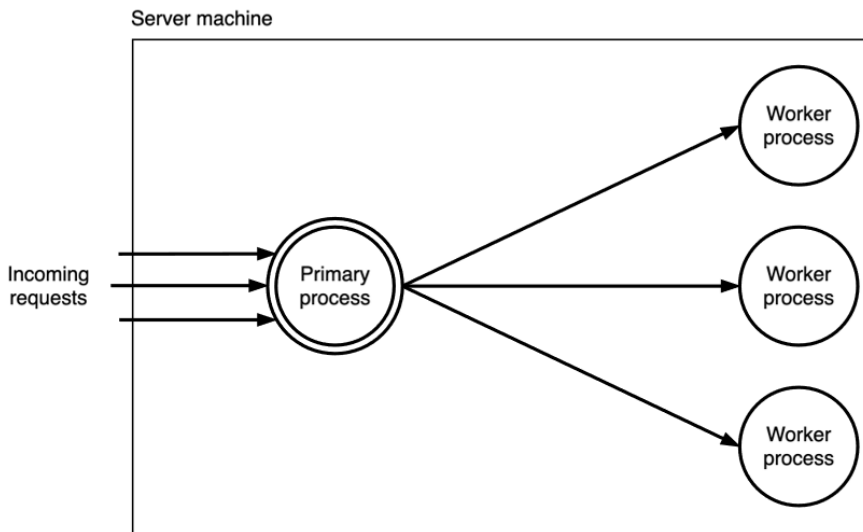


Figure 12.2: Cluster module schematic

The **primary process** is responsible for spawning a number of processes (**workers**), each representing an instance of the application we want to scale. Each incoming connection is then distributed across the cloned workers, spreading the load across them.

Since every worker is an independent process, you can use this approach to spawn as many workers as the number of CPUs available in the system. With this approach, you can easily allow a Node.js application to take advantage of all the computing power available in the system.

Notes on the behavior of the cluster module

In most systems, the `node:cluster` module uses an explicit round-robin load-balancing algorithm. This algorithm is used inside the primary process, which makes sure the requests are evenly distributed across all the workers. At the time of writing, round-robin scheduling is enabled by default on all platforms except Windows, and it can be globally modified by setting the variable `cluster.schedulingPolicy` and using the constants `cluster.SCHED_RR` (round robin) or `cluster.SCHED_NONE` (handled by the operating system).



The round-robin algorithm distributes the load evenly across the available instances on a rotational basis. The first request is forwarded to the first instance, the second to the next instance in the list, and so on. When the end of the list is reached, the iteration starts again from the beginning. In the `node:cluster` module, the round-robin logic is a little bit *smarter* than the traditional implementation. In fact, it is enriched with some extra behaviors that aim to avoid overloading a given worker process.

When we use the `node:cluster` module, every invocation to `server.listen()` in a worker process is delegated to the primary process. This allows the primary process to receive all the incoming messages and distribute them to the pool of workers. The `cluster` module makes this delegation process very simple for most use cases, but there are several edge cases in which calling `server.listen()` in a worker module might not do what you expect:

- `server.listen({fd})`: If a worker listens using a specific file descriptor, for instance, by invoking `server.listen({fd: 17})`, this operation might produce unexpected results. File descriptors are mapped at the process level, so if a worker process maps a file descriptor, this won't match the same file in the primary process. One way to overcome this limitation is to create the file descriptor in the primary process and then pass it to the worker process. This way, the worker process can invoke `server.listen()` using a descriptor that is known to the primary.
- `server.listen(handle)`: Listening using handle objects (`FileHandle`) explicitly in a worker process will cause the worker to use the supplied handle directly, rather than delegating the operation to the primary process.
- `server.listen(0)`: Calling `server.listen(0)` will generally cause servers to listen on a random port. However, in a cluster, each worker will receive the same “random” port each time they call `server.listen(0)`. In other words, the port is random only the first time; it will be fixed from the second call on. If you want every worker to listen on a different random port, you have to generate the port numbers yourself.

Building a simple HTTP server

Let's now start working on an example. Let's build a small HTTP server, cloned and load-balanced using the `node:cluster` module. First, we need an application to scale, and for this example, we don't need too much, just a very basic HTTP server.

So, let's create a file called `app.js` containing the following code:

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

const server = createServer((_req, res) => {
  // simulates CPU intensive work
  let i = 1e7
  while (i > 0) {
    i--
  }

  console.log(`Handling request from ${process.pid}`)
  res.end(`Hello from ${process.pid}\n`)
})

server.listen(8080, () => console.log(`Started at ${process.pid}`))
```

The HTTP server we just built responds to any request by sending back a message containing its **process identifier (PID)**; this is useful for identifying which instance of the application is handling the request. In this version of the application, we have only one process, so the PID that you see in the responses and the logs will always be the same.

Also, to simulate some actual CPU work, we perform an empty loop 10 million times: without this, the server load would be almost insignificant, and it would be quite hard to draw conclusions from the benchmarks we are going to run.



The app module we create here is just a simple abstraction for a generic web server. We are not using a web framework such as Express or Fastify for simplicity, but feel free to rewrite these examples using your web framework of choice.

You can now check if all works as expected by running the application as usual and sending a request to `http://localhost:8080` using either a browser or `curl`.

You can also try to measure the requests per second that the server is able to handle on one process. For this purpose, you can use `autocannon` to load the server with 10 concurrent connections for 10 seconds. As a reference, the result we got on our machine is in the order of 300 requests per second.



The load tests in this chapter are simplified for learning purposes and are not exact performance measures. For production applications, run your own benchmarks after each change to see which techniques work best for your specific case.

Now that we have a simple test web application and some reference benchmarks, we are ready to try some techniques to improve the performance of the application.

Scaling with the cluster module

Let's now update `app.js` to scale our application using the `node:cluster` module:

```
import { createServer } from 'node:http'
import { cpus } from 'node:os'
import cluster from 'node:cluster'

if (cluster.isPrimary) { // 1
  const availableCpus = cpus()
  console.log(`Clustering to ${availableCpus.length} processes`)
  for (const _ of availableCpus) {
    cluster.fork()
  }
} else { // 2
  const server = createServer((_req, res) => {
    // simulates CPU intensive work
    let i = 1e7
    while (i > 0) {
      i--
    }

    console.log(`Handling request from ${process.pid}`)
    res.end(`Hello from ${process.pid}\n`)
  })

  server.listen(8080, () => console.log(`Started at ${process.pid}`))
}
```

As we can see, using the `cluster` module requires very little effort. Let's analyze what is happening:

1. When we launch `app.js` from the command line, we are executing the primary process. In this case, the `cluster.isPrimary` variable is set to `true`, and the only work we are required to do is forking the current process using `cluster.fork()`. In the preceding example, we are starting as many workers as there are logical CPU cores in the system to take advantage of all the available processing power.
2. When `cluster.fork()` is executed from the master process, the current module (`app.js`) is run again, but this time in worker mode (`cluster.isWorker` is set to `true`, while `cluster.isPrimary` is `false`). When the application runs as a worker, it can start doing some actual work. In this case, it starts a new HTTP server.



It's important to remember that each worker is a different Node.js process with its own event loop, memory space, and loaded modules.

It's interesting to note that the usage of the `node:cluster` module is based on a recurring pattern, which makes it very easy to run multiple instances of an application:

```
if (cluster.isPrimary) {  
  // fork()  
} else {  
  // do work  
}
```



Under the hood, the `cluster.fork()` function uses the `child_process.fork()` API; therefore, we also have a communication channel available between the master and the workers. The worker processes can be accessed from the variable `cluster.workers`, so broadcasting a message to all of them would be as easy as running the following line of code:

```
Object.values(cluster.workers).forEach(worker =>  
  worker.send('Hello from the primary'))
```

Now, let's try to run our HTTP server in cluster mode. If our machine has more than one core, we should see several workers being started by the master process, one after the other. For example, in a system with four logical cores, the terminal should look like this:

```
Started 14107  
Started 14099  
Started 14102  
Started 14101
```

If we now try to hit our server again using the URL `http://localhost:8080`, we should notice that each request will return a message with a different PID, which means that these requests have been handled by different workers, confirming that the load is being distributed among them.

Now, we can try to load test our server again with `autocannon` to discover the performance increase obtained by scaling our application across multiple processes. As a reference, in our machine, which has 8 cores, we saw a performance increase of about 5.3x (1,600 req/sec versus 300 req/sec), an impressive gain for such a simple change!

Resiliency and availability with the cluster module

Because workers are all separate processes, they can be killed or respawned depending on a program's needs, without affecting other workers. As long as there are some workers still alive, the server will continue to accept connections. If no workers are alive, existing connections will be dropped, and new connections will be refused. Node.js does not automatically manage the number of workers; it is the application's responsibility to manage the worker pool based on its own needs.

As we already mentioned, this way of scaling an application also brings other advantages, in particular, the ability to maintain a certain level of service, even in the presence of malfunctions or crashes. This property is also known as **resiliency**, and it contributes to the availability of a system.

By starting multiple instances of the same application, we are creating a redundant system, which means that if one instance goes down for whatever reason, we still have other instances ready to serve requests. This pattern is straightforward to implement using the `node:cluster` module. Let's see how it works!

Let's take the code from the previous section as a starting point. In particular, let's modify the `app.js` module so that it crashes after a random interval of time:

```
// ...
} else {
  // Inside our worker block
  setInterval(
    () => {
      if (Math.random() < 0.5) {
        throw new Error(`Oops... ${process.pid} crashed!`)
      }
    },
    Math.ceil(Math.random() * 3) * 1000
  )
  // ...
```

With this change in place, every few seconds, each worker checks whether it should crash. Sometimes it does, sometimes it doesn't. The result is unpredictable, a bit chaotic, and surprisingly close to what might happen in a real production system. Failures often occur without warning, triggered by edge cases, resource limits, or bugs that only show up under certain conditions.

Can you see how this could lead to problems? If no one is watching these processes, and nothing is restarting them, the entire application will eventually stop working. You can see this in action by simply starting this version of the server and waiting a few seconds.

If we were running only one instance of the application, the situation would be significantly worse. A crash would mean a complete stop, with no process left to handle incoming requests. Unless an external monitoring tool is in place to restart the application and does so quickly, users would experience downtime. Even in the best case, there would be a gap between the failure and the recovery, during which requests might be delayed or entirely lost. That kind of disruption is far from ideal, especially for users who expect a fast and reliable experience.

With multiple instances in place, though, we introduce resilience. When one worker goes down, the others continue serving requests without interruption. This simple setup helps us build a system that is better prepared to handle the uncertainty and messiness of real-world failures.

By using the `node:cluster` module to spawn multiple processes, we are already taking a solid step toward building a more resilient web server. The next improvement is to ensure that whenever a worker process exits with an error, it is immediately replaced. This way, the system can recover from accidental process crashes without manual intervention. Let's update `app.js` to watch for worker exits and automatically spawn a replacement when needed:

```
// ...
if (cluster.isPrimary) {
  // ...
  cluster.on('exit', (worker, code) => {
    if (code !== 0 && !worker.exitedAfterDisconnect) {
      console.log(
        `Worker ${worker.process.pid} crashed. Starting a new worker`
      )
      cluster.fork()
    }
  })
} else {
  // ...
}
```

In the updated code, whenever the primary process receives an `'exit'` event from a worker, it checks whether the termination was intentional or the result of a crash. This is determined by inspecting the exit code and the `worker.exitedAfterDisconnect` flag, which indicates whether the worker was explicitly disconnected by the primary process. If the termination was due to an error, a new worker is immediately started to replace it.

What's important here is that while the crashed worker is being replaced, the remaining workers continue to handle incoming requests. This helps maintain the application's availability, even as individual processes fail and recover.

To verify this behavior, we can run a stress test using a tool such as autocannon. When the test completes, one of the key metrics to examine is the number of failed requests. Here's an example of what the output might look like:

```
[...]  
11k requests in 10.02s, 1.47 MB read  
29 errors (0 timeouts)
```

This result shows that the application successfully responded to most requests, despite continuous crashes. Roughly 99.7% of the requests completed successfully, which is a strong indication that our solution is holding up under pressure.

Of course, results will vary depending on how many worker processes are running and how often they crash during the test. Still, this gives us a realistic sense of how the system behaves in a failure-prone environment.

In this example, most of the failed requests are caused by crashes interrupting existing connections (in-flight requests). Unfortunately, there's little we can do to avoid those once a process terminates abruptly. But the important takeaway is that the application recovers automatically and remains responsive overall. For a system that crashes frequently by design, this level of availability is more than acceptable, and it validates the effectiveness of our approach.



In many real-world systems, client-side retries are essential for reliability, allowing failed requests to be retried automatically. But retries have risks. If a request modifies data and fails midway, retrying without care can lead to inconsistent or duplicated results. This is why **idempotency** is so important. For example, a payment might succeed on the server but fail to return a response, leading the client to retry and charge the customer twice. Such issues can cause loss of trust, customer complaints, and financial discrepancies. Critical operations like payments, deposits, or order processing must be designed so retries are safe, using techniques such as idempotent handling, transactions, rollbacks, request identifiers, or, in more complex cases, distributed transaction patterns like **Saga**. For more guidance, see the AWS white paper [Cloud design patterns, architectures, and implementations \(nodejsdp.link/cloud-design-patterns\)](https://aws.amazon.com/blogs/architecture/cloud-design-patterns/).

Zero-downtime restart

So far, we've focused on handling unexpected crashes by running multiple worker processes. This improves resilience and helps ensure that requests are still served even when individual processes fail. But what happens if we need to restart the entire server application, for instance, when we want to deploy a new version to production?

During a deployment, a Node.js application typically needs to be restarted. You provision the new version of the code, stop the server, and then start it again so it can load and run the updated application. While the server is stopped and restarted, there is a brief period when no requests can be handled. This results in a loss of availability. That may be acceptable for a personal blog or a side project, but it is not suitable for production systems, especially those with **service-level agreements (SLAs)** and for quickly evolving applications that are updated multiple times a day through continuous delivery pipelines.

To avoid this problem, we need to implement **zero-downtime restarts**. This technique allows the application to be updated and restarted without interrupting its ability to serve requests. It preserves availability throughout the entire deployment process.

With the `node:cluster` module, implementing this technique is relatively straightforward. The pattern involves restarting the workers one at a time, so the remaining workers can continue handling requests and keep the application available throughout the process.

Let's add this feature to our clustered server. All we need to do is include a bit of extra logic in the primary process to manage the restarts in a controlled and sequential way:

```
import { once } from 'node:events'
// ...
if (cluster.isPrimary) {
  // ...
  process.on('SIGUSR2', async () => { // 1
    const workers = Object.values(cluster.workers)
    for (const worker of workers) { // 2
      console.log(`Stopping worker: ${worker.process.pid}`)
      worker.disconnect() // 3
      await once(worker, 'exit')
      if (!worker.exitedAfterDisconnect) {
        continue
      }
      const newWorker = cluster.fork() // 4
      await once(newWorker, 'listening') // 5
    }
  })
} else {
  // ...
}
```

This is how the preceding code block works:

1. The workers restart action is triggered when the application receives the SIGUSR2 signal. Note that we are using an async function to implement the event handler, as we will need to perform some asynchronous tasks here.

2. When a `SIGUSR2` signal is received, we iterate over all the values of the `cluster.workers` object. Every element is a worker object that we can use to interact with a given worker currently active in the pool of workers.
3. The first thing we do for the current worker is invoke `worker.disconnect()`, which stops the worker gracefully. This means that if the worker is currently handling requests, this won't be interrupted abruptly; instead, it will wait for it to complete handling the request. The worker exits only after the completion of all in-flight requests.
4. When the terminated process exits, we can spawn a new worker.
5. We wait for the new worker to be ready and listening for new connections before we proceed with restarting the next worker.



Since our program makes use of Unix signals, it will not work properly on Windows systems (unless you are using the Windows Subsystem for Linux). Signals are the simplest mechanism to implement our solution. However, this isn't the only one. In fact, other approaches include listening for a command coming from a socket, a pipe, or the standard input.

Now, we can test our zero-downtime restart by running the application and then sending a `SIGUSR2` signal. However, we first need to obtain the PID of the master process. The following command can be useful to identify it from the list of all the running processes:

```
ps -af
```

The master process should be the parent of a set of node processes. Once we have the PID we are looking for, we can send the signal to it:

```
kill -SIGUSR2 <PID>
```

Now, the output of the application should display something like this:

```
Restarting workers
Stopping worker: 19389
Started 19407
Stopping worker: 19390
Started 19409
```

We can try to use `autocannon` again to verify that we don't have any considerable impact on the availability of our application during the restart of the workers. To make sure the new code is running after the restart, try changing something in the worker code. For example, you could update the response format and see whether the change appears once the restart is complete.



`pm2` (nodejssdp.link/pm2) is a small utility, based on `node:cluster`, which offers load balancing, process monitoring, zero-downtime restarts, and other goodies.

Dealing with stateful communications

The `node:cluster` module does not work well with stateful communications where the application state is not shared between the various instances. This is because different requests belonging to the same stateful session may potentially be handled by a different instance of the application. This is not a problem limited only to the `node:cluster` module, but, in general, it applies to any kind of stateless, load-balancing algorithm. Consider, for example, the situation described by *Figure 12.3*:

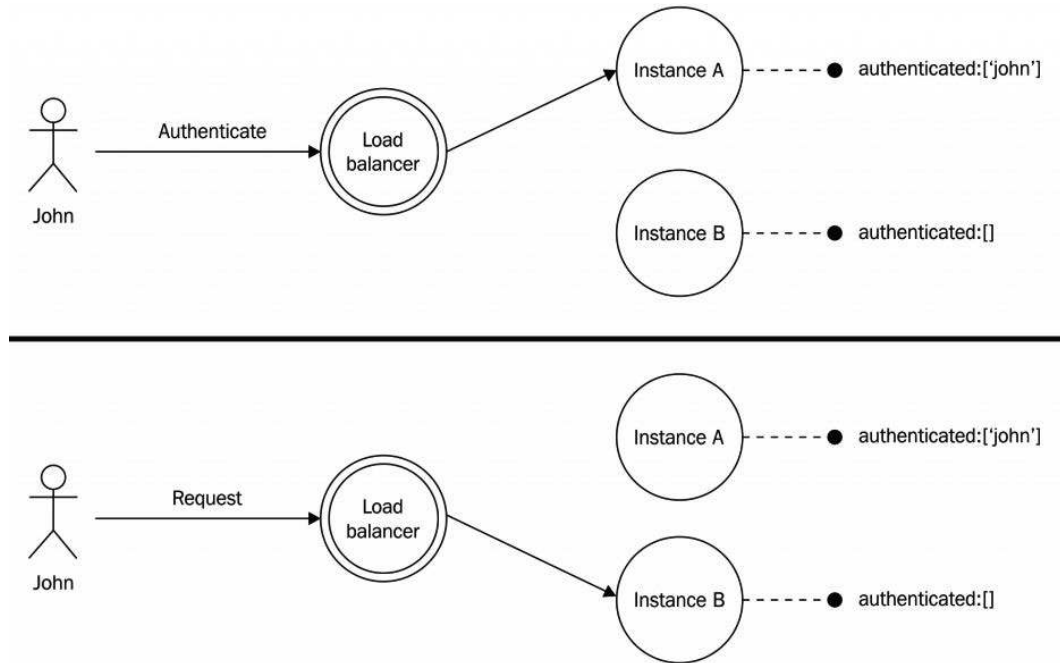


Figure 12.3: An example issue with a stateful application behind a load balancer

In this example, the user **John** sends a request to log in. The request is handled by one of the application instances (**Instance A**), which stores the result of the authentication locally, for example, in memory. This means that only Instance A knows John is authenticated.

Later, when John sends another request, the load balancer might forward it to a different instance, such as **Instance B**. Since Instance B does not have any record of John's authentication, it treats him as unauthenticated and rejects the request.

This setup creates a problem. The application cannot be scaled reliably across multiple instances if the authentication state is stored locally. Fortunately, there are a couple of simple solutions that can help resolve this issue.

Sharing the state across multiple instances

The first option to scale an application using stateful communications is to find a way to reliably share state across all the instances.

This can be achieved with a shared datastore, such as, for example, a database like PostgreSQL (nodejsdp.link/postgresql), MongoDB (nodejsdp.link/mongodb), or CouchDB (nodejsdp.link/couchdb), or we can also use an in-memory store such as Redis (nodejsdp.link/redis) or Memcached (nodejsdp.link/memcached).

Figure 12.4 outlines this simple and effective solution:

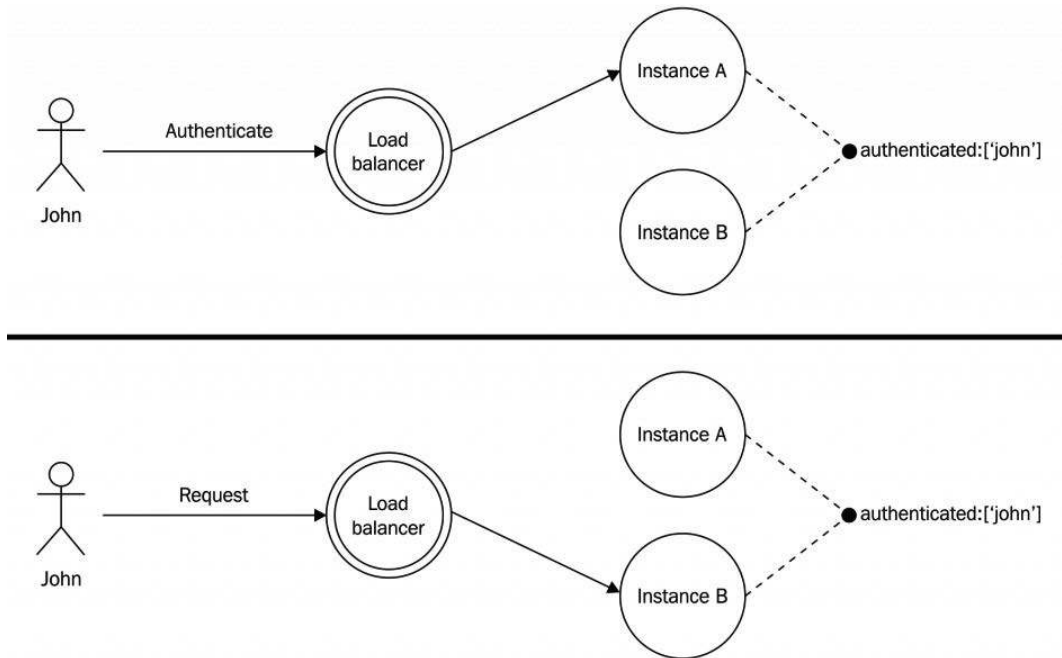


Figure 12.4: Application behind a load balancer using a shared data store

The only drawback of using a shared store for the communication state is that applying this pattern might require a significant amount of refactoring of the codebase. For example, we might be using an existing library that keeps the communication state in memory, so we must figure out how to configure, replace, or reimplement this library to use a shared store.



Using a shared data store is often the preferred choice for building scalable, reliable applications, as it lets multiple instances share the same state and maintain consistency across the system. The trade-off is added cost and complexity. The data store must scale with traffic, remain highly available, and withstand failures. You also take on operational tasks such as backups, securing access, monitoring performance, and keeping the system updated. These responsibilities grow as your application scales.

Sticky load balancing

In cases where refactoring might not be feasible, for instance, because of too many changes required or stringent time constraints, we can rely on a less invasive solution: **sticky load balancing** (or **sticky sessions**). In this approach, the load balancer ensures that all requests associated with a given session are always routed to the same instance of the application.

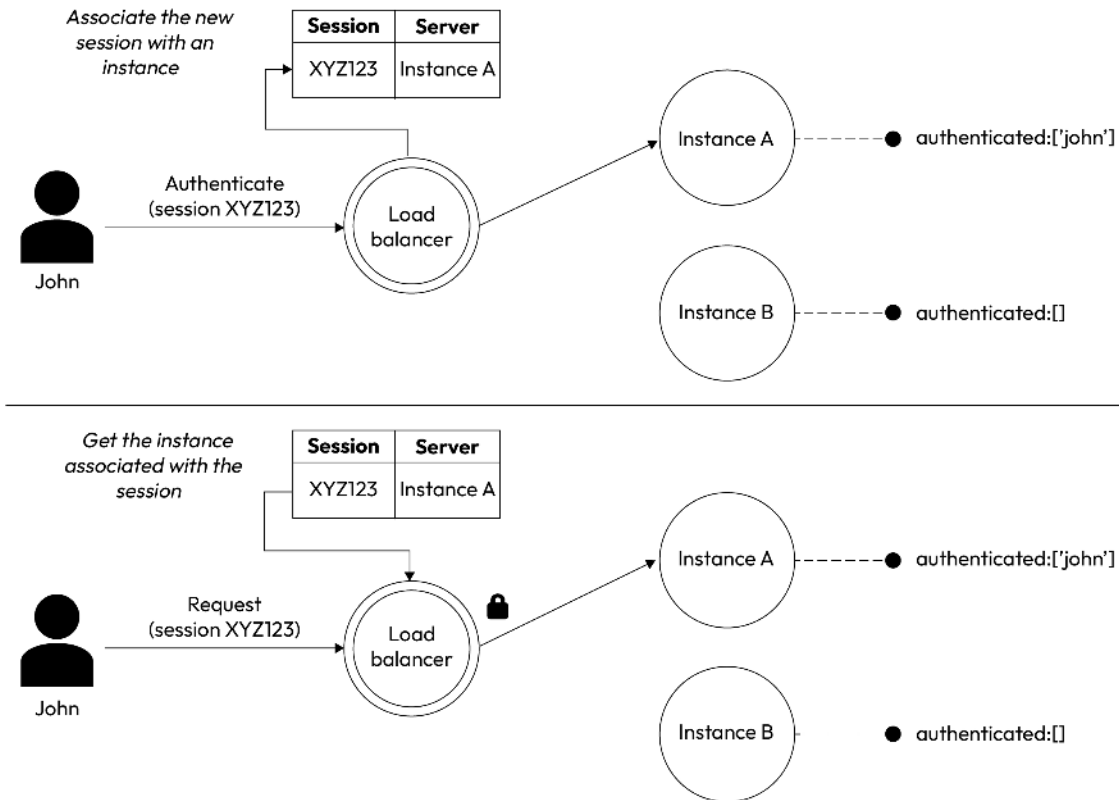


Figure 12.5: An example illustrating how sticky load balancing works

As shown in Figure 12.5, when the load balancer receives an initial request that starts a new session, it selects an application instance using its load balancing algorithm and stores a mapping between the session and that instance. On subsequent requests from the same session, the load balancer uses this mapping to route traffic directly to the same instance, skipping the usual selection process. This mechanism typically relies on inspecting a session identifier, which is often included in a cookie set by the application or the load balancer itself.

An alternative approach to achieving stickiness is to use the client's IP address to determine the target instance. In this case, the IP is passed through a hash function that consistently maps it to the same instance. This method has the advantage of not requiring the load balancer to maintain a session-to-instance mapping. However, it can be unreliable for clients with frequently changing IP addresses, such as mobile users switching between networks.



Sticky load balancing is not supported by default in the `node:cluster` module, but it can be added using the `sticky-session` package from npm (nodejsdp.link/sticky-session). Be aware, though, that this library hasn't been actively maintained in a while. Yet, it can still serve as a useful reference and a source of inspiration for learning how sticky session handling can be implemented.

One big limitation of sticky load balancing is that it takes away many of the benefits of having a redundant system. In a well-designed setup, all application instances should be able to handle any request. If one instance goes down, another should be able to take its place without any issues. Sticky sessions make this harder by tying a user session to a specific instance, which reduces flexibility and makes failover more difficult. For this reason, sticky load balancing is usually not recommended. A better option is to store session data in a shared data store that all instances can access. This way, any instance can handle any request, and the system can scale and recover more easily.

Even better, when possible, is to design applications that do not require session state on the server at all. For example, you can include all the necessary information in each request, so the server does not need to remember anything. This is the idea behind stateless authentication methods like **JSON Web Tokens** (JWTs), where the session data is encoded directly into the token and sent with every request.

One real-world example of a library that typically needs sticky load balancing is `Socket.IO` (nodejsdp.link/socket-io), which keeps a persistent connection and stores session data in memory.

Scaling with a reverse proxy

The `node:cluster` module, although very convenient and simple to use, is not the only option we have to scale a Node.js web application. Traditional techniques are often preferred because they offer more control and power in highly available production environments.

The alternative to using `node:cluster` is to start multiple standalone instances of the same application running on different ports or machines and then use a **reverse proxy** (or gateway) to provide access to those instances, distributing the traffic across them. In this configuration, we don't have a master process distributing requests to a set of workers, but a set of distinct processes running on the same machine (using different ports) or scattered across different machines inside a network. To provide a single access point to our application, we can use a reverse proxy, a special device or service placed between the clients and the instances of our application, which takes any request and forwards it to a destination server, returning the result to the client as if it were itself the origin. In this scenario, the reverse proxy is also used as a load balancer, distributing the requests among the instances of the application.



For a clear explanation of the differences between a reverse proxy and a forward proxy, you can refer to the Apache HTTP server documentation at [nodejsdp.link/forward-reverse](https://httpd.apache.org/docs/2.4/proxy.html).

Figure 12.6 shows a typical multi-process, multi-machine configuration with a reverse proxy acting as a load balancer on the front:

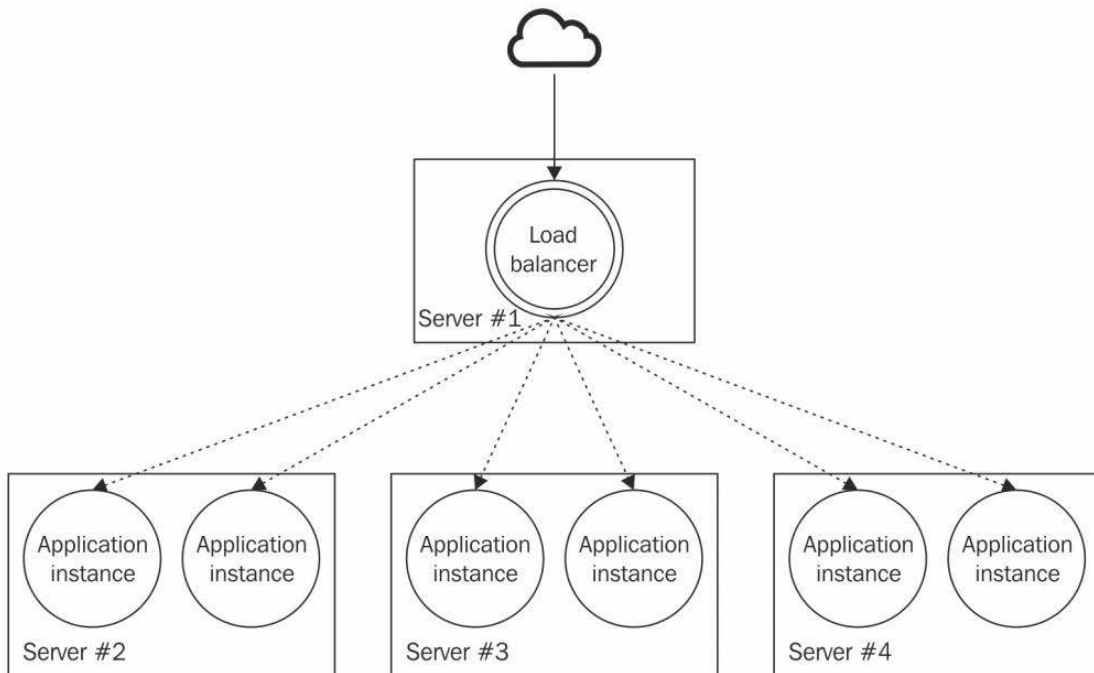


Figure 12.6: A typical multi-process, multi-machine configuration with a reverse proxy acting as a load balancer

For a Node.js application, there are many reasons to choose this approach in place of the cluster module:

- A reverse proxy can distribute the load across several machines, not just several processes.
- The most popular reverse proxies on the market support sticky load balancing out of the box.
- A reverse proxy can route a request to any available server, regardless of its programming language or platform.
- We can choose more powerful load-balancing algorithms.
- Many reverse proxies offer additional powerful features such as URL rewrites, caching, SSL termination point, security features (for example, denial-of-service protection), or even the functionality of fully-fledged web servers that can be used to, for example, serve static files.

That said, the `node:cluster` module could also be easily combined with a reverse proxy, if necessary, for example, by using `node:cluster` to scale vertically inside a single machine and then using the reverse proxy to scale horizontally across different nodes.



Pattern

Use a reverse proxy to balance the load of an application across multiple instances running on different ports or machines.

We have many options to implement a load balancer using a reverse proxy. The following is a list of the most popular solutions:

- **Nginx** (nodejsdp.link/nginx): This is a web server, reverse proxy, and load balancer, built upon the non-blocking I/O model.
- **HAProxy** (nodejsdp.link/haproxy): This is a fast load balancer for TCP/HTTP traffic.
- **Caddy** (nodejsdp.link/caddy): This is a powerful, extensible web server written in Go, with automatic HTTPS being one of the standout features that helped make it famous, among many others.
- **Node.js-based proxies**: There are many solutions for the implementation of reverse proxies and load balancers directly in Node.js. This might have advantages and disadvantages, as we will see later.
- **Cloud-based proxies**: In the era of cloud computing, it's not rare to utilize a load balancer as a service. This can be convenient because it requires minimal maintenance, it's usually highly scalable, and sometimes it can support dynamic configurations to enable on-demand scalability.



A newer alternative worth exploring is **Watt** (nodejsdp.link/wattpm), a Node.js application server that supports running and managing multiple services from a central point. It also offers features such as a built-in mesh network, logging, monitoring with **Prometheus**, and integration with **OpenTelemetry**. These capabilities can be useful when building more advanced Node.js deployments.

In the next few sections of this chapter, we will analyze a sample configuration using Nginx. Later, we will work on building our very own minimal load balancer using nothing but Node.js!

Load balancing with Nginx

To give you an idea of how reverse proxies work, we will now build a scalable architecture based on Nginx, but first, we need to install it. You can do that by using your system packages of choice, or you can follow the official instructions at nodejsdp.link/nginx-install.



Note that the following examples have been tested with the latest version of Nginx available at the time of writing (1.27.5).

Since we are not going to use `node:cluster` to start multiple instances of our server, we need to slightly modify the code of our application so that we can specify the listening port using a command-line argument. This will allow us to launch multiple instances on different ports. Let's consider the main module of our example application (`app.js`):

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

const server = createServer((_req, res) => {
  let i = 1e7
  while (i > 0) {
    i--
  }
  console.log(`Handling request from ${process.pid}`)
  res.end(`Hello from ${process.pid}\n`)
})

const port = Number.parseInt(process.env.PORT || process.argv[2]) || 8080
server.listen(port, () => console.log(`Started at ${process.pid}`))
```

The only difference between this version and the first version of our web server is that here, we are making the port number configurable through the `PORT` environment variable or a command-line argument. This is needed because we want to be able to start multiple instances of the server and allow them to listen on different ports.

Another important feature that we won't have available without `node:cluster` is the automatic restart in case of a crash. Luckily, this is easy to fix by using a dedicated supervisor, that is, an external process that monitors our application and restarts it if necessary. The following are some possible choices:

- Node.js-based supervisors such as `pm2` (nodejsdp.link/pm2)
- OS-based monitors such as `systemd` (nodejsdp.link/systemd) or `runit` (nodejsdp.link/runit)
- More advanced monitoring solutions such as `monit` (nodejsdp.link/monit) or `supervisord` (nodejsdp.link/supervisord)
- Container-based runtimes such as `Kubernetes` (nodejsdp.link/kubernetes), `Nomad` (nodejsdp.link/nomad), or `Docker Swarm` (nodejsdp.link/swarm)

For this example, we are going to use `pm2` (version 6.0.6), which is one of the most common Node.js-based supervisors. We can install it globally by running the following command:

```
npm install pm2@^6.0.6 -g
```

The next step is to start the four instances of our application, all on different ports and supervised by pm2:

```
pm2 start --namespace 'app' --name 'app1' app.js -- 8081
pm2 start --namespace 'app' --name 'app2' app.js -- 8082
pm2 start --namespace 'app' --name 'app3' app.js -- 8083
pm2 start --namespace 'app' --name 'app4' app.js -- 8084
```

We can check the list of started processes using this command:

```
pm2 ls
```



You can use `pm2 stop all` to stop all the Node.js processes previously started with pm2. Alternatively, you can use `pm2 stop <namespace>` to stop all the apps under a specific namespace. In our case, `pm2 stop app` will stop all the processes we just spawned for our application. After stopping the processes, running `pm2 list` will still show the applications, but their status will be marked as “stopped.” If you want to remove them entirely from pm2's process list, you can use `pm2 delete app`.

Now, it's time to configure the Nginx server as a load balancer.

First, we need to create a minimal configuration file in our working directory that we will call `nginx.conf`.



Note that, because Nginx allows you to run multiple applications behind the same server instance, it is more common to use a global configuration file, which, in Unix systems, is generally located under `/usr/local/nginx/conf`, `/etc/nginx`, or `/usr/local/etc/nginx`. Here, by having a configuration file in our working folder, we are taking a simpler approach. This is okay for the sake of this demo, as we want to run just one application locally, but we advise you to follow the recommended best practices for production deployments.

Next, let's write the `nginx.conf` file and apply the following configuration, which is the very minimum required to get a working load balancer for our Node.js processes:

```
daemon off; ## 1
error_log /dev/stderr info; ## 2

events { ## 3
    worker_connections 2048;
}

http { ## 4
    access_log /dev/stdout;
```

```
upstream my-load-balanced-app {  
    server 127.0.0.1:8081;  
    server 127.0.0.1:8082;  
    server 127.0.0.1:8083;  
    server 127.0.0.1:8084;  
}  
  
server {  
    listen 8080;  
  
    location / {  
        proxy_pass http://my-load-balanced-app;  
    }  
}
```

Let's discuss this configuration together:

1. The declaration `daemon off` allows us to run Nginx as a standalone process using the current unprivileged user and by keeping the process running in the foreground of the current terminal (which allows us to shut it down using `Ctrl + C`).
2. We use `error_log` and, later in the `http` block, `access_log` to stream errors and access logs, respectively, to the standard output and standard error, so we can read the logs in real time straight from our terminal.
3. The `events` block allows us to configure how network connections are managed by Nginx. Here, we are setting the maximum number of simultaneous connections that can be opened by an Nginx worker process to 2048.
4. The `http` block allows us to define the configuration for a given application. In the `upstream my-load-balanced-app` section, we are defining the list of backend servers used to handle the network requests. In the `server` section, we use `listen 8080` to instruct the server to listen on port 8080, and finally, we specify the `proxy_pass` directive, which essentially tells Nginx to forward any request to the server group we defined before (`my-load-balanced-app`).

That's it! Now, we only need to start Nginx using our configuration file with the following command:

```
nginx -c ${PWD}/nginx.conf
```

Our system should now be up and running, ready to accept requests and balance the traffic across the four instances of our Node.js application.



By default, Nginx uses a round-robin algorithm to distribute traffic across various nodes, but this behavior can be configured (nodejsdp.link/nginx-lb).

Simply point your browser to the address `http://localhost:8080` to see how the traffic is balanced by our Nginx server. You can also try again to load test this application using `autocannon`. Since we are still running all the processes in one local machine, your results should not diverge much from what you got when benchmarking the version using the `node:cluster` module approach.

This example demonstrated how to use Nginx to load balance traffic. For simplicity, we kept everything locally on our machine, but nonetheless, this was a great exercise to get us ready to deploy an application on multiple remote servers. If you want to try to do that, you will essentially have to follow these steps:

1. Provision n backend servers running the Node.js application (running multiple instances with a service monitor such as `forever` or by using the `node:cluster` module).
2. Provision a load balancer machine that has Nginx installed and all the necessary configuration to route the traffic to the n backend servers. Every process in every server should be listed in the `upstream` block of your Nginx configuration file using the correct address of the various machines in the network.
3. Make your load balancer publicly available on the internet by using a public IP and possibly a public domain name.
4. Try to send some traffic to the load balancer's public address by using a browser or a benchmarking tool such as `autocannon`.



For simplicity, you can perform all these steps manually by spinning up servers through your cloud provider's admin interface and by using SSH to log in to those. Alternatively, you could choose tools that allow you to automate these tasks by writing **infrastructure as code** (IaC), such as **Terraform** (nodejsdp.link/terraform), **Ansible** (nodejsdp.link/ansible), and **Packer** (nodejsdp.link/packer). Using IaC is always the recommended approach because it gives you a reproducible, auditable, and *versionable* way to manage your servers.

In this example, we used a predefined number of backend servers. In the next section, we will explore a technique that allows us to load balance traffic to a dynamic set of backend servers.

Dynamic horizontal scaling

Dynamic horizontal scaling is a common cloud pattern where application capacity adjusts in real time based on traffic. In this model, new server instances can be provisioned or decommissioned in seconds simply by calling an API. This helps reduce infrastructure costs while keeping the application responsive and highly available. When traffic spikes cause performance issues, additional servers can be started automatically; when demand drops, unused servers can be shut down. For predictable traffic patterns, scaling can also follow a schedule, such as reducing capacity during off-peak hours and increasing it before peak times.

This flexibility comes with a challenge: the load balancer must always know which servers are available. It needs to stay in sync with the system's current state as services are added or removed. In the next section, we will explore how to solve this by building a load balancer that can adapt to these changes step by step

Using a service registry

A common pattern to solve the challenge of having an always-up-to-date view of the network topology is to use a central repository called a **service registry**. This helps track which services are available, how many instances are running, and what their network addresses are. The service registry maintains a current and accurate picture of the servers and the services they provide.

Figure 12.7 shows a multiservice architecture with a load balancer on the front, dynamically configured using a service registry:

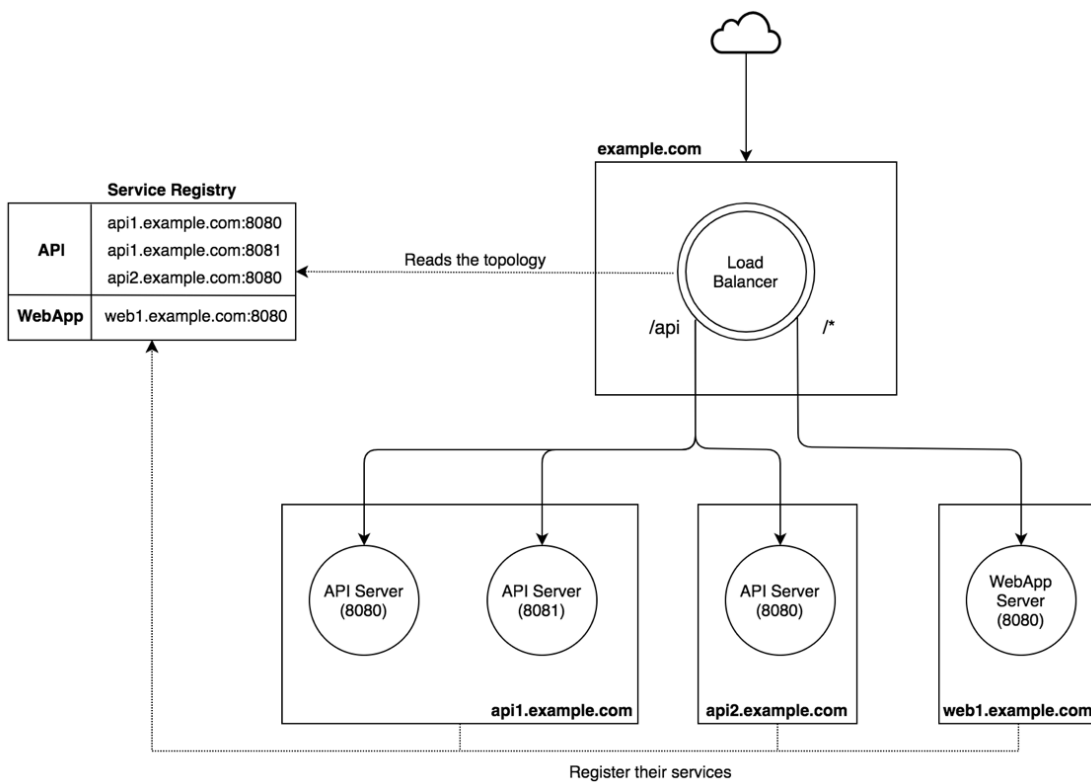


Figure 12.7: A multiservice architecture with a load balancer dynamically configured using a service registry

The architecture in *Figure 12.7* assumes the presence of two services, **API** and **WebApp**. There can be one or many instances of each service, spread across multiple servers.

When a request to `example.com` is received, the load balancer checks the prefix of the request path. If the prefix is `/api`, the request is load balanced between the available instances of the **API** service. In *Figure 12.7*, we have two instances running on the server `api1.example.com` and one instance running on `api2.example.com`. For all the other path prefixes, the request is load balanced between the available instances of the **WebApp** service. In the diagram, we have only one **WebApp** instance, which is running on the server `web1.example.com`. The load balancer obtains the list of servers and service instances running on every server using the service registry.

For this to work in complete automation, each application instance has to register itself with the service registry the moment it comes up online and unregister itself when it stops. This way, the load balancer can always have an up-to-date view of the servers and the services available on the network.



Pattern (Service Registry)

Use a central repository to store an always-up-to-date view of the servers and the services available in a system.

While this pattern is useful to load balance traffic, it has the added benefit of being able to decouple service instances from the servers on which they are running. In a way, the Service Registry pattern can be seen as a practical application of the Service Locator design pattern, but applied to network services.

Implementing a dynamic load balancer

To support a dynamic network infrastructure, we can use a reverse proxy such as **Nginx** or **HAProxy**. In this setup, a script or automation tool can be triggered by the service registry whenever there is a change in the service topology. The script regenerates the load balancer's configuration based on the updated information and then reloads it to apply the changes. For Nginx, this can be done using the following command line:

```
nginx -s reload
```



This command does not restart the Nginx process entirely. Instead, it reloads the configuration from file, allowing the changes to take effect almost immediately. The update is fast and does not interrupt existing connections.

The same result can also be achieved using a cloud-based solution, but there is a third option that is a bit more exciting and hands-on: building our own load balancer using Node.js. As we have discussed throughout this book, Node.js is well suited for building network applications. This has been one of its core strengths since the beginning. So why not take on the challenge of creating a load balancer entirely in Node.js?



To be clear, this approach is not recommended for production, as building a secure, reliable, and performant load balancer requires deep expertise. However, creating one as an exercise is valuable for experimenting with patterns like dynamic load balancing and for understanding how tools like Nginx and HAProxy work, enabling better use and troubleshooting of those tools.

For this exercise, we are going to use **Consul** (nodejsdp.link/consul) as the service registry to replicate the multiservice architecture we saw in *Figure 12.7*.



Consul is an open-source service networking solution that provides features such as service discovery, health checking, key-value storage, and multi-data center support. We chose it because it can be easily spun up locally for development and offers a simple HTTP-based API. Unlike Nginx, which is typically configured via static files, Consul's configuration can be updated dynamically through its REST API. It is a solid and widely adopted product, so becoming familiar with it may prove useful in real-world projects down the line.

We will also be leveraging two external npm packages:

- `httpxy` (nodejsdp.link/httpxy): To simplify the creation of a reverse proxy/load balancer in Node.js
- `portfinder` (nodejsdp.link/portfinder): To find a free port in the system

Before we begin, make sure that Consul is installed on your machine. At the time of writing, the latest available version is 1.21.1, which we will use as the reference for this section. If you need guidance on how to install it, you can refer to the official documentation at nodejsdp.link/consul-install.

For this example, we have implemented a very minimal Consul client that interacts with its HTTP API. The implementation details are not critical for following along, but if you're curious, you can find the code in the companion repository for this book (the `consul.js` file). This client exposes three different methods:

- `registerService()`: To register a given service
- `deregisterService()`: To remove a given service
- `getAllServices()`: To retrieve the current list of all the registered services

Let's begin by implementing our services. These will be simple HTTP servers, like the ones we used earlier when experimenting with `node:cluster` and Nginx. As before, we are not aiming to build fully featured web services. Instead, we will create a script that can spawn a lightweight service stub, responding to every request with a message such as `${serviceType} response from ${pid}`. Our focus is to implement the logic that registers each server instance with the service registry on startup and removes it when the server shuts down. This setup will allow us to simulate multiple instances of different services and verify that traffic is correctly routed by our custom load balancer.

Let's see how this looks (the app.js file):

```
import { createServer } from 'node:http'
import { randomUUID } from 'node:crypto'
import portfinder from 'portfinder' // v1.0.37
import { ConsulClient } from './consul.js'

const serviceType = process.argv[2] // 1
if (!serviceType) {
  console.error('Usage: node app.js <service-type>')
  process.exit(1)
}

const consulClient = new ConsulClient() // 2

const port = await portfinder.getPort() // 3
const address = process.env.ADDRESS || 'localhost'
const serviceId = randomUUID()

async function registerService() { // 4
  await consulClient.registerService({
    id: serviceId,
    name: serviceType,
    address,
    port,
    tags: [serviceType],
  })

  console.log(
    `${serviceType} registered successfully as ${serviceId} ` +
    `on ${address}:${port}`
  )
}

async function unregisterService(err) { // 5
  err && console.error(err)
  console.log(`deregistering ${serviceId}`)
  try {
    await consulClient.deregisterService(serviceId)
  } catch (deregisterError) {
```

```

    console.error(`Failed to deregister service: `+
      `${deregisterError.message}`)
  }
  process.exit(err ? 1 : 0)
}

process.on('uncaughtException', unregisterService) // 6
process.on('SIGINT', unregisterService)

const server = createServer((_req, res) => { // 7
  // Simulate some processing time
  let i = 1e7
  while (i > 0) {
    i--
  }
  console.log(`Handling request from ${process.pid}`)
  res.end(`${serviceType} response from ${process.pid}\n`)
})

server.listen(port, address, async () => {
  console.log(
    `Started ${serviceType} on port ${port} with PID ${process.pid}`)
  await registerService()
})

```

Let's take a closer look at some key parts of this code:

1. The script begins by reading the service type from the command-line arguments. This value is required to identify what kind of service we want to simulate through this stub. If the argument is missing, the script prints a usage message and exits.
2. A new instance of `ConsulClient` is created. This client will be used to communicate with the Consul service registry (running locally) to handle registration and deregistration.
3. The script uses the `portfinder` module to automatically select an available port. This helps ensure that multiple instances can run without having to manually assign ports. The server will bind to this port when it starts. Along with selecting the port, the script also determines the network address to bind to. It checks for an `ADDRESS` environment variable, and if not set, defaults to "localhost". This allows the address to be configured externally when needed. A unique service ID is also generated using `randomUUID()`. This ID will be used to register the service with Consul, ensuring that each instance can be uniquely identified.

4. A `registerService()` function is defined to register the current instance with Consul. It sends the service ID, type, network address, port, and a tag identifying the service type. Once the registration is complete, a confirmation message is logged to the console.
5. A corresponding `unregisterService()` function is also defined. This function logs any errors (if present), attempts to deregister the service from Consul, and then gracefully exits the process. It ensures that the registry remains accurate even when a service crashes or is stopped.
6. The script sets up listeners for `SIGINT` and `uncaughtException`. When either event occurs, the `unregisterService()` function is called to clean up before the process exits. This is important for maintaining consistency in the service registry.
7. Finally, a simple HTTP server is created. It simulates some processing time and responds to every request with a message that includes the service type and the process ID. When the server starts listening, it logs a message and immediately registers the service with Consul.

Now, it's time to implement the load balancer. Let's do that by creating a new module called `loadBalancer.js`:

```
import { createServer } from 'node:http'
import { createProxyServer } from 'httpxy' // v0.1.7
import { ConsulClient } from './consul.js'

const routing = [ // 1
  {
    path: '/api',
    service: 'api-service',
    index: 0,
  },
  {
    path: '/',
    service: 'webapp-service',
    index: 0,
  },
]

const consulClient = new ConsulClient() // 2
const proxy = createProxyServer()

const server = createServer(async (req, res) => { // 3
  const route = routing.find(route => req.url.startsWith(route.path))

  try {
    const services = await consulClient.getAllServices() // 4
    const servers = Object.values(services).filter(service =>
```

```

    service.Tags.includes(route.service)
  )

  if (servers.length > 0) {
    route.index = (route.index + 1) % servers.length // 5
    const server = servers[route.index]
    const target = `http://${server.Address}:${server.Port}`
    proxy.web(req, res, { target })
    return
  }
} catch (err) {
  console.error(err)
}

// if servers not found or error occurs
res.writeHead(502)
return res.end('Bad gateway')
})

server.listen(8080, () => {
  console.log('Load balancer started on port 8080')
})

```

Once again, let's explain the most important part of our example code:

1. We define the routing configuration for our load balancer. Each entry maps a URL path to a service name registered in Consul. The index field is used to keep track of the last server used for that service, which enables a round-robin strategy when routing requests.
2. We create an instance of our custom ConsulClient to interact with the service registry. Then, we initialize a proxy server using the httpxy module, which we will use to forward incoming requests to the appropriate service instance.
3. Inside the request handler, we attempt to match the request URL against the configured routes. If a match is found, we use the associated service name to determine which backend servers to contact.
4. We query Consul for the full list of registered services, then filter it to find only the instances tagged with the service name from the matched route. This gives us the current list of healthy servers that can handle the request.
5. If at least one matching server is found, we increment the index for that route and select the corresponding server in a round-robin fashion. The request is then forwarded to the selected target using proxy.web(). If no servers are found or an error occurs, we respond with a 502 Bad Gateway error.

It should now be clear how simple it is to implement a load balancer using only Node.js and a service registry, and how much flexibility this approach can offer.



To keep the implementation simple, we left out some useful optimizations. In this version, the load balancer queries Consul for the list of services on every request, which can add overhead at high request rates. A better approach would be to cache the list and refresh it periodically, for example every 10 seconds. We could also use the `node:cluster` module to run multiple load balancer instances and take advantage of all CPU cores. Another valuable improvement is using Consul's built-in health checks to detect and remove unhealthy services automatically, such as when a crash or out-of-memory error prevents them from deregistering.

Now, we should be ready to give our system a try!

The following command allows us to start the Consul service registry locally:

```
consul agent -dev
```

Now, we are ready to start the load balancer:

```
node loadBalancer.js
```

Now, if we try to access some of the services exposed by the load balancer, we will notice that it returns an HTTP 502 error, because we haven't started any servers yet. Try it yourself:

```
curl localhost:8080/api
```

The preceding command should return the following output:

```
Bad Gateway
```

The situation will change if we spawn some instances of our services, for example, two `api-service` instances and one `webapp-service` instance (in different terminals):

```
app.js api-service
app.js api-service
app.js webapp-service
```

Now, the load balancer should automatically see the new servers and start distributing requests across them. Let's try again with the following command:

```
curl localhost:8080/api
```

The preceding command should now return a message that looks like the following:

```
api-service response from 6972
```

By running this again, we should now receive a message from another server, confirming that the requests are being distributed evenly among the different servers, for example:

```
api-service response from 6979
```

You can also experiment by starting additional instances or stopping some of them to see how the load balancer continues routing traffic only to the ones that are still available.

The advantages of this pattern are immediate. We can now scale our infrastructure dynamically, on demand, or based on a schedule, and our load balancer will automatically adjust with the new configuration without any extra effort!

Now that we know how to perform dynamic load balancing using a load balancer and a service registry, we are ready to explore some interesting alternative approaches, starting with peer-to-peer load balancing.

Peer-to-peer load balancing

When exposing a complex internal network to the public internet, using a reverse proxy is often essential. It simplifies external access by hiding internal details and providing a single, consistent entry point for outside users. However, when dealing with services that are only used internally, we have more flexibility and control.

Imagine a service, called **Service A**, that depends on another internal service, **Service B**. **Service B** is deployed across multiple machines and is only available within the internal network. Based on what we've learned so far, **Service A** would typically send requests to **Service B** through a load balancer. The load balancer would then take care of distributing the traffic across all available instances of **Service B**.

But there is another option. Instead of routing requests through a central load balancer, **Service A** can handle the distribution itself. In this setup, **Service A** becomes responsible for balancing the requests across the different instances of **Service B**. This works only if **Service A** knows how to reach those instances, which is usually the case in controlled internal networks. This approach is known as **peer-to-peer load balancing** (or **client-side load balancing**).

Figure 12.8 illustrates the difference between centralized load balancing and peer-to-peer load balancing:

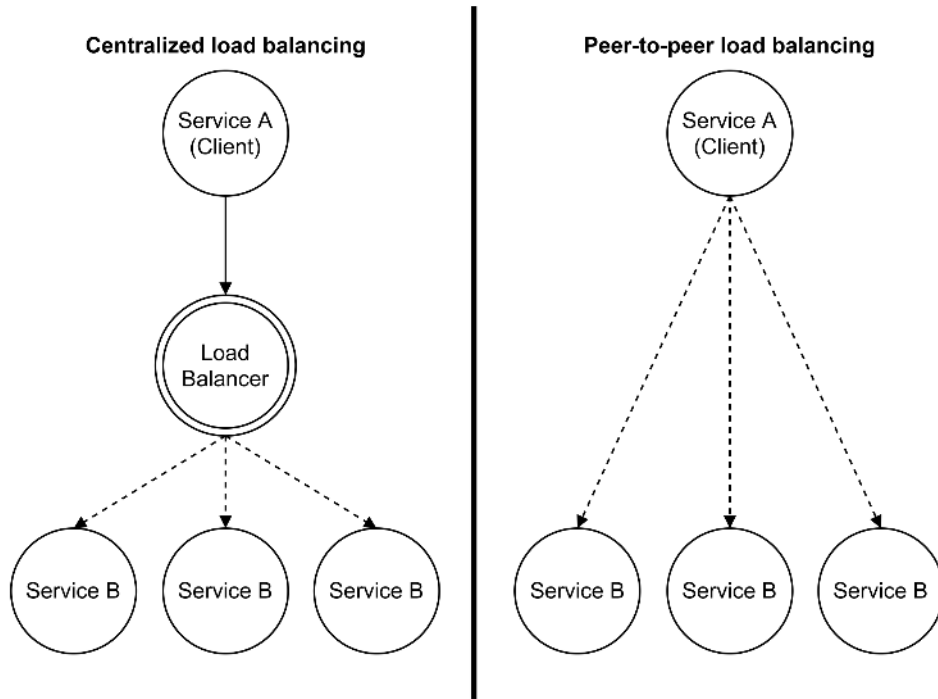


Figure 12.8: Centralized load balancing versus peer-to-peer load balancing

This is an extremely simple and effective pattern that enables truly distributed communications without bottlenecks or single points of failure. Besides that, it also has the following properties:

- It reduces infrastructure complexity by removing an extra network component (the load balancer).
- It enables faster communication since messages are sent directly to their destination.
- It improves scalability because the system is no longer limited by the performance of a central load balancer.

However, this approach also introduces new responsibilities. Without a load balancer, the client must now be aware of the underlying infrastructure and implement its own load-balancing logic. It may also need a way to keep its view of the available service instances up to date.

As always, we have some pros and some cons. There's no free lunch, just a different set of trade-offs. But having options is always a good thing, because our job is to evaluate them and choose the one that offers the best trade-offs for the problem we're trying to solve.



Peer-to-peer load balancing is a pattern used extensively in the ZeroMQ (nodejsdp.link/zeromq) library, which we will use in the next chapter.

In the next section, we will showcase an example of implementing peer-to-peer load balancing in an HTTP client.

Implementing an HTTP client that can balance requests across multiple servers

We've already seen how to build a load balancer in Node.js that distributes incoming requests across available servers. Applying the same idea on the client side is not very different. The main change is that the client becomes responsible for choosing which server to send each request to.

To do this, we can simply wrap the existing client API and add a basic load-balancing mechanism. The following module (`balancedRequest.js`) shows how this can be done:

```
const servers = [
  { host: 'localhost', port: 8081 },
  { host: 'localhost', port: 8082 },
]
let i = 0

export function balancedRequest(url, fetchOptions = {}) {
  i = (i + 1) % servers.length
  const server = servers[i]

  const rewrittenUrl = new URL(url, `http://${server.host}:${server.port}`)
  rewrittenUrl.host = `${server.host}:${server.port}`

  return fetch(rewrittenUrl.toString(), fetchOptions)
}
```

The code above defines a wrapper around the `fetch` API that handles load balancing for us. It maintains a list of available servers and uses a round-robin algorithm to select one for each request. You can call the `balancedRequest()` function as many times as needed, without worrying about which server to contact. The function will automatically distribute the requests across all available servers, one at a time, helping to balance the load in a simple and transparent way.

Let's see how we can use this function in the `client.js` file:

```
import { balancedRequest } from './balancedRequest.js'

for (let i = 0; i < 10; i++) {
  const response = await balancedRequest(`/?request=${i}`)
  const body = await response.text()
  console.log(
    `Request ${i} completed\nStatus: ${response.status}\nBody: ${body}`
  )
}
```

```
)
}
```

This code sends 10 HTTP requests using the `balancedRequest()` function. Each request is automatically routed to a different server in the pool, and the response status and body are logged to the console for each one. Note that we are passing a different value for the request query string parameter in each call. This allows us to track each request individually and confirm that the responses are coming from different servers, which helps verify that the load balancing is working as expected.

Before we can test this setup, we need a sample server application to handle the incoming requests. Below is the code (`app.js`) for a basic server that will respond to each request so we can verify that load balancing is working as expected:

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

const { pid } = process
const server = createServer((req, res) => {
  const url = new URL(req.url, `http://${req.headers.host}`)
  const searchParams = url.searchParams

  console.log(
    `Handling request ${searchParams.get('request')} from ${pid}`
  )
  res.end(`Hello from ${pid}\n`)
})

const port = Number.parseInt(process.env.PORT || process.argv[2]) || 8080
server.listen(port, () => console.log(`Started at ${pid}`))
```

This is a simple HTTP server that listens on a specified port and logs incoming requests. For each request, it extracts the request query string parameter and logs it along with the PID. The server then responds with a message that includes the PID, making it easy to see which instance handled each request.

We're now ready to run the example. Start two instances of the sample server in separate terminal windows:

```
node app.js 8081
node app.js 8082
```

Once both servers are running, you can start the client application:

```
node client.js
```

If you observe the output from the servers and the client, you'll see that each request is handled by a different server. This confirms that our client-side load balancing is working correctly, even without a dedicated load balancer in place.



An improvement to the wrapper we created previously would be to integrate a service registry directly into the client and obtain the server list dynamically.

In the next section, we will explore the field of containers and container orchestration and see how, in this specific context, the runtime takes ownership of many scalability concerns.

Scaling applications using containers

In this section, we will show how containers and orchestration platforms like Kubernetes can help you write simpler Node.js applications by offloading scaling concerns such as load balancing, elastic scaling, and high availability to the platform. These are broad topics beyond the full scope of this book, so our aim is not an in-depth guide but to share basic examples for getting started with Node.js. More importantly, we want you to grasp the core principles of container orchestration and see how they can help you build more scalable and reliable applications.

What is a container?

You’ve probably heard the classic developer phrase, “It works on my machine,” often right after someone reports a production issue. Containers aim to make that excuse obsolete by ensuring applications behave the same everywhere, whether on your laptop, a colleague’s computer, or a cloud server.

A **Linux container**, as defined by the **Open Container Initiative** (OCI) (nodejsdp.link/opencontainers), is “a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.” In practice, this means you can package and run applications seamlessly across different machines, from local development to production in the cloud.

Containers are highly portable and run with minimal overhead, nearly as fast as native applications. They let you define and run *isolated* processes directly on Linux, and tools like **Docker** (nodejsdp.link/docker) make creating and running OCI-compliant containers straightforward.



On Windows and macOS, Docker uses a virtualized Linux environment under the hood.

Thanks to their portability and performance, containers have become a major advancement over traditional virtual machines for virtualizing applications in the cloud. You can install Docker by following the instructions for your operating system in the official documentation: nodejsdp.link/docker-docs.

Creating and running a container with Docker

Let's write an extremely simple web server application that we will be using as a reference for this section (`app.js`):

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

const version = 1

const server = createServer((_req, res) => {
  res.end(`Hello from ${hostname()} (v${version})`)
})

server.listen(8080)
```

When this web server receives a request, it responds with the machine's hostname and the current version of the application. For example, if you run the server locally using the command `node app.js` and send a request to it, you should receive a response like the following:

```
Hello from my-amazing-laptop.local (v1)
```

Let's see how we can run this application as a container. The first thing we need to do is create a `package.json` file for the project:

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

To *dockerize* our application, we need to follow a two-step process:

1. Build a container image.
2. Run a container instance from the image.

To create the **container image** for our application, we must define a Dockerfile. A container image (or Docker image) is the actual package and conforms to the OCI standard. It contains all the source code and the necessary dependencies and describes how the application must be executed. A **Dockerfile** is a file (named Dockerfile) that defines the build script used to build a container image for an application. So, without further ado, let's write the Dockerfile for our application:

```
FROM node:24-slim
EXPOSE 8080
COPY app.js package.json /app/
WORKDIR /app
CMD ["npm", "start"]
```

Our Dockerfile is quite short, but there are a lot of interesting things here, so let's discuss them one by one:

- `FROM node:24-slim` indicates the base image that we want to use. A base image allows us to build “on top” of an existing image. In this specific case, we are starting from a small (*slim*) image that already contains version 24 of Node.js. This means we don't have to be worried about describing how Node.js needs to be packaged into the container image.
- `EXPOSE 8080` informs Docker that the application will be listening for TCP connections on port 8080.
- `COPY app.js package.json /app/` copies the files `app.js` and `package.json` into the `/app` folder of the container filesystem. Containers are isolated, so, by default, they can't share files with the host operating system; therefore, we need to copy the project files into the container to be able to access and execute them.
- `WORKDIR /app` sets the working directory for the container to `/app`.
- `CMD ["npm", "start"]` specifies the command that is executed to start the application when we run a container from an image. Here, we are just running `npm start`, which, in turn, will run `node app.js`, as specified in our `package.json`. Remember that we can run both `node` and `npm` in the container only because those two executables are made available through the base image.

Now, we can use the Dockerfile to build the container image with the following command:

```
docker build .
```

This command will look for a Dockerfile in the current working directory and execute it to build our image.

The output of this command should be something like this:

```
[internal] load build definition from Dockerfile
  transferring dockerfile: 131B
[internal] load metadata for docker.io/library/node:24-slim
[internal] load .dockerignore
  transferring context: 2B
[internal] load build context
  transferring context: 60B
[1/3] FROM docker.io/library/node:24-slim@
sha256:5ae787590295f944e7dc200bf54861bac09bf21b5fdb4c9b97aee7781b6d95a2
  resolve docker.io/library/node:24-slim@
sha256:5ae787590295f944e7dc200bf54861bac09bf21b5fdb4c9b97aee7781b6d95a2

...truncated for brevity...

[2/3] COPY app.js package.json /app/
[3/3] WORKDIR /app
exporting to image
exporting layers
writing image
sha256:777ea4cf37c27ce9b41ab0fa2d889f0868d863167942306d98bfc074e4ffd245
```



Note that if you have never used the `node:24-slim` image before (or if you have recently wiped your Docker cache), you will also see some additional output, indicating the download of this container image.

The final hash shown is the ID of the container image we just built. We can use it to run a container instance with the following command. It is not necessary to use the full hash; the first few characters are usually enough, as long as they uniquely identify the image among those available on your system:

```
docker run -it -p 8080:8080 777ea4c
```

This command is essentially telling Docker to run the application from image `777ea4c` in “interactive mode” (which means that it will not go in the background) and that port `8080` of the container will be mapped to port `8080` of the host machine (our operating system).

Now, we can access the application at `http://localhost:8080`. So, if we use `curl` to send a request to the web server, we should get a response like the following:

```
Hello from bcf13fdcd776 (v1)
```

Note that the hostname is now different. This is because every container is running in a sandboxed environment that, by default, doesn't have access to most of the resources in the underlying operating system.

At this point, you can stop the container by just pressing *Ctrl* + *C* in the terminal window where the container is running.



When building an image, we can use the `-t` flag to *tag* the resulting image. A tag can be used as a more predictable alternative to a generated hash to identify and run container images. For instance, if we want to call our container image `hello-web:v1`, we can use the following commands:

```
docker build -t hello-web:v1 .  
docker run -it -p 8080:8080 hello-web:v1
```

When using tags, you might want to follow the conventional format of `image-name:version`.

What is Kubernetes?

We just ran a Node.js application using containers. That's worth celebrating! But while it's an important step, it's only scratching the surface. Containers really shine when we start building more complex systems, especially applications made of multiple services that need to run across many cloud servers. In these situations, Docker alone isn't enough. We need a system that can coordinate and manage all those running containers across our infrastructure. This is where container orchestration comes in.

A container orchestration tool takes care of several important tasks:

- It allows us to combine multiple servers (called **nodes**) into a single **cluster**, where resources can be managed. Nodes can be added or removed as needed, without interrupting running services.
- It ensures high availability. If a container crashes or fails a health check, it will be restarted automatically. If a node goes down, the running workloads will be rescheduled to other available nodes.
- It provides features for service discovery and load balancing, so that services can communicate with each other reliably.
- It manages access to persistent storage, making it possible to retain data across restarts and rescheduling.
- It supports automated rollouts and rollbacks, so you can deploy updates safely and without downtime.
- It offers secret storage and configuration management, helping you inject environment values and sensitive data securely.

One of the most widely adopted orchestration systems is **Kubernetes** (nodejshp.link/kubernetes), originally developed and open-sourced by Google in 2014. The name comes from the Greek word *κυβερνήτης*, meaning “helmsman” or “pilot,” someone who steers and guides. The system was built using the experience gained from running large-scale workloads inside Google's infrastructure for years.

What makes Kubernetes stand out is its declarative configuration model. You don't need to write instructions for every step. Instead, you describe the final state you want, and Kubernetes figures out how to make it happen and keep it that way. For example, you can declare that you want three replicas (instances) of your app running, and Kubernetes will make sure that three containers are always up. If one crashes, it starts a new one automatically.

The building blocks of this configuration model are Kubernetes objects. These objects represent different elements in your infrastructure, such as:

- Your containerized applications
- The resources they need, such as CPU, memory, or storage
- Policies that define how the application behaves, such as restart strategies or upgrade rules

Each object acts as a record of intent. Once applied to the cluster, Kubernetes continuously checks that the current state matches your declared intent and makes corrections when needed.



Kubernetes is more than a tool; it's a flexible framework for running containerized applications. While its core concepts stay the same, there are many ways to run it in production, from managed cloud services like **EKS**, **AKS**, and **GKE** to self-hosted clusters, even on Raspberry Pi boards if you want full control and a bit of adventure. Its modular architecture adapts to many environments, but from a developer's perspective, interacting with a cluster is largely consistent. In this section, we'll focus on the core principles that apply to all setups.

We can interact with Kubernetes using a command-line tool called `kubectl` (nodejsdp.link/kubectl-install), which lets us deploy apps, inspect resources, manage logs, and more.

There are different ways to create a Kubernetes cluster, depending on whether you're working in development, testing, or production. For local experiments, the easiest option is `minikube` (nodejsdp.link/minikube-install), a tool that creates a fully functional single-node Kubernetes cluster right on your laptop.

Make sure you install both `kubectl` and `minikube` if you want to follow along. In the next section, we'll deploy our sample containerized Node.js app using a local Kubernetes setup.



To dig deeper, you can also explore the official Kubernetes tutorials (nodejsdp.link/kubernetes-tutorials). These interactive lessons walk you through key features with hands-on examples and are a great way to get comfortable with the basics.

Deploying and scaling an application on Kubernetes

In this section, we will be running our simple web server application on a local minikube cluster. So, make sure you have `kubectl` and `minikube` correctly installed and started.



On macOS and Linux environments, make sure to run `minikube start` and `eval $(minikube docker-env)` to initialize the working environment. The second command makes sure that when you use `docker` and `kubectl` in your current terminal, you will interact with the local minikube cluster. If you open multiple terminals, you should run `eval $(minikube docker-env)` on every terminal. You can also run `minikube dashboard` to run a convenient web dashboard that allows you to visualize and interact with all the objects in your cluster.

The first thing that we want to do is build our Docker image and give it a meaningful name:

```
docker build -t hello-web:v1 .
```

If you have configured your environment correctly, the `hello-web` image will be available to be used in your local Kubernetes cluster.



Using local images is sufficient for local development. When you are ready to go to production, the best option is to publish your images to a Docker container registry such as Docker Hub (nodejsdp.link/docker-hub), Docker registry (nodejsdp.link/docker-registry), Google Container Registry (nodejsdp.link/gc-container-registry), Amazon Elastic Container Registry (nodejsdp.link/ecr), or GitHub Container Registry (nodejsdp.link/gh-container-registry). Once you have your images published to a container registry, you can easily deploy your application to different hosts without having to rebuild the corresponding images each time.

Creating a Kubernetes deployment

Now, to run an instance of this container in the minikube cluster, we have to create a **deployment** (which is a Kubernetes object) using the following command:

```
kubectl create deployment hello-web --image=hello-web:v1
```

This should produce the following output:

```
deployment.apps/hello-web created
```

This command is basically telling Kubernetes to run an instance of the `hello-web:v1` container as an application called `hello-web`.

You can verify that the deployment is running with the following command:

```
kubectl get deployments
```

This should print something like this:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hello-web	1/1	1	1	7s

This table is basically saying that our hello-web deployment is alive and that there is one **Pod** allocated for it. A Pod is a basic unit in Kubernetes and represents a set of containers that must run together in the same Kubernetes node. Containers in the same Pod have shared resources such as storage and network. Generally, a Pod contains only one container, but it's not uncommon to see more than one container in a Pod when these containers are running tightly coupled applications.

You can list all the Pods running in the cluster with this:

```
kubectl get pods
```

This should print something like:

NAME	READY	STATUS	RESTARTS	AGE
hello-web-65f47d9997-df7nr	1/1	Running	0	2m19s

Now, in order to be able to access the web server from our local machine, we need to *expose* the deployment:

```
kubectl expose deployment hello-web --type=LoadBalancer --port=8080
minikube service hello-web
```

The first command tells Kubernetes to create a LoadBalancer object that exposes the instances of the hello-web app, connecting to port 8080 of every container.

The second command is a minikube helper command that allows us to get the local address to access the load balancer. This command will also open a browser window for you, so now you should see the container response in the browser, which should look like this:

```
Hello from hello-web-65f47d9997-df7nr (v1)
```

Scaling a Kubernetes deployment

Now that our application is running and is accessible, let's start to experiment with some of the capabilities of Kubernetes. For instance, why not try to scale our application by running five instances instead of just one? This is as easy as running the following:

```
kubectl scale --replicas=5 deployment hello-web
```

Now, `kubectl get deployments` should show us the following status:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hello-web	5/5	5	5	9m18s

And `kubectl get pods` should produce something like this:

NAME	READY	STATUS	RESTARTS	AGE
hello-web-65f47d9997-df7nr	1/1	Running	0	9m24s
hello-web-65f47d9997-g98jb	1/1	Running	0	14s
hello-web-65f47d9997-hbdkx	1/1	Running	0	14s
hello-web-65f47d9997-jnfd7	1/1	Running	0	14s
hello-web-65f47d9997-s54g6	1/1	Running	0	14s

If you try to hit the load balancer now, chances are you will see different hostnames as the traffic gets distributed across the available instances. This should be even more apparent if you try to hit the load balancer while putting the application under stress, for instance, by running an autocannon load test against the load balancer URL.

Kubernetes rollouts

Now, let's try out another feature of Kubernetes: rollouts. What if we want to release a new version of our app?

We can set `const version = 2` in our `app.js` file and create a new image:

```
docker build -t hello-web:v2 .
```

At this point, to upgrade all the running Pods to this new version, we must run the following command:

```
kubectl set image deployment/hello-web hello-web=hello-web:v2
```

The output of this command should be as follows:

```
deployment.apps/hello-web image updated
```

If everything worked as expected, you should now be able to refresh your browser page and see something like the following:

```
Hello from hello-web-567b986bfb-qjvfw (v2)
```

What just happened behind the scenes is that Kubernetes started to roll out the new version of our image by replacing the containers one by one. When a container is replaced, the running instance is stopped gracefully. This way requests that are currently in progress can be completed before the container is shut down.

This wraps up our mini tutorial on Kubernetes. The key takeaway is that with a container orchestrator like Kubernetes, application code stays simpler since scaling, rollouts, and restarts are handled by the platform, which is a major advantage. The trade-off is the need to learn and manage the platform, which may not be worth it for small production apps. For large-scale systems serving millions, the investment can pay off greatly.

One more thing worth emphasizing is that in Kubernetes, containers are often “disposable” and can be killed or restarted at any time and, therefore, they need to be stateless. This can be a double-edged sword: on one hand, designing stateless applications from the start may feel like a premature optimization; on the other hand, if you need to scale quickly, your architecture will already be prepared to scale on the X-axis of the Scale Cube. Either way, any data that must persist should be stored in databases or persistent volumes.



If you want to clean up your system from the containers you just ran in the preceding examples and stop minikube, you can do so with the following commands:

```
kubectl scale --replicas=0 deployment hello-web
kubectl delete -n default service hello-web
minikube stop
```

In the next and last part of this chapter, we will explore some interesting patterns to decompose a monolithic application into a set of decoupled microservices, something that is critically important if you have built a monolithic application and are now suffering from scalability issues.

Decomposing complex applications

So far in this chapter, we’ve mostly explored the X-axis of the scale cube. This kind of scaling is often the most straightforward, as it involves running multiple instances of the same application to distribute traffic and improve availability.

Now we’ll turn our attention to the Y-axis of the scale cube, which focuses on **decomposing** (breaking down) an application by business functionality. This approach not only helps scale the system’s capacity and manage its complexity but also makes it easier to scale teams. By assigning ownership of individual services to different teams, organizations can distribute responsibility and move faster within well-defined domains.

Monolithic architecture

The term **monolithic** might make us think of a system without modularity, where all the services of an application are interconnected and almost indistinguishable. However, this is not always the case. Often, monolithic systems have a highly modular architecture and a good level of decoupling between their internal components.

A perfect example is the Linux OS kernel, which is part of a category called **monolithic kernels** (in perfect opposition to its ecosystem and the Unix philosophy). Linux has thousands of services and modules that we can load and unload dynamically, even while the system is running. However, they all run in kernel mode, which means that a failure in any of them could bring the entire OS down (have you ever seen a kernel panic?). This approach is opposite to the microkernel architecture, where only the core services of the operating system run in kernel mode, while the rest run in user mode, usually each one with its own process. The main advantage of this approach is that a problem in any of these services would more likely cause it to crash in isolation, instead of affecting the stability of the entire system.



The Torvalds-Tanenbaum debate on kernel design is probably one of the most famous *flame wars* in the history of computer science, where one of the main points of dispute was exactly monolithic versus microkernel design. You can find a web version of the discussion (it originally appeared on Usenet) at nodejsdp.link/torvalds-tanenbaum.

It's remarkable how these design principles, which are more than 40 years old, can still be applied today and in totally different environments. Modern monolithic applications are comparable to monolithic kernels: if any of their components fail, the entire system is affected, which, translated into Node.js terms, means that all the services are part of the same code base and run in a single process (when not cloned).

Figure 12.9 shows an example monolithic architecture:

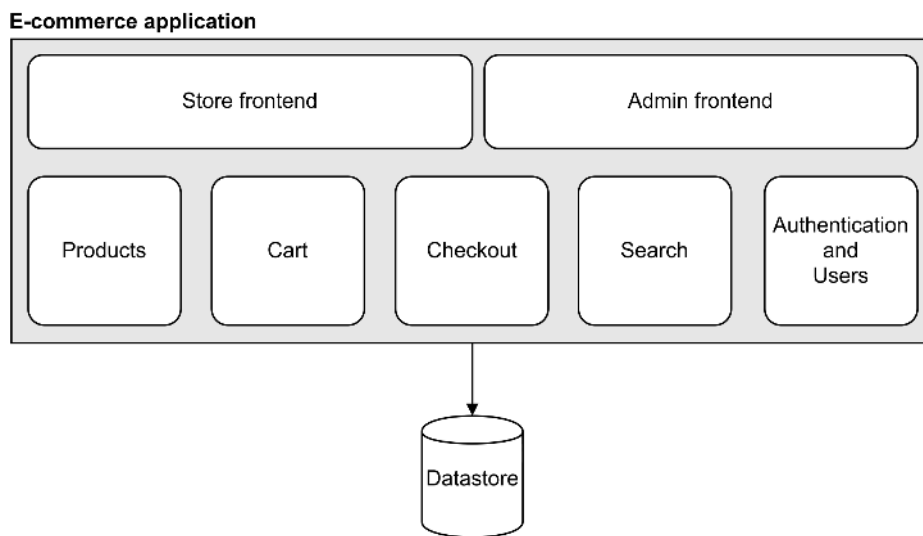


Figure 12.9: Example of a monolithic architecture

Figure 12.9 shows the architecture of a typical e-commerce application. Its structure is modular: we have two different frontends, one for the main store and another for the administration interface. Internally, we have a clear separation of the services implemented by the application. Each service is responsible for a specific portion of the application business logic: **Products**, **Cart**, **Checkout**, **Search**, and **Authentication and Users**. However, the preceding architecture is monolithic since every module is part of the same codebase and runs as part of a single application. A failure in any of its components can potentially tear down the entire online store.

Another common challenge in monolithic architectures is how easily tight coupling can creep in between modules. Since all components share the same codebase and runtime environment, the boundaries between services are often not strictly enforced. For instance, if the **Checkout** module needs to update product availability (responsibility of the **Products** module), a developer might choose the simplest path: directly querying the product database from within the Checkout code. While this might work in the short term, it blurs the line between responsibilities and breaks the separation of concerns between the two modules. Over time, these shortcuts accumulate, increasing the interdependence between modules and making the system more fragile and difficult to maintain and evolve.

High coupling is one of the most common reasons that monoliths become difficult to scale. When modules depend too much on each other, even small changes can ripple through the system in unpredictable ways. Every dependency becomes a potential liability. Over time, the codebase begins to resemble a Jenga tower: removing or modifying one block can destabilize the whole structure. As a result, teams often try to mitigate these risks by introducing stricter conventions, development guidelines, and review processes to avoid accidental breakage.

In short, while monolithic systems can start simple and modular, their tight internal connections can make them harder to evolve, especially as the application and the team behind it grow.

The microservice architecture

One of the most important patterns for building large-scale Node.js systems is this: do not build large-scale Node.js systems! It may sound like a joke, but the point is serious. One of the best ways to scale in capacity and complexity is to avoid making the system too large in the first place. The alternative lies in the Y-axis of the Scale Cube: break the system down by service and functionality. Instead of one big application, create smaller, focused components, each responsible for a specific slice of functionality. This aligns with the Unix philosophy and Node.js's core principle of “make each program do one thing well,” making services easier to understand, scale, and maintain. The most common form of this approach is the **microservice** architecture, where a set of small, self-contained services replaces a single monolith. “Micro” suggests small, but not to the extreme. Splitting into too many tiny services often causes more problems than it solves. There is no strict size rule; the goal is to achieve **loose coupling**, **high cohesion**, and manageable **integration complexity**. A good microservice encapsulates a meaningful piece of business functionality, is independently deployable, and communicates clearly with other services.

An example of a microservice architecture

Let's now see what the monolithic e-commerce application would look like using a microservice architecture:

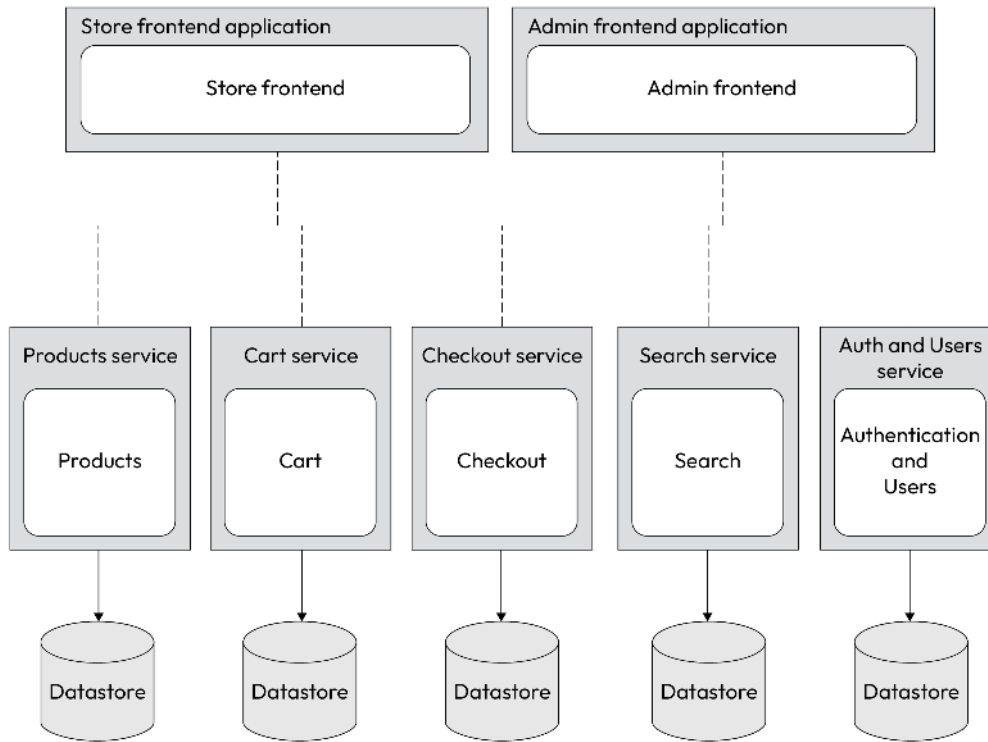


Figure 12.10: An example implementation of an e-commerce system using the microservice architecture

As we can see from *Figure 12.10*, each fundamental component of the e-commerce application is now a self-sustaining and independent entity, living in its own context, with its own database. In practice, they are all independent applications exposing a set of related services.

The **data ownership** of a service is an important characteristic of the microservice architecture. This is why the database also must be split to maintain the proper level of isolation and independence. If a unique shared database is used, it would become much easier for the services to work together; however, this would also introduce a coupling between the services (based on data), nullifying some of the advantages of having different applications.

The dashed lines connecting all the nodes tell us that, in some way, they must communicate and exchange information for the entire system to be fully functional. As the services do not share the same database, there is more communication involved to maintain the consistency of the whole system. For example, the **Checkout** service needs to know some information about **Products**, such as the price and restrictions on shipping, and at the same time, it needs to update the data stored in the **Products** service, such as the product's availability when the checkout is complete. In *Figure 12.10*, we tried to represent the way the nodes communicate generically. Surely, the most popular strategy is using web services, but as we will see later, this is not the only option.



Pattern (microservice architecture)

Split a complex application by creating several small, self-contained services.

Microservices: advantages and disadvantages

In this section, we are going to highlight some of the advantages and disadvantages of implementing a microservice architecture. As we will see, this approach promises to bring a radical change in the way we develop our applications, revolutionizing the way we see scalability and complexity, but on the other hand, it introduces new nontrivial challenges.



Martin Fowler wrote a great article about microservices that you can find at nodejssdp.link/microservices.

Every service is expendable

The main technical advantage of having each service living in its own application context is that crashes do not propagate to the entire system. The goal is to build truly independent services that are smaller, easier to change, or can even be rebuilt from scratch. If, for example, the **Checkout** service of our e-commerce application suddenly crashes because of a serious bug, the rest of the system would continue to work as normal. Some functionality may be affected (for example, the ability to purchase a product), but the rest of the system would continue to work.

Also, imagine if we suddenly realized that the database or the programming language we used to implement a component was not a good design decision. In a monolithic application, there would be very little we could do to change things without affecting the entire system. Instead, in a microservice architecture, we could more easily reimplement the entire service from scratch, using a different database or platform, and the rest of the system would not even notice it, as long as the new implementation maintains the same interface to the rest of the system.

Reusability across platforms and languages

Splitting a big monolithic application into many small services allows us to create independent units that can be reused much more easily. **Elasticsearch** (nodejsdp.link/elasticsearch) is a great example of a reusable search service. **ORY** (nodejsdp.link/ory) is another example of a reusable open-source technology that provides a complete authentication and authorization service that can be easily integrated into a microservice architecture.

The main advantage of the microservice approach is that the level of information hiding is usually much higher compared to monolithic applications. This is possible because the interactions usually happen through a remote interface such as a web API or a message broker, which makes it much easier to hide implementation details and shield the client from changes in the way the service is implemented or deployed. For example, if all we have to do is invoke a web service, we are shielded from the way the infrastructure behind it is scaled, from what programming language it uses, from what database it uses to store its data, and so on. All these decisions can be revisited and adjusted as needed, with potentially no impact on the rest of the system.

A way to scale the application

Going back to the scale cube, it's clear that microservices are equivalent to scaling an application along the Y-axis, so it's already a solution for distributing the load across multiple machines. Also, we should not forget that we can combine microservices with the other two dimensions of the cube to scale the application even further. For example, each service could be cloned to handle more traffic, and the interesting aspect is that they can be scaled independently, allowing better resource management.

At this point, it would look like microservices are the solution to all our problems. However, this is far from being true. Let's see the challenges we face using microservices.

The challenges of microservices

Having more nodes to manage introduces a higher complexity in terms of integration, deployment, and code sharing: it fixes some of the pains of traditional architectures, but it also opens many new questions. How do we make the services interact? How can we keep sanity with deploying, scaling, and monitoring such a high number of applications? How can we share and reuse code between services?

Fortunately, cloud services and modern DevOps methodologies can provide some answers to those questions, and using Node.js can help a lot. Its module system is a perfect companion to share code between different projects. Node.js was made to be a node in a distributed system, such as those of a microservice architecture.

In the following sections, we will introduce some integration patterns that can help with managing and integrating services in a microservice architecture.

Integration patterns in a microservice architecture

One of the toughest challenges of microservices is connecting all the nodes to make them collaborate. For example, the **Cart** service of our e-commerce application would make little sense without some products to add, and the **Checkout** service would be useless without a list of products to buy (a cart).

As we already mentioned, there are also other factors that necessitate an interaction between the various services. For example, the **Search** service must know which products are available and must also ensure it keeps its information up to date. The same can be said about the **Checkout** service, which must update the information about product availability when a purchase is completed.

When designing an integration strategy, it's also important to consider the coupling that it's going to introduce between the services in the system. We should not forget that designing a distributed architecture involves the same practices and principles we use locally when designing a module or subsystem. Therefore, we also need to take into consideration properties such as the reusability and extensibility of the service.

The API proxy

The first pattern we are going to show makes use of an **API proxy** (also commonly identified as an **API gateway**), a server that proxies the communications between a client and a set of remote APIs. In a microservice architecture, its main purpose is to provide a single access point for multiple API endpoints, but it can also offer load balancing, caching, authentication, and traffic limiting, all of which are features that prove to be very useful to implement a solid API solution.

This pattern should not be new to us since we already saw it in action in this chapter when we built the custom load balancer with `http-proxy` and `consul`. For that example, our load balancer was exposing only two services, and then, thanks to a service registry, it was able to map a URL path to a service, and hence to a list of servers. An API proxy works in the same way; it is essentially a reverse proxy and often also a load balancer, specifically configured to handle API requests. *Figure 12.11* shows how we can apply such a solution to our e-commerce application:

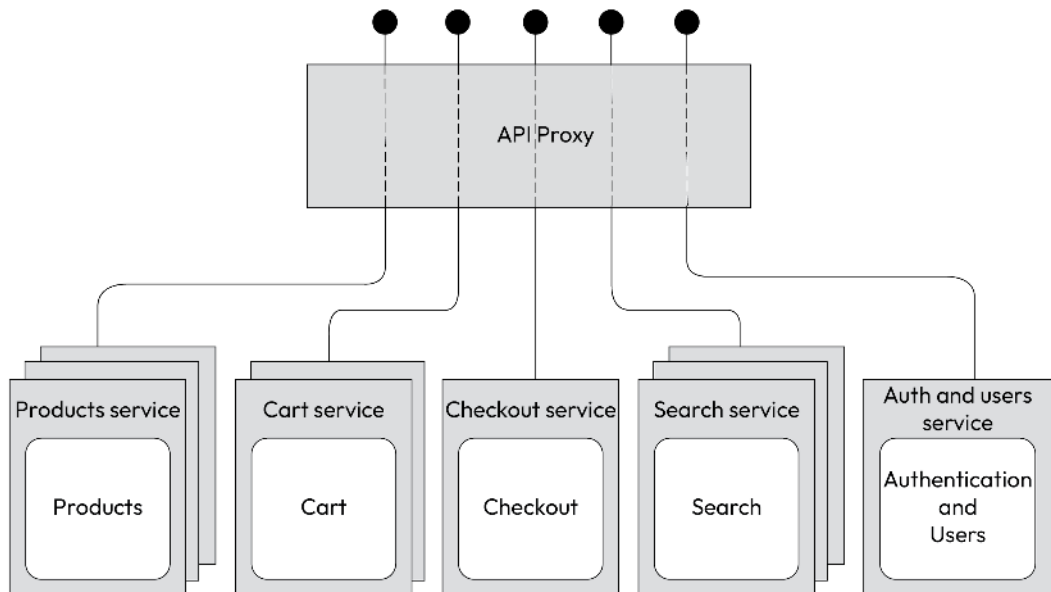


Figure 12.11: Using the API Proxy pattern in an e-commerce application

From the preceding diagram, it should be clear how an API proxy can hide the complexity of its underlying infrastructure. This is handy in a microservice infrastructure, as the number of nodes may be high, especially if each service is scaled across multiple machines. The integration achieved by an API proxy is therefore only structural since there is no semantic mechanism. It simply provides a familiar monolithic view of a complex microservice infrastructure.

Since the API Proxy pattern essentially abstracts the complexity of connecting to all the different APIs in the system, it might also allow for some freedom to restructure the various services. Maybe, as your requirements change, you will need to split an existing microservice into two or more decoupled microservices or, conversely, you might realize that, in your business context, it's better to join two or more services together. In both cases, the API Proxy pattern will allow you to make all the necessary changes with potentially no impact on the upstream systems accessing the data through the proxy.



The ability to enable incremental change in an architecture over time is a very important characteristic in modern distributed systems. If you are interested in studying this broad subject in greater depth, we recommend the book *Building Evolutionary Architectures*: nodejsdp.link/evolutionary-architectures.

API orchestration

The pattern we are going to describe next is probably the most natural and explicit way to integrate and compose a set of services, and it's called **API orchestration**. Daniel Jacobson, VP of Engineering for the Netflix API, in one of his blog posts (nodejsdp.link/orchestration-layer), defines API orchestration as follows:



*"An **API Orchestration Layer (OL)** is an abstraction layer that takes generically-modeled data elements and/or features and prepares them in a more specific way for a targeted developer or application."*

The "generically-modeled elements and/or features" fit the description of a service in a microservice architecture perfectly. The idea is to create an abstraction to connect those bits and pieces to implement new services specific to a particular application.

Let's see an example using the e-commerce application. Refer to *Figure 12.12*:

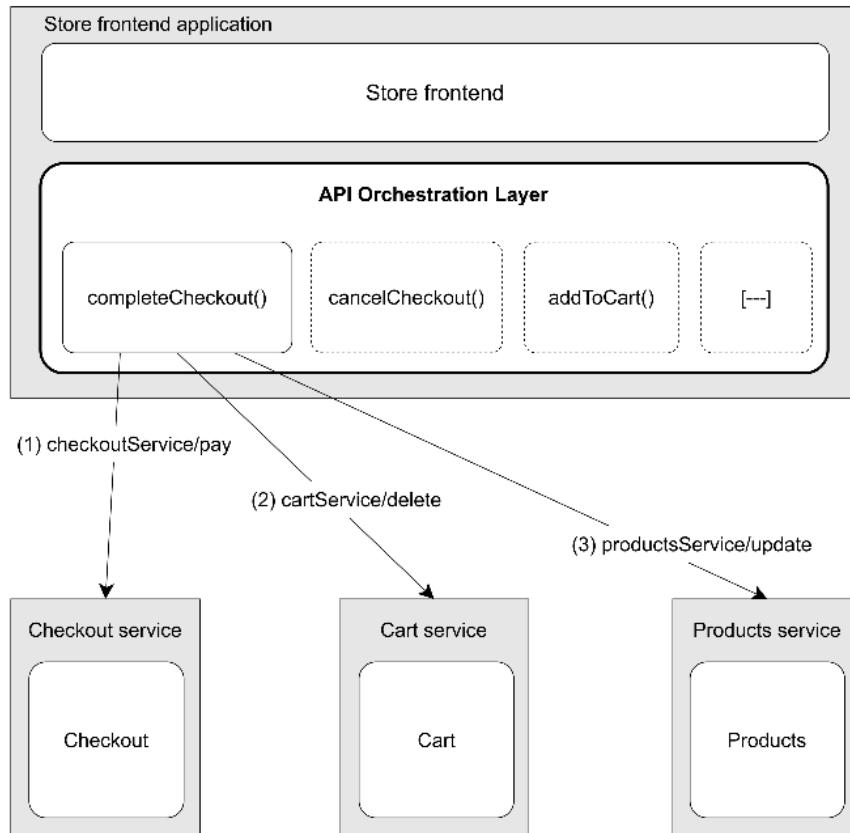


Figure 12.12: An example usage of an orchestration layer to interact with multiple microservices

Figure 12.12 shows how the store frontend application uses an orchestration layer to build more complex and specific features by composing and orchestrating existing services. The described scenario takes, as an example, a hypothetical `completeCheckout()` service that is invoked the moment a customer clicks the **Pay** button at the end of the checkout.

The figure shows how `completeCheckout()` is a composite operation made of three different steps:

1. First, we complete the transaction by invoking `checkoutService/pay`.
2. Then, when the payment is successfully processed, we need to tell the **Cart** service that the items were purchased and that they can be removed from the cart. We do that by invoking `cartService/delete`.
3. Also, when the payment is complete, we need to update the availability of the products that were just purchased. This is done through `productsService/update`.

As we can see, we took three operations from three different services, and we built a new API that coordinates the services to maintain the entire system in a consistent state.

Another common operation performed by the API orchestration layer is **data aggregation**, or in other words, combining data from different services into a single response. Imagine we wanted to list all the products contained in a cart. In this case, the orchestration would need to retrieve the list of product IDs from the **Cart** service and then retrieve the complete information about the products from the **Products** service. The ways in which we can combine and coordinate services are infinite, but the important pattern to remember is the role of the orchestration layer, which acts as an abstraction between several services and a specific application.

The orchestration layer is a great candidate for further functional splitting. It is, in fact, very common to have it implemented as a dedicated, independent service, in which case it takes the name of **API Orchestrator**. This practice is perfectly in line with the microservice philosophy.

Figure 12.13 shows this further improvement of our architecture:

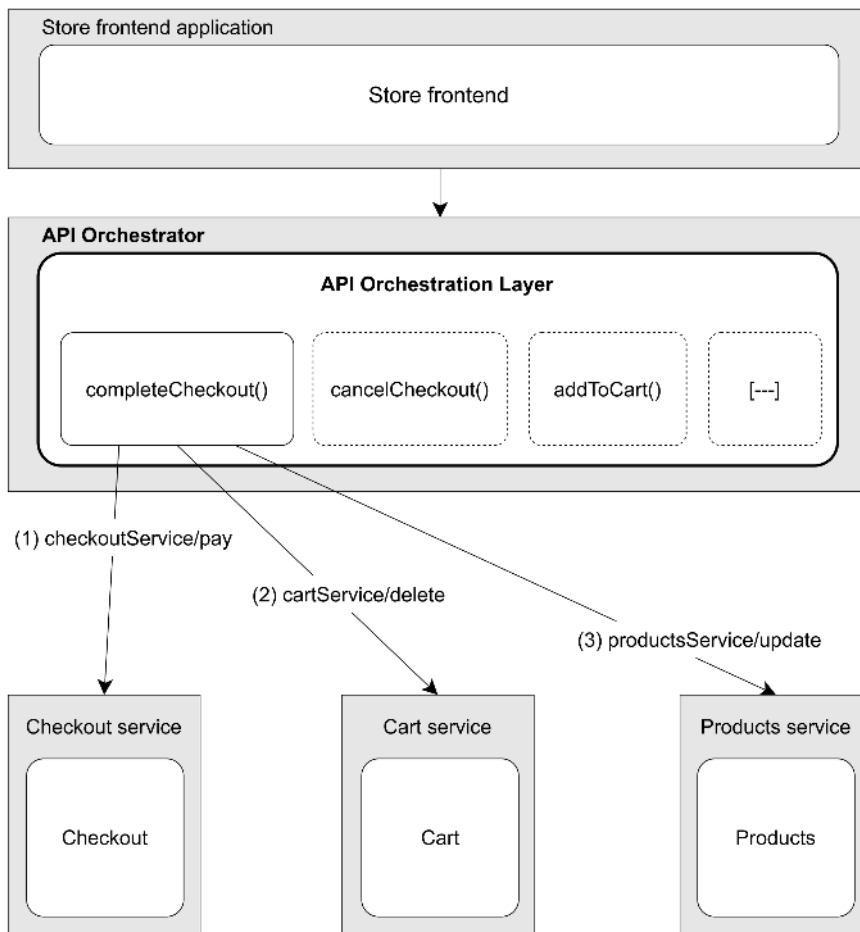


Figure 12.13: An application of the API Orchestrator pattern for our e-commerce example

Creating a standalone orchestrator, as shown in the previous figure, can help in decoupling the client application (in our case, **Store frontend**) from the complexity of the microservice infrastructure. This is similar to the API proxy, but there is a crucial difference: an orchestrator performs a *semantic* integration of the various services; it's not just a naïve proxy, and it often exposes an API that is different from the one exposed by the underlying services.



While API orchestration focuses on combining multiple APIs into a single experience, it's not always the best fit, especially when different frontends have distinct requirements. This is where the **Backend for Frontend (BFF)** pattern comes in. A BFF is a dedicated backend designed specifically for a single frontend (such as a web app, mobile app, or smartwatch). Rather than having a one-size-fits-all API, each frontend gets its own backend that delivers only the data it needs, in the exact format it expects. This keeps the frontend development experience as simple as possible. Developers can work with APIs that are already crafted to fit the specific needs of their interface, reducing complexity, avoiding the over-fetching or under-fetching of data, and improving performance. BFFs are particularly helpful when frontend teams operate independently or when client-specific optimization is a priority.

Integration with a message broker

The Orchestrator pattern gave us a mechanism to integrate the various services in an explicit way. This has both advantages and disadvantages. It is easy to design, easy to debug, and easy to scale, but unfortunately, it requires complete knowledge of the underlying architecture and how each service works. If we were talking about objects instead of architectural nodes, the orchestrator would be an anti-pattern called a **god object**, which defines an object that knows and does too much, which usually results in high coupling, low cohesion, but most importantly, high complexity.

The pattern we are now going to show tries to distribute, across the services, the responsibility of synchronizing the information of the entire system. However, the last thing we want to do is create direct relationships between services, which would result in high coupling and a further increase in the complexity of the system, due to the increasing number of interconnections between nodes. The goal is to keep every service decoupled: every service should be able to work, even without the rest of the services in the system or in combination with new services and nodes.

The solution is to use a message broker, a system capable of decoupling the sender from the receiver of a message, allowing us to implement a centralized Publish/Subscribe pattern. This is, in practice, an implementation of the Observer pattern for distributed systems. We will talk more about this pattern later in *Chapter 13, Messaging and Integration Patterns*.

Figure 12.14 shows an example of how this applies to the e-commerce application:

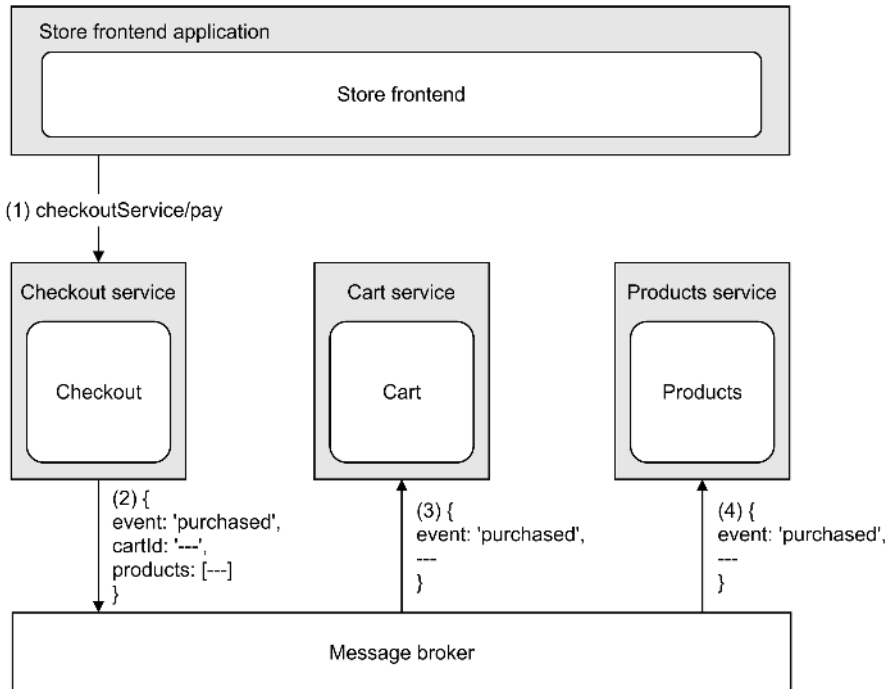


Figure 12.14: Using a message broker to distribute events in our e-commerce application

As we can see from Figure 12.14, the client of the **Checkout** service, which is the frontend application, does not need to carry out any explicit integration with the other services.

All it must do is invoke `checkoutService/pay` to complete the checkout process and take the money from the customer; all the integration work happens in the background:

1. **Store frontend** invokes the `checkoutService/pay` operation on the **Checkout** service.
2. When the operation completes, the **Checkout** service generates an event, attaching the details of the operation, that is, the `cartId` and the list of products that were just purchased. The event is published to the message broker. At this point, the **Checkout** service does not know who is going to receive the message.
3. The **Cart** service is subscribed to the broker, so it's going to receive the `purchased` event that was just published by the **Checkout** service. The **Cart** service reacts by removing the cart identified with the ID contained in the message from its database.
4. The **Products** service was subscribed to the message broker as well, so it receives the same `purchased` event. It then updates its database based on this new information, adjusting the availability of the products included in the message.

This whole process happens without any explicit intervention from external entities such as an orchestrator. The responsibility of spreading the knowledge and keeping information in sync is distributed across the services themselves. There is no *god* service that must know how to move the gears of the entire system, since each service oversees its own part of the integration.

The message broker is a fundamental element used to decouple the services and reduce the complexity of their interaction. It might also offer other interesting features, such as persistent message queues and guaranteed ordering of the messages. We will talk more about this in the next chapter.

Summary

In this chapter, we explored how to design Node.js architectures that scale in both capacity and complexity. We saw that scaling is not only about handling more traffic or reducing response time, but also about improving availability and fault tolerance. These goals often align, and in Node.js scaling early can be a smart move thanks to its efficiency and low resource requirements. The Scale Cube showed us three dimensions of scaling. We focused on the *X* and *Y* axes, which led us to two essential patterns: load balancing and microservices. You learned how to run multiple instances of a Node.js application, distribute traffic across them, and use this setup for fail tolerance and zero-downtime restarts. We covered dynamic and auto-scaled infrastructures, saw how a service registry can help, and explored solutions like Nginx, Consul, and Kubernetes. We then moved to the *Y*-axis, breaking applications into services to form a microservice architecture. Microservices make it easier to distribute load and split complexity, but they shift challenges toward integrating independent services. We closed this section by examining architectural solutions for that integration. In the next and final chapter, we will complete our Node.js Design Patterns journey by exploring messaging patterns and advanced integration techniques for complex distributed architectures. You've come a long way, let's finish strong!

Exercises

- **12.1 A scalable book library:** Revisit the book library application we built in *Chapter 10, Universal JavaScript for Web Applications*, reconsidering it after what we have learned in this chapter. Can you make our original implementation more scalable? Some ideas might be to use the `cluster` module to run multiple instances of the server, making sure you handle failures by restarting workers that might accidentally die. Alternatively, why not try to run the entire application on Kubernetes?

- **12.2 Exploring the Z-axis:** Throughout this chapter, we did not show you any examples about how to shard data across multiple instances, but we explored all the necessary patterns to build an application that achieves scalability along the Z-axis of the scale cube. In this exercise, you are challenged to build a REST API that allows you to get a list of (randomly generated) people whose first name starts with a given letter. You could use a library such as *faker* ([nodejsdp.link/faker](https://www.npmjs.com/package/faker)) to generate a sample of random people, and then you could store this data in different JSON files (or different databases), splitting the data into three different groups. For instance, you might have three groups called A-D, E-P, and Q-Z. *Ada* will go in the first group, *Peter* in the second, and *Ugo* in the third. Now, you can run one or more instances of a web server for every group, but you should expose only one public API endpoint to be able to retrieve all the people whose name starts with a given letter (for instance, `/api/people/byFirstName/{letter}`). Hint: You could use just a load balancer and map all the possible letters to the respective backend of the instances that are responsible for the associated group. Alternatively, you could create an API orchestration layer that encodes the mapping logic and redirects the traffic accordingly. Can you also throw a service discovery tool into the mix and apply dynamic load balancing, so that groups receiving more traffic can scale as needed?
- **12.3 Music addiction:** Imagine you must design the architecture of a service like Spotify or Apple Music. Can you try to design this service as a collection of microservices by applying some of the principles discussed in this chapter? Bonus points if you can implement a minimal version of this idea with Node.js! If this turns out to be the next big start-up idea and makes you a millionaire, well... don't forget to thank the authors of this book! :)

13

Messaging and Integration Patterns

First, congratulations. If you've made it this far, you've explored some of the most powerful design patterns and architectural ideas in the Node.js ecosystem. You're not just learning Node.js, you've been progressing toward mastery, one pattern, one principle, and one chapter at a time.

This final chapter adds the missing piece that makes distributed systems truly work: **integration**. If scalability is about distributing systems, integration is what holds them together. In the previous chapter, we learned how to split applications across multiple processes and machines. To make those parts useful, they often need to communicate. There are two main ways to do that: use shared storage as a central source of truth, or exchange messages containing data, events, or commands. Messaging is what truly unlocks scalability and flexibility in distributed architectures.

Messages are everywhere: over the internet, between processes, inside applications, and even in operating systems. In distributed architectures, messaging refers to patterns, tools, and conventions for reliably exchanging information over a network. These systems can be centralized with a broker or peer-to-peer, one-way or request/reply, and may use queues for reliability.

One of the most influential works on this topic is *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf, which describes over 60 patterns. Here, we'll focus on the most essential ones and a few modern alternatives, always with a Node.js perspective. Here's what you'll learn:

- The fundamentals of messaging systems
- The Publish/Subscribe pattern
- Task distribution and pipeline patterns
- Request/reply communication models

Let's dive in and wrap up our Node.js Design Patterns journey with one of the most powerful and practical areas of modern architecture.

Fundamentals of a messaging system

When talking about messages and messaging systems, there are four fundamental elements to take into consideration:

- The direction of the communication, which can be one-way only or a request/reply exchange
- The purpose of the message, which also determines its content
- The timing of the message, which can be sent and received in context (synchronously) or out of context (asynchronously)
- The delivery of the message, which can happen directly or via a broker

In the sections that follow, we are going to formalize these aspects to provide a base for our later discussions.

One way versus request/reply patterns

The most fundamental aspect of a messaging system is the direction of the communication, which often also determines its semantics.

The simplest communication pattern is when the message is pushed *one way* from a source to a destination; this is a trivial situation, and it doesn't need much explanation:

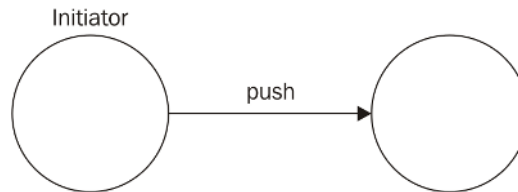


Figure 13.1 – One-way communication

A typical example of **one-way communication** is an email or a web server that sends a message to a connected browser using *server-sent events* (nodejsdp.link/sse), or a system that distributes tasks to a set of workers.

On the other side, we have the **Request/Reply** exchange pattern, where the message in one direction is always matched (excluding error conditions) by a message in the opposite direction. A typical example of this exchange pattern is the invocation of a web service or sending a query to a database. The following diagram shows this simple and well-known scenario:

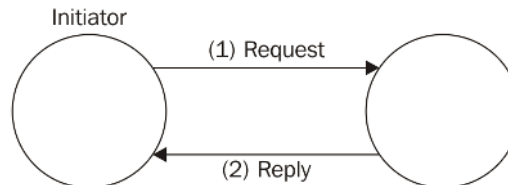


Figure 13.2 – Request/Reply message exchange pattern

The Request/Reply pattern might seem a trivial pattern to implement; however, as we will see later, it becomes more complicated when the communication channel is asynchronous or involves multiple nodes. Look at the example represented in the next diagram:

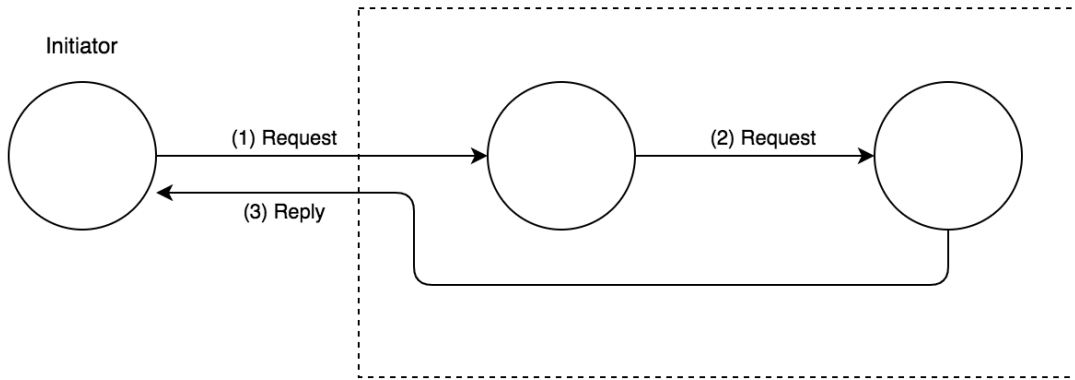


Figure 13.3 – Multi-node request/reply communication

With the setup shown in *Figure 13.3*, we can better appreciate the complexity of some Request/Reply patterns. If we consider the direction of the communication between any two nodes, we can surely say that it is one-way. However, from a global point of view, the initiator sends a request and in turn receives an associated response, even if from a different node. In these situations, what really differentiates a Request/Reply pattern from a bare one-way loop is the relationship between the request and the reply, which is kept in the initiator. The reply is usually handled in the same context as the request.

Message types

A **message** is essentially a means to connect different software components, and there are different reasons for doing so: it might be because we want to obtain some information held by another system or component, to execute operations remotely, or to notify some peers that something has just happened.

The message content will also vary depending on the reason for the communication. In general, we can identify three types of messages, depending on their purpose:

- Command messages
- Event messages
- Document messages

Command messages

You should already be familiar with the **command message** as it's essentially a serialized Command object (we learned about this in the *Command* section of *Chapter 9, Behavioral Design Patterns*).

The purpose of this type of message is to trigger the execution of an action or a task on the receiver. For this to be possible, the Command message must contain the essential information to run the task, which usually includes the name of the operation and a list of arguments.

The Command message can be used to implement **Remote Procedure Call (RPC)** systems or distributed computations, or can be more simply used to request some data. RESTful HTTP calls are simple examples of commands; each HTTP verb has a specific meaning and is associated with a precise operation: GET, to retrieve the resource; POST, to create a new one; PUT/PATCH, to update it; and DELETE, to destroy it.

Event messages

An **event message** is used to notify another component that something has occurred. It usually contains the *type* of the event and sometimes also some details such as the context, the subject, or the actor involved.

In web development, we're using event messages when we, for example, leverage WebSockets (or server-sent events) to send real-time updates from the server to the client. These updates let users know that some data has changed or that an action has taken place elsewhere in the system, and this information can be used on the client side to re-render the view and automatically display the most recent state.

To make this more concrete, imagine a chat application. When a user sends a message in a specific channel, the system generates an event that describes what happened: who sent the message, to which channel, and at what time. This event can then be broadcast to all other connected users, so everyone stays in sync. Another common example is an e-commerce dashboard that displays live sales data. When a customer completes a purchase, the backend emits an event that captures details about the transaction. That event becomes a message the system can send to the dashboard, updating it in real time with the new sale.

Events are a powerful integration mechanism in distributed applications. They allow different components, services, or even entirely separate systems to stay aligned without needing to know too much about each other. This loose coupling is essential for building systems that scale, evolve, and recover gracefully.

Document messages

The **document message** is primarily meant to transfer data between components and machines. Unlike a command message, which tells the receiver what to do, a document message simply contains data (such as the results of a database query) with no instructions attached.

On the other hand, the main difference between a document message and an event message is the absence of an association with a particular occurrence of something that happened. Often, the replies to command messages are document messages, as they usually contain only the data that was requested or the result of an operation.

Now that we know how to categorize the semantics of a message, let's learn about the semantics of the communication channel used to move our messages around.

Push versus pull delivery semantics

In distributed systems, message delivery falls into two key paradigms: **pull**, where the consumer requests data, and **push**, where the producer sends data proactively. Knowing the difference between these models is essential for systems that must balance performance, scalability, and reliability.

Pull delivery (consumer-initiated)

This model relies on clients or consumers actively requesting information from a provider. The consumer controls when to fetch data, reducing complexity on the producer side.

Here are some practical examples:

- **Browsers sending HTTP requests for web pages:** The browser asks a server for a web page, polling the data from it only when the user initiates the request.
- **Applications querying databases with SQL:** The application requests specific data from the database, which returns only what is asked for at that moment.
- **Monitoring systems polling services at intervals:** The monitoring tool regularly asks a service for its status or metrics instead of waiting for notifications.
- **Cloud queue systems (e.g., AWS SQS):** The consumer explicitly requests messages from the queue, sometimes using **long polling** so the request stays open until a message is available, reducing unnecessary repeated calls.

Push delivery (producer-initiated)

In this model, the data source or messaging system proactively sends updates to consumers. Information is delivered in real time, without waiting for consumer requests.

Let's see some examples:

- **Webhooks triggered by external services:** An external system sends data to your application as soon as an event occurs, without waiting for your app to request it.
- **Message Queues (MQs) – RabbitMQ pushing messages to subscribers:** The broker delivers messages to subscribed consumers as they become available, so the consumers do not need to ask for them explicitly.
- **Real-time WebSocket or server-sent event feeds:** The server continuously sends updates over an open connection, keeping clients in sync without repeated requests.
- **Database change streams and triggers delivering immediate updates:** The database notifies connected applications as soon as relevant data changes, allowing them to react instantly.

Choosing between pull and push delivery

Choosing between push and pull delivery depends on the type of updates, how quickly they must arrive, and the operational complexity you can manage.

Push delivery is ideal when information must reach consumers immediately, as in chat, live prices, or alerts. It delivers low latency but adds complexity in managing connections, retries, acknowledgments, and backpressure, and it can be resource-intensive under high load.

Pull delivery gives control to consumers, making it simpler, easier to debug, and more predictable in resource usage. It can be cost-effective, especially with caching, but it adds delay, and frequent polling can increase server load.

Many systems use hybrids: **long polling** for near real-time responsiveness, or **push notifications** to signal changes, followed by pull for full data.

The general rule is to use push for immediacy if you can handle the cost, pull for simplicity when small delays are acceptable, or a combination of both for flexibility.

Asynchronous messaging, queues, and streams

By now in the book, you're familiar with asynchronous operations and their benefits. These same principles also apply to communication between services and components.

Think of **synchronous communication** like a phone call: both participants need to be connected at the same time, on the same channel, ready to speak and listen. If one party hangs up, the communication is over, and you can't easily handle another call without ending the current one. This tight coupling means your system needs more coordination and can only handle one interaction at a time per channel.

On the other hand, **asynchronous communication** is more like sending an SMS. You can send a message without knowing whether the recipient is currently holding their phone in their hand, staring at the screen. It's up to them to decide when to check their messages and what to do with them. They might respond instantly, hours later, or not at all. You can send multiple messages to different people without waiting for replies, and responses can arrive in any order.

One of the key features of asynchronous communication is **decoupling**: the sender and receiver don't need to be available at the same time. Messages can be stored and delivered whenever the receiver is ready. This is made possible by an MQ, a component that temporarily holds messages until they can be processed. It acts as a buffer between the **producer** (sender) and the **consumer** (receiver), allowing the system to absorb load spikes and recover gracefully from interruptions.

Let's look at the following diagram:

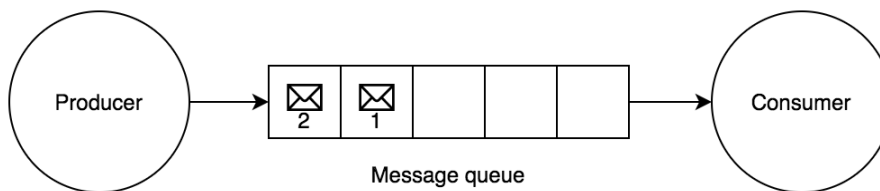


Figure 13.4 – A message queue

If the consumer is busy, disconnected, or crashes, the queue will continue to accept and store messages. Once the consumer is back online, the messages are delivered in the order they were received. Depending on the architecture, the queue may live on the producer's side, be shared between the producer and consumer (as in peer-to-peer setups), or be hosted by a third-party **message broker** that mediates the communication.

Another important concept is the **log**, or **stream**: a data structure that serves a similar purpose to a queue but with a different set of trade-offs.

A stream is an append-only, durable sequence of messages that consumers can read from. But unlike a queue, where messages are removed once consumed, messages in a stream remain accessible even after being processed. Consumers can read new messages as they arrive or go back and read historical data at any time.

This difference has significant implications:

- **Queues are typically point to point:** Each message is delivered to a single consumer
- **Streams are multi-subscriber friendly:** Many consumers can independently read the same messages, starting from different positions or time points

This makes streams ideal for systems where multiple components need to react to the same events, or when you need to replay past messages for debugging, analytics, or rebuilding state.

Figure 13.5 gives you an idea of the structure of a stream compared to that of a message queue:

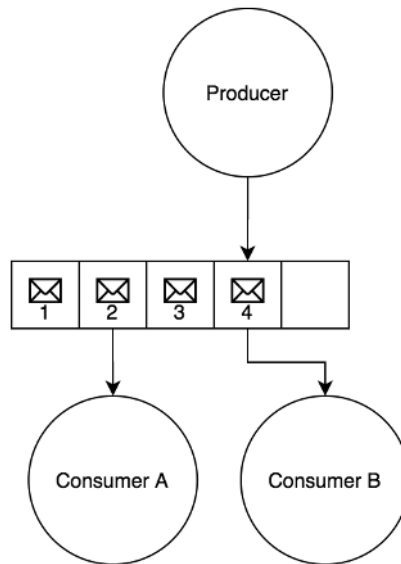


Figure 13.5 – A stream

We'll revisit these two models (queues and streams) in more depth later in the chapter, with hands-on examples showing when and how to use each.

The last key element to consider in a messaging system is how the nodes are connected: either directly to each other in a peer-to-peer fashion, or through an intermediary that handles the communication.

Peer-to-peer or broker-based messaging

Messages can be delivered directly to the receiver in a **peer-to-peer** fashion, or through a centralized intermediary system called a **message broker**. The main role of the broker is to decouple the receiver of the message from the sender. The following diagram shows the architectural difference between the two approaches:

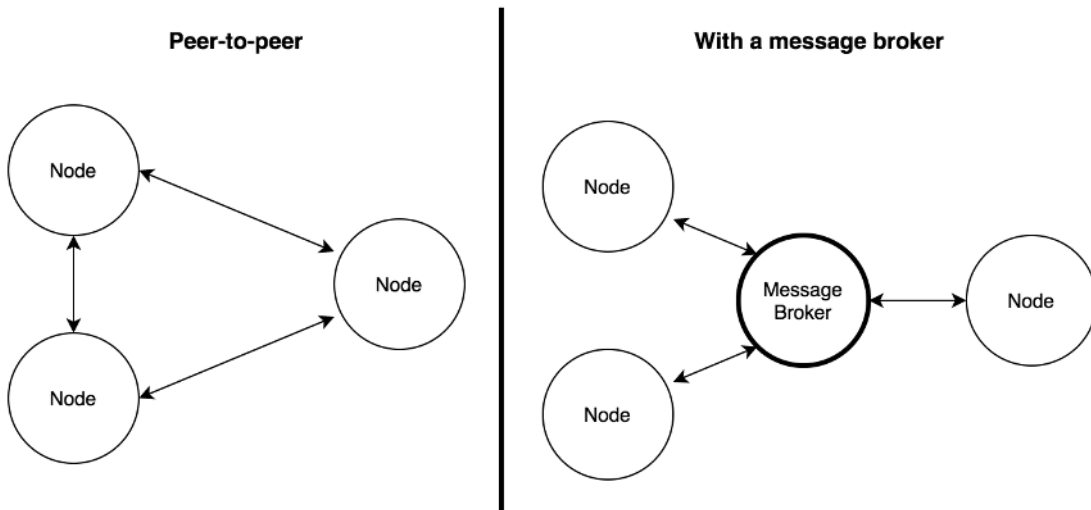


Figure 13.6 – Peer-to-peer communication versus message brokering

In a peer-to-peer architecture, every node is directly responsible for the delivery of the message to the receiver. This implies that the nodes must know the address and port of the receiver, and they have to agree on a protocol and message format. The broker eliminates these complexities from the equation: each node can be totally independent and can communicate with an unspecified number of peers without directly knowing their details.

A broker can also act as a bridge between different communication protocols. For example, the popular RabbitMQ broker (nodejsdp.link/rabbitmq) supports **Advanced Message Queuing Protocol (AMQP)**, **Message Queue Telemetry Transport (MQTT)**, and **Simple/Streaming Text Orientated Messaging Protocol (STOMP)**, enabling multiple applications that support different messaging protocols to interact.



MQTT (nodejsdp.link/mqtt) is a lightweight messaging protocol specifically designed for machine-to-machine communication (such as the Internet of Things). AMQP (nodejsdp.link/amqp) is a more complex messaging protocol, designed to be an open-source alternative to proprietary messaging middleware. STOMP (nodejsdp.link/stomp) is a lightweight text-based protocol, which comes from “the HTTP school of design.” All three are application layer protocols and are based on TCP/IP.

Besides the advantages in terms of decoupling and interoperability, a broker can offer additional features such as persistent queues, routing, message transformations, and monitoring, without mentioning the broad range of messaging patterns that many brokers support out of the box.

Of course, nothing prevents us from implementing all these features using a peer-to-peer architecture, but unfortunately, there is much more effort involved. Nonetheless, there might be different reasons for choosing a peer-to-peer approach instead of a broker:

- By removing the broker, we are removing a single point of failure from the system
- A broker must be scaled, while in a peer-to-peer architecture, we only need to scale the single nodes of the application
- Exchanging messages without intermediaries can greatly reduce the latency of the communication

By using a peer-to-peer messaging system, we can have much more flexibility and power because we are not bound to any technology, protocol, or architecture.

As always, there's no one-size-fits-all solution. It ultimately comes down to trade-offs. Understanding your specific context and requirements will help you decide which approach best suits your use case.

Now that we know the basics of a messaging system, let's explore some of the most important messaging patterns. Let's start with the Publish/Subscribe pattern.

Publish/Subscribe pattern

Publish/Subscribe (often abbreviated to Pub/Sub) is probably the best-known one-way messaging pattern. We should already be familiar with it, as it's nothing more than a distributed Observer pattern. As in the case of Observer, we have a set of *subscribers* registering their interest in receiving a specific category of messages. On the other side, the *publisher* produces messages that are distributed across all the relevant subscribers. *Figure 13.7* shows the two main variants of the Pub/Sub pattern; the first is based on a peer-to-peer architecture, and the second uses a broker to mediate the communication:

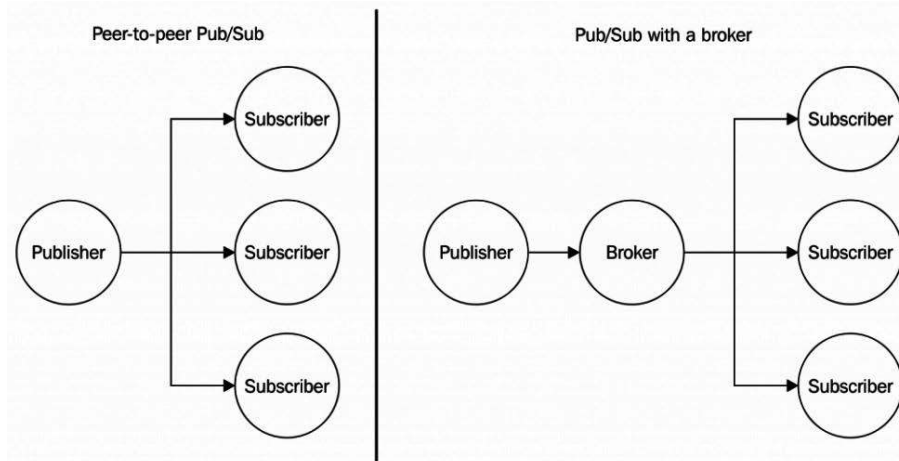


Figure 13.7 – Publish/Subscribe messaging pattern

What makes Pub/Sub so special is the fact that the publisher doesn't know in advance who the recipients of the messages are. As we said, it's the subscriber that must register its interest to receive a particular message, allowing the publisher to work with an unspecified number of receivers. In other words, the two sides of the Pub/Sub pattern are *loosely coupled*, which makes this an ideal pattern to integrate the nodes of an evolving distributed system.

The presence of a broker further improves the decoupling between the nodes of the system because the subscribers interact only with the broker, not knowing which node is the publisher of a message. As we will see later, a broker can also provide a message queuing system, allowing reliable delivery even in the presence of connectivity problems between the nodes.

Now, let's work on an example to demonstrate this pattern.

Building a minimalist real-time chat application

To show how the Pub/Sub pattern simplifies communication in distributed systems, we will build a simple browser-based real-time chat using WebSocket and Node.js. It will be an anonymous chat with no authentication, authorization, or session management, so we can focus on the core concept: how Pub/Sub allows multiple clients to exchange messages efficiently at scale. We will start with a basic chat, then run multiple server instances to see how it behaves when scaled horizontally. Finally, we will add a messaging system to connect all instances and share messages in real time, demonstrating the true strength of the Pub/Sub pattern.

Implementing the server side

Now, let's take this challenge one step at a time. Let's first build a basic chat application, then we'll scale it to multiple instances.

To implement the real-time capabilities of a typical chat application, we will rely on the `ws` package (nodejsdp.link/ws), which is a pure WebSocket implementation for Node.js. Implementing real-time applications in Node.js is pretty simple, and the code we are going to write will confirm this assumption. So, let's create the server side of our chat application in a file called `index.js`:

```
import { createServer } from 'node:http'
import { WebSocketServer } from 'ws' // v8.18.2
import staticHandler from 'serve-handler' // v6.1.6

// serve static files
const server = createServer((req, res) => { // 1
  return staticHandler(req, res, { public: 'web' })
})

const wss = new WebSocketServer({ server }) // 2
wss.on('connection', client => {
  console.log('Client connected')
  client.on('message', msg => { // 3
```

```
    console.log(`Message: ${msg}`)
    broadcast(msg)
  })
})

function broadcast(msg) { // 4
  for (const client of wss.clients) {
    if (client.readyState === WebSocket.OPEN) {
      client.send(msg)
    }
  }
}

server.listen(process.argv[2] || 8080)
```

That's it! That's all we need to implement the server-side component of our chat application. This is how it works:

1. We first create an HTTP server and forward every request to a special handler (`nodejsdp.link/serve-handler`), which will take care to serve all the static files from the web directory. This is needed to access the client-side resources of our application (for example, HTML, JavaScript, and CSS files).
2. We then create a new instance of the WebSocket server, and we attach it to our existing HTTP server. Next, we start listening for incoming WebSocket client connections by attaching an event listener for the connection event.
3. Each time a new client connects to our server, we start listening for incoming messages. When a new message arrives, we broadcast it to all the connected clients.
4. The `broadcast()` function is a simple iteration over all the known clients, where the `send()` function is invoked on each connected client.

This is all we need for now! Of course, the server that we just implemented is very minimal and basic, but as we will see, it does its job.

Implementing the client side

Next, it's time to implement the client side of our chat application. This can be done with another compact and simple fragment of code, essentially a minimal HTML page with some basic JavaScript code. Let's create this page in a file named `web/index.html`, as follows:

```
<body>
  <div>
    <div id="messages">
      <!-- Messages will be added here dynamically -->
    </div>
    <div>
```

```

    <form id="msgForm">
      <textarea
        id="msgBox"
        placeholder="Type a message..."
        rows="1"
      ></textarea>
      <button type="submit" id="sendButton">Send</button>
    </form>
  </div>
</div>

<script>
  const messagesArea = document.getElementById("messages")
  const messageInput = document.getElementById("msgBox")
  const sendButton = document.getElementById("sendButton")
  const form = document.getElementById("msgForm")

  // WebSocket connection
  const ws = new WebSocket(`ws://${window.document.location.host}`)

  // Receive messages from WebSocket
  ws.onmessage = async function (message) {
    const content = await message.data.text()

    // Create received message element
    const messageDiv = document.createElement("div")
    const messageContent = document.createElement("div")
    messageContent.textContent = content
    messageDiv.appendChild(messageContent)
    const messageTime = document.createElement("div")
    messageTime.textContent = new Date().toLocaleTimeString([], {
      hour: "2-digit",
      minute: "2-digit",
    })
    messageDiv.appendChild(messageTime)

    // Add to messages area
    messagesArea.appendChild(messageDiv)
  }

  // Send message function

```

```
function sendMessage() {
  const content = messageInput.value.trim()
  if (!content || ws.readyState !== WebSocket.OPEN) return

  // Send via WebSocket
  ws.send(content)

  // Clear input
  messageInput.value = ""
  messageInput.style.height = "auto"
}

// Form submission handler
form.addEventListener("submit", (event) => {
  event.preventDefault()
  sendMessage()
})

// Enter key to send (Shift+Enter for new line)
messageInput.addEventListener("keydown", function (e) {
  if (e.key === "Enter" && !e.shiftKey) {
    e.preventDefault()
    sendMessage()
  }
})

// Initial scroll to bottom
messagesArea.scrollTop = messagesArea.scrollHeight
</script>
</body>
</html>
```



Note that here we have removed all the CSS styling and some extra piece of functionality (e.g., the auto-resizing textbox) for brevity. You can find the full version of this file in the book's code repository.

The HTML page we just created is intentionally simple web development. We use the native WebSocket object to connect to our Node.js server, and we listen for incoming messages, displaying each one in a new div as it arrives. To send messages, we use a basic form with a text input and a button. The code includes comments to guide you through each part of the implementation, so everything should be easy to understand just by reading through it.

In this example, we use `textContent` when displaying incoming messages:

```
messageContent.textContent = content
```

This is a deliberate and important choice. Unlike `innerHTML`, which parses the input as HTML and can execute malicious scripts, `textContent` treats the input strictly as text. This simple decision protects our chat application from **Cross-Site Scripting (XSS)** attacks (nodejsdp.link/xss).

If we had used `innerHTML`, a user could send a message as follows:

```
<script>alert('You've been hacked!')</script>
```

And the browser would execute that script when displaying the message. More malicious scripts could lead to data theft or other unauthorized actions. By using `textContent`, any HTML or script tags are displayed harmlessly as plain text. Always prefer `textContent` over `innerHTML` unless you explicitly trust and intend to render HTML content.

Please note that when stopping or restarting the chat server, the WebSocket connection is closed, and the client will not try to reconnect automatically (as we might expect from a production-grade application). This means that it is necessary to refresh the browser after a server restart to reestablish the connection (or implement a reconnection mechanism, which we will not cover here for brevity). Also, in this initial version of our app, the clients will not receive any message sent while they were not connected to the server.

Running and scaling the chat application

We can try to run our application immediately. Just launch the server with the following command:

```
node index.js 8080
```

Then, open a couple of browser tabs, or even two different browsers, point them at <http://localhost:8080>, and start chatting:

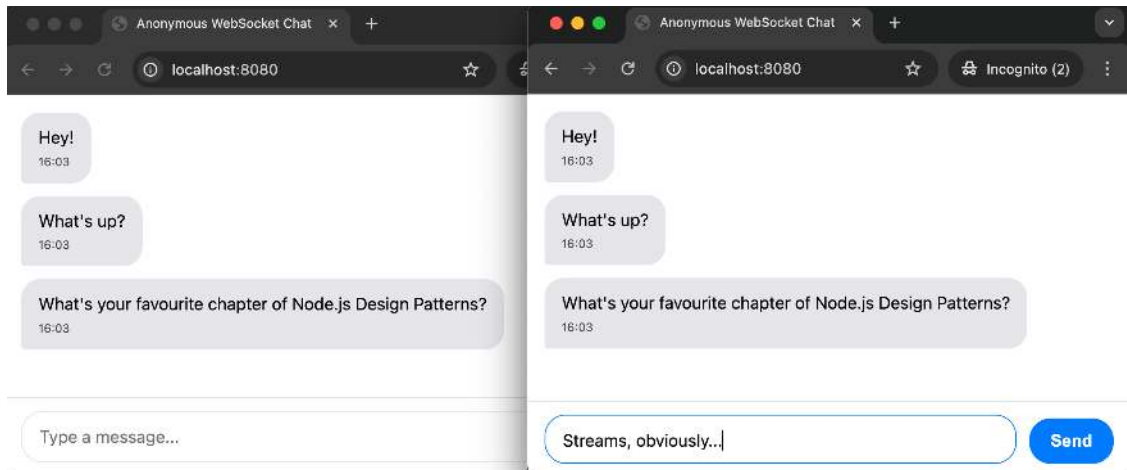


Figure 13.8 – Our new chat application in action

Very minimal, but it works!

Now, we want to see what happens when we try to scale our application by launching multiple instances. Let's try to do that. Let's start another server on another port:

```
node index.js 8081
```

The desired outcome should be that two different clients, connected to two different servers, can exchange chat messages. Unfortunately, this is not what happens with our current implementation. We can test this by opening another browser tab to `http://localhost:8081`.



In a real-world application, we could use a load balancer to distribute the load across our instances, but for this demo, we will not use one. This allows us to access each server instance in a deterministic way to verify how it interacts with the other instances.

When sending a chat message on one instance, we only broadcast the message locally, distributing it only to the clients connected to that server. In practice, the two servers are not talking to each other. We need to integrate them, and that's exactly what we are going to see next.

Using Redis as a simple message broker

We start our analysis of the most common Pub/Sub implementations by introducing **Redis** (`nodejsdp.link/redis`), which is a very fast and flexible in-memory data structure store. Redis is often used as a database or a cache server; however, among its many features, there is a pair of commands specifically designed to implement a centralized Pub/Sub message exchange pattern.

Redis's message brokering capabilities are (intentionally) very simple and basic, especially if we compare them to those of more advanced message-oriented middleware. However, this is one of the main reasons for its popularity. Often, Redis is already available in an existing infrastructure, for example, used as a cache server or as a session data store. Its speed and flexibility make it a very popular choice for sharing data in a distributed system. So, as soon as the need for a publish/subscribe broker arises in a project, the simplest and most immediate choice is to reuse Redis itself, avoiding the need to install and maintain a dedicated message broker.

Let's now work on an example to demonstrate the simplicity and power of using Redis as a message broker.



This example requires a working installation of Redis, listening on its default port. You can find more details at nodejsdp.link/redis-quickstart. The repository accompanying this book contains instructions on how to run a Docker-based Redis instance (nodejsdp.link/pubsub-redis).

Our plan of action is to integrate our chat servers using Redis as a message broker. Each instance publishes any message received from its clients to the broker, and at the same time, it subscribes to any message coming from other server instances. As we can see, each server in our architecture is both a subscriber and a publisher. The following diagram shows a representation of the architecture that we want to obtain:

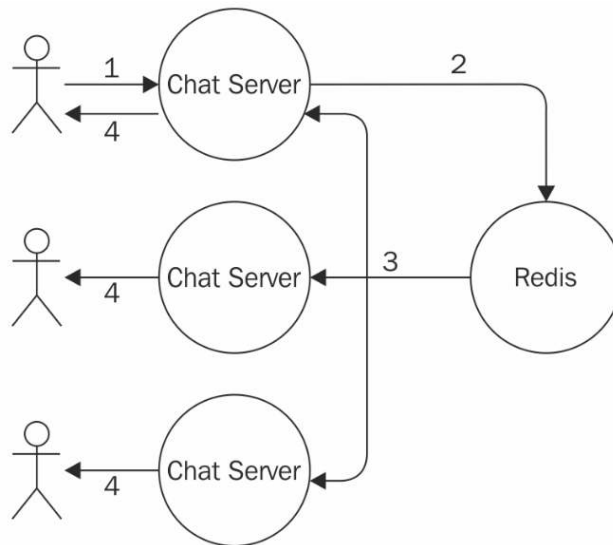


Figure 13.9 – Using Redis as a message broker for our chat application

Based on the architecture described in Figure 13.9, we can sum up the journey of a message as follows:

1. The message is typed into the textbox of the web page and sent to the connected instance of our chat server.
2. The message is then published to the broker.

3. The broker dispatches the message to all the subscribers, which in our architecture are all the instances of the chat server.
4. In each instance, the message is distributed to all the connected clients.

Let's see how this works in practice. Let's modify the server code by adding the publish/subscribe logic:

```
import { createServer } from 'node:http'
import { WebSocketServer } from 'ws' // v8.18.2
import staticHandler from 'serve-handler' // v6.1.6
import Redis from 'ioredis' // v5.6.1

const redisPub = new Redis() // 1
const redisSub = new Redis()

const server = createServer((req, res) => {
  return staticHandler(req, res, { public: 'web' })
})

const wss = new WebSocketServer({ server })
wss.on('connection', client => {
  console.log('Client connected')
  client.on('message', msg => {
    console.log(`Sending message to Redis: ${msg}`)
    redisPub.publish('chat_messages', msg) // 2
  })
})

redisSub.subscribe('chat_messages') // 3
redisSub.on('message', (channel, msg) => {
  if (channel === 'chat_messages') {
    console.log(`Received message from Redis: ${msg}`)
    for (const client of wss.clients) {
      if (client.readyState === WebSocket.OPEN) {
        client.send(Buffer.from(msg))
      }
    }
  }
})
```

The changes that we made to our original chat server are highlighted in the preceding code. This is how the new implementation works:

1. To connect our Node.js application to the Redis server, we use the `ioredis` package (`nodejsdp.link/ioredis`), which is a complete Node.js client supporting all the available Redis commands. Next, we instantiate two different connections, one used to subscribe to a channel and the other to publish messages. This is necessary in Redis because once a connection is put in subscriber mode, only commands related to the subscription can be used. This means that we need a second connection for publishing messages.
2. When a new message is received from a connected client, we publish the message in the `chat_messages` channel. We don't directly broadcast the message to our clients because our server is subscribed to the same channel (as we will see in a moment), so it will come back to us through Redis. For the scope of this example, this is a simple and effective mechanism. However, depending on the requirements of your application, you may instead want to broadcast the message immediately and ignore any message arriving from Redis and originating from the current server instance. We leave this to you as an exercise.
3. As we said, our server must also subscribe to the `chat_messages` channel, so we register a listener to receive all the messages published into that channel (either by the current server instance or any other chat server instance). When a message is received, we simply broadcast it to all the clients connected to the current WebSocket server.

These few changes are enough to integrate all the chat server instances that we might decide to start. To prove this, you can try starting multiple instances of our application:

```
node index.js 8080
node index.js 8081
node index.js 8082
```

You can then connect multiple browser tabs to each instance and verify that the messages you send to one instance are successfully received by all the other clients connected to the other instances.

Congratulations! We just integrated multiple nodes of a distributed real-time application using the Publish/Subscribe pattern.



Redis allows us to publish and subscribe to channels identified by a string, for example, `chat.nodejs`. But it also allows us to use glob-style patterns to define subscriptions that can potentially match multiple channels, for example, `chat.*`.

Peer-to-peer Publish/Subscribe with ZeroMQ

The presence of a broker can considerably simplify the architecture of a messaging system. However, in some circumstances, this may not be the best solution. This includes all the situations where low latency is critically important, or when scaling complex distributed systems, or when the presence of a single point of failure is not an option. The alternative to using a broker is, of course, implementing a peer-to-peer messaging system.

Introducing ZeroMQ

If our project is suited for a peer-to-peer architecture, ZeroMQ is a great option to consider. We briefly encountered it in *Chapter 9, Behavioral Design Patterns*, and here we will explore it in more detail. ZeroMQ is a fast, low-level networking library with a minimal API that provides building blocks for messaging systems, including atomic messages, load balancing, and queues. It supports multiple transports such as (inproc://), inter-process communication (ipc://), multicast using the PGM protocol (pgm:// or epgm://), and TCP (tcp://). It also includes tools for implementing the Publish/Subscribe pattern, which fits our example perfectly. In this section, we will remove the Redis broker from our chat application and let the nodes communicate directly in a peer-to-peer fashion using ZeroMQ's publish/subscribe sockets.



A ZeroMQ socket can be considered as a network socket on steroids, which provides additional abstractions to help implement the most common messaging patterns. For example, we can find sockets designed to implement publish/subscribe, request/reply, or one-way push communications.

Designing a peer-to-peer architecture for the chat server

When we remove the broker from our architecture, each instance of the chat server must directly connect to the other available instances to receive the messages they publish. In ZeroMQ, we have two types of sockets specifically designed for this purpose: PUB and SUB. The typical pattern is to bind a PUB socket to a local port where it will start listening for incoming subscription requests from sockets of type SUB.

Each subscription can specify a *topic* to indicate which types of messages should be delivered to the connected SUB sockets. When a message is sent through the PUB socket on a specific topic, it is broadcast to all SUB sockets that have subscribed to that same topic.

The following diagram shows the pattern applied to our distributed chat server architecture (with only two instances, for simplicity):

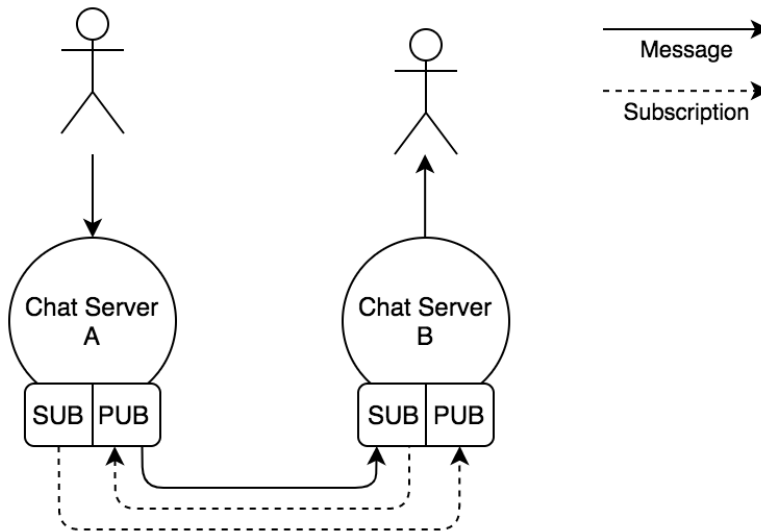


Figure 13.10 – Chat server messaging architecture using ZeroMQ PUB/SUB sockets

Figure 13.10 shows us the flow of information when we have two instances of the chat application, but the same concept can be applied to N instances. This architecture tells us that each node must be aware of the other nodes in the system to be able to establish all the necessary connections. It also shows us how the subscriptions go from a SUB socket to a PUB socket, while messages travel in the opposite direction.

Using the ZeroMQ PUB/SUB sockets

Let's see how the ZeroMQ PUB/SUB sockets work in practice by modifying our chat server. In this example, we are going to use the `zeromq` package :

```

import { createServer } from 'node:http'
import { parseArgs } from 'node:util'
import { WebSocketServer } from 'ws' // v8.18.2
import staticHandler from 'serve-handler' // v6.1.6
import zmq from 'zeromq' // v6.3.0

const { values: args } = parseArgs({ // 1
  options: {
    http: {
      type: 'string',
    },
  },
  pub: {
    type: 'string',
  },
})
  
```

```

    },
    sub: {
      type: 'string',
      multiple: true,
    },
  },
  args: process.argv.slice(2),
}))

if (!(args.http && args.pub && args.sub)) {
  console.error(
    'Usage: node index.js --http <port> --pub <port> '+
    '--sub <port1> [--sub <port2> ...]'
  )
  process.exit(1)
}

// serve static files
const server = createServer((req, res) => {
  return staticHandler(req, res, { public: 'web' })
})

// Initialize ZeroMq sockets
const pubSocket = new zmq.Publisher() // 2
await pubSocket.bind(`tcp://127.0.0.1:${args.pub}`)
const subSocket = new zmq.Subscriber() // 3
for (const port of args.sub) {
  console.log(`Subscribing to port ${port}`)
  await subSocket.connect(`tcp://127.0.0.1:${port}`)
}
subSocket.subscribe('chat_messages')

// Receive messages from other servers
async function receiveMessages() { // 4
  for await (const [_topic, msg] of subSocket) {
    console.log(`Received message from another server: ${msg.toString()}`)
    broadcast(Buffer.from(msg))
  }
}
receiveMessages()

```

```

const wss = new WebSocketServer({ server })
wss.on('connection', client => {
  console.log('Client connected')
  client.on('message', msg => {
    console.log(`Message: ${msg}`)
    broadcast(msg)
    pubSocket.send(['chat_messages', msg]) // 5
  })
})

function broadcast(msg) {
  for (const client of wss.clients) {
    if (client.readyState === WebSocket.OPEN) {
      client.send(msg)
    }
  }
}

server.listen(args.http)

```

The preceding code clearly shows that the logic of our application became slightly more complicated; however, it's still straightforward considering that we are implementing a peer-to-peer Publish/Subscribe pattern. Let's see how all the pieces come together:

1. We use the Node.js `parseArgs()` utility to process command-line options. We want to receive three options: `http` (port on which to bind our HTTP server for the frontend), `pub` (port on which to bind for publishing), and `sub` (an array of ports to connect to for subscription). If any of these options are missing, we print an error and exit.
2. Now, we create our Publisher socket and bind it to the port provided in the `--pub` command-line argument.
3. We create the Subscriber socket, and we connect it to the Publisher sockets of the other instances of our application. The ports of the target Publisher sockets are provided in the `--sub` command-line arguments (there might be more than one). We then create the actual subscription, by providing `chat_messages` as a topic.
4. The `receiveMessages()` function, which is immediately invoked after being declared, is used to receive messages arriving at our Subscriber socket. It uses a `for await...of` loop, since `subSocket` is an async iterable. With each message we receive, we `broadcast()` the actual payload to all the clients connected to the current WebSocket server.
5. When a new message is received by the WebSocket server of the current instance, we broadcast it to all the connected clients, but we also publish it through our Publisher socket. Once again, we use `chat_messages` as a topic.

We have now built a simple distributed system, integrated using a peer-to-peer Publish/Subscribe pattern!

Let's fire it up! Let's start three instances (in three different terminals) of our application by making sure to connect their Publisher and Subscriber sockets properly:

```
node index.js --http 8080 --pub 5000 --sub 5001 --sub 5002
node index.js --http 8081 --pub 5001 --sub 5000 --sub 5002
node index.js --http 8082 --pub 5002 --sub 5000 --sub 5001
```

The first command will start an instance with an HTTP server listening on port 8080, while binding its Publisher socket on port 5000 and connecting the Subscriber socket to ports 5001 and 5002, which are where the Publisher sockets of the other two instances should be listening. The other two commands work in a similar way.

Now, the first thing you will see is that ZeroMQ will not complain if a Subscriber socket can't establish a connection to a Publisher socket. For example, at the time of the first command, there are no Publisher sockets listening on ports 5001 and 5002; however, ZeroMQ is not throwing any error. This is because ZeroMQ is built to be resilient to faults, and it implements a built-in connection retry mechanism. This feature also comes in particularly handy if any node goes down or is restarted. The same *forgiving* logic applies to the Publisher socket: if there are no subscriptions, it will simply drop all the messages, but it will continue working.

At this point, we can try to navigate with a browser to any of the server instances that we started and verify that the messages are properly propagated to all the chat servers.



In the previous example, we assumed a static architecture where the number of instances and their addresses are known in advance. We can introduce a service registry, as explained in *Chapter 12, Scalability and Architectural Patterns*, to connect our instances dynamically. It is also important to point out that ZeroMQ can be used to implement a broker using the same primitives we demonstrated here.

Reliable message delivery with queues

An important abstraction in a messaging system is the MQ. With an MQ, the sender and the receiver(s) of the message don't necessarily need to be active and connected at the same time to establish a communication, because the queuing system takes care of storing the messages until the destination is able to receive them. This behavior differs from the *fire-and-forget* paradigm, where a subscriber can receive messages only during the time it is connected to the messaging system.

A subscriber that can always reliably receive all the messages, even those sent when it's not listening for them, is called a **durable subscriber**.

We can summarize the **delivery semantics** of a messaging system in three categories:

- **At most once:** Also known as *fire-and-forget*, the message does not persist, and the delivery is not acknowledged. This means that the message can be lost in cases of crashes or disconnections of the receiver. This is the model we have been using in our chat implementations so far in this chapter.
- **At least once:** The message is guaranteed to be received at least once, but duplicates might occur if, for example, the receiver crashes before notifying the sender of the reception. This implies that the message must be persisted in the eventuality that it has to be sent again.
- **Exactly once:** This is the most reliable delivery semantic. It guarantees that the message is received once and only once. This comes at the expense of a slower and more data-intensive mechanism for acknowledging the delivery of messages.

We have a durable subscriber when our messaging system can achieve an “at least once” or “exactly once” delivery semantic, and to do that, the system must use an MQ to accumulate the messages while the subscriber is disconnected. The queue can be stored in memory or persisted on disk to allow the recovery of its messages even if the queuing system restarts or crashes.

The following diagram shows a graphical representation of a durable subscriber backed by a message queue:

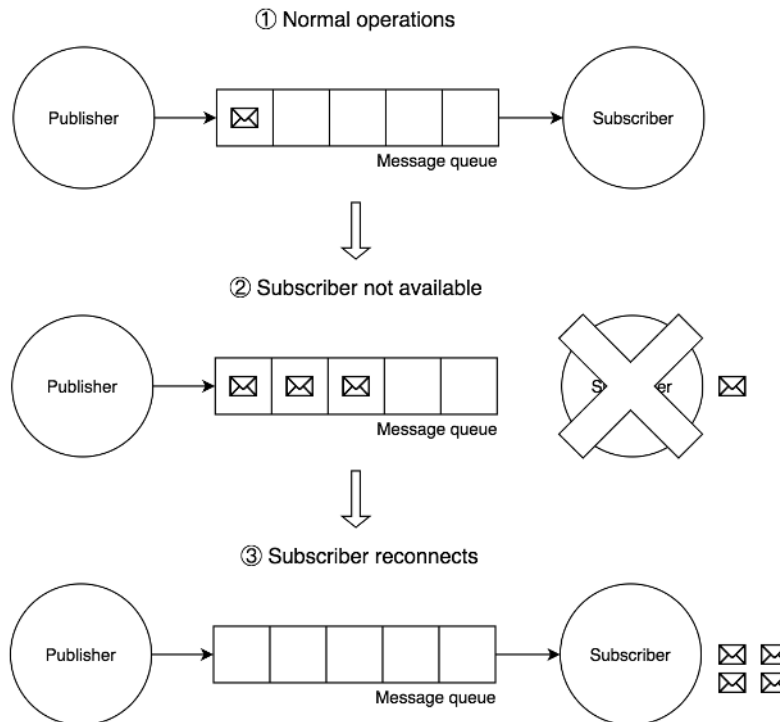


Figure 13.11 – Example behavior of a messaging system backed by a queue

Figure 13.11 shows how a message queue can help us implement the Durable Subscriber pattern. As we can see, during normal operations (1), messages travel from the publisher to the subscriber through the message queue. When the subscriber goes offline (2) because of a crash, a malfunction, or simply a planned maintenance period, any message sent by the publisher is stored and accumulated safely in the message queue. Afterward, when the subscriber comes back online (3), all messages accumulated in the queue are sent to the subscriber, so no message is lost.

The durable subscriber is probably the most important pattern enabled by a message queue, but it's certainly not the only one, as we will see later in the chapter.

Next, we are going to learn about AMQP, which is the protocol we are going to use throughout the rest of the chapter to implement our message queue examples.

Introducing AMQP

A message queue is often used in systems where messages must never be lost, such as banking, air traffic control, or healthcare applications. Enterprise-grade queues use robust protocols and persistent storage to ensure delivery even during failures. This reliability has traditionally made them complex and often proprietary, which in turn can lead to vendor lock-in.

In recent years, open protocols like AMQP, STOMP, and MQTT have brought advanced messaging into the mainstream. For our queuing system, we will use **AMQP**. It is an open standard supported by many message-queuing systems and defines both a communication protocol and a model for routing, filtering, queuing, reliability, and security.

The following diagram shows us all the AMQP components:

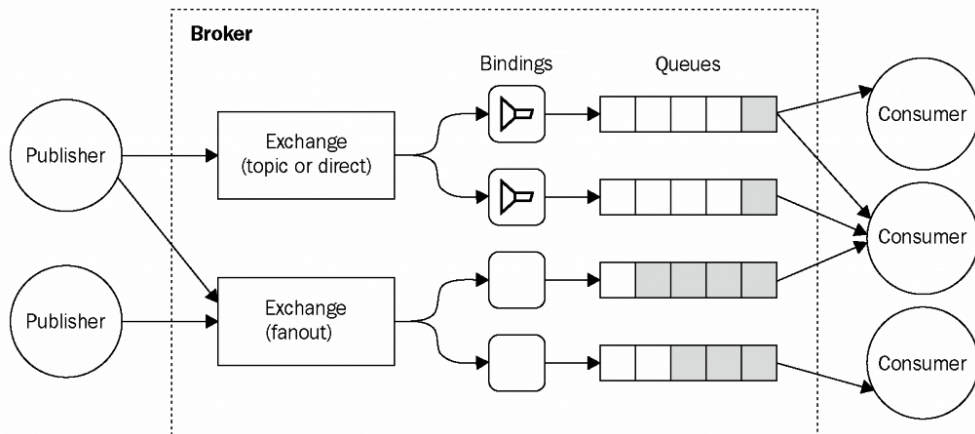


Figure 13.12 – Example of an AMQP-based messaging system

As shown in *Figure 13.12*, in AMQP, there are three essential components:

- **Queue:** The data structure responsible for storing the messages consumed by the clients. The messages from a queue are pushed (or pulled) to one or more consumers. If multiple consumers are attached to the same queue, the messages are load-balanced across them. A queue can be any of the following:
 - **Durable:** This means that the queue is automatically recreated if the broker restarts. A durable queue does not mean that its contents are preserved as well; in fact, only messages that are marked as persistent are saved to the disk and restored in case of a restart.
 - **Exclusive:** This means that the queue is bound to only one subscriber connection. When the connection is closed, the queue is destroyed.
 - **Auto-delete:** This will cause the queue to be deleted when the last subscriber disconnects.
- **Exchange:** This is where a message is published. An exchange routes the messages to one or more queues depending on the algorithm it implements:
 - **Direct exchange:** It routes the messages by matching an entire routing key (for example, `chat.msg`)
 - **Topic exchange:** It distributes the messages using a glob-like pattern matched against the routing key (for example, `chat.#` matches all the routing keys starting with `chat.`)
 - **Fan-out exchange:** It broadcasts a message to all the connected queues, ignoring any routing key provided
- **Binding:** This is the link between exchanges and queues. It also defines the routing key, or the pattern used to filter the messages that arrive from the exchange.

These components are managed by a broker, which exposes an API for creating and manipulating them. When connecting to a broker, a client creates a **channel** (an abstraction of a connection) that is responsible for maintaining the state of the communication with the broker.



In AMQP, we can obtain the Durable Subscriber pattern by creating any type of queue that is not exclusive or auto-delete.

The AMQP model is way more complex than the messaging systems we have used so far (Redis and ZeroMQ). However, it offers a set of features and a level of reliability that would be very hard to obtain using only primitive publish/subscribe mechanisms.



You can find a detailed introduction to the AMQP model on the RabbitMQ website at nodejshp.link/amqp-components.

Durable subscribers with AMQP and RabbitMQ

Let's now practice what we learned about durable subscribers and AMQP and work on a small example. A typical scenario where it's important not to lose any messages is when we want to keep the different services of a microservice architecture in sync (we already described this integration pattern in the previous chapter). If we want to use a broker to keep all our services on the same page, it's important that we don't lose any information; otherwise, we might end up in an inconsistent state.

Designing a history service for the chat application

Let's now extend our small chat application using a microservice approach. Let's add a history service that persists our chat messages inside a database, so that when a client connects, we can query the service and retrieve the entire chat history. We are going to integrate the history service with the chat server using the RabbitMQ broker (nodejsdp.link/rabbitmq) and AMQP.

The following diagram shows our planned architecture:

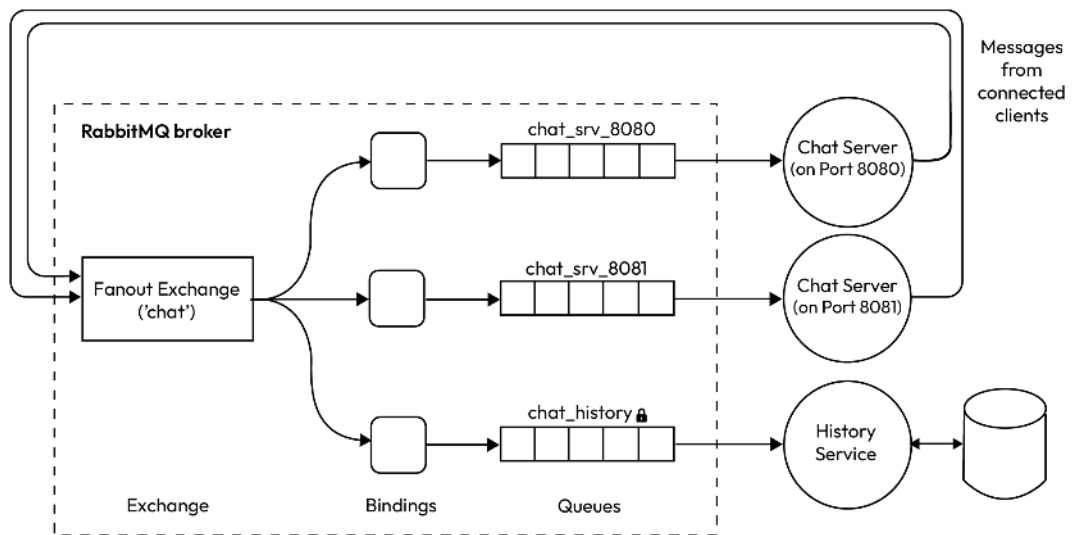


Figure 13.13 – Architecture of our chat application with AMQP and history service

As shown in Figure 13.13, we are going to use a single fan-out exchange; we don't need any complicated routing logic, so our scenario does not require any exchange more complex than that. Next, we will create one queue for each instance of the chat server.

These queues are exclusive since we are not interested in receiving any messages missed while a chat server is offline; that's the job of our history service, which can eventually also implement more complicated queries against the stored messages. In practice, this means that our chat servers are not durable subscribers, and their queues will be destroyed as soon as the connection is closed. The history service instead, cannot afford to lose any messages; otherwise, it would not fulfill its very purpose. Therefore, the queue we are going to create for it must be durable, so that any message that is published while the history service is disconnected will be kept in the queue and delivered when it comes back online.

We are going to use the familiar LevelDB as the storage engine for the history service, while we will use the `amqplib` package (nodejsdp.link/amqplib) to connect to RabbitMQ using the AMQP protocol.



The example that follows requires a working RabbitMQ server, listening on its default port. For more information, please refer to its official installation guide at nodejsdp.link/rabbitmq-getstarted. The repository accompanying this book contains instructions on how to run a Docker-based RabbitMQ server instance (nodejsdp.link/pubsub-amqp).

Implementing a history service using AMQP

Let's now implement our history service! We are going to create a standalone application (in a typical microservice fashion), which is implemented in the `historySvc.js` module. The module is made up of two parts: an HTTP server to expose the chat history to clients and an AMQP consumer responsible for capturing the chat messages and storing them in a local database. This service will be the source of truth for keeping historic messages for all the connected chat clients.

Let's see what this looks like in the code that follows:

```
import { createServer } from 'node:http'
import { Level } from 'level' // v10.0.0
import { monotonicFactory } from 'ulid' // v3.0.1
import amqp from 'amqplib' // v0.10.8

const ulid = monotonicFactory() // 1
const db = new Level('msgHistory', { valueEncoding: 'json' })

const connection = await amqp.connect('amqp://localhost') // 2
const channel = await connection.createChannel()
await channel.assertExchange('chat', 'fanout') // 3
const { queue } = channel.assertQueue('chat_history') // 4
await channel.bindQueue(queue, 'chat') // 5

channel.consume(queue, async msg => { // 6
  const data = JSON.parse(msg.content.toString())
  console.log(`Saving message: ${msg.content.toString()}`)
```

```

    await db.put(ulid(), data)
    channel.ack(msg)
  })

createServer(async (req, res) => { // 7
  const url = new URL(req.url, 'http://localhost')
  const lt = url.searchParams.get('lt')
  res.writeHead(200, { 'Content-Type': 'application/json' })
  const messages = []
  for await (const [key, value] of db.iterator({
    reverse: true,
    limit: 10,
    lt,
  })) {
    messages.unshift({ id: key, ...value })
  }
  res.end(JSON.stringify(messages, null, 2))
}).listen(8090)

```

As we mentioned, this code sets up a service that listens for messages from a RabbitMQ fan-out exchange, persists them into a LevelDB database, and exposes an HTTP endpoint to retrieve the stored messages. It demonstrates how to decouple message consumption from data storage in a distributed system. Let's walk through the most important parts:

1. We create a **Universally Unique Lexicographically Sortable Identifier (ULID)** generator (nodejsdp.link/ulid). ULIDs are like UUIDs but lexicographically sortable by time, making them perfect for storing our records in order by time in the database. This makes it efficient to read the records back when needed. We also initialize a Level client, setting the encoding to JSON (which allows us to store structured data in every record).
2. At this point, we start to set up the AMQP components, which requires executing a few different commands. We first establish a connection with the AMQP broker, which in our case is RabbitMQ. Then, we create a channel, which is like a session that will maintain the state of our communications.
3. Next, we set up an exchange, named `chat`. As we already mentioned, it is a fan-out exchange. The `assertExchange()` command will make sure that the exchange exists on the broker; otherwise, it will create it.
4. We also create a queue called `chat_history`. By default, the queue is durable (not exclusive and not auto-delete), so we don't need to pass any extra options to support durable subscribers.
5. Next, we bind the queue to the exchange we previously created. Here, we don't need to specify any other option (such as a routing key or pattern), as the exchange is of the fan-out type, so it doesn't perform any filtering.

6. We can now begin to listen for messages coming from the queue we just created. Note that this time we assume that every message contains an object with the `text` and `timestamp` keys serialized in JSON. This design allows us to retain the original timestamp of when a message was sent together with its text. We save every message that we receive in a Level database using a new monotonic ULID as the key to keep the messages sorted by insertion date. It's also interesting to note that we are acknowledging every message using `channel.ack(msg)`, but only after the message is successfully saved into the database. If the ACK (acknowledgment) is not received by the broker, the message is kept in the queue to be processed again. We are using an at-least-once delivery semantic here, which means that if our service crashes or is interrupted just after saving the message to the database (and before sending the ACK), we might end up with duplicated messages. This is a reasonable design choice for our app.
7. Finally, we create a simple HTTP server that serves stored messages. When a client makes a request, we fetch the latest 10 messages from LevelDB. Note that we can optionally retrieve the last 10 messages before a given message ID (using the optional query parameter `lt`). This is something that can be used to implement a performant pagination system. We are not going to leverage this in our implementation for the other components (frontend and backend), but it gives you a good starting point if you decide to take this simple chat application to the next level (pun intended).



If we are not interested in sending explicit acknowledgments, we can pass the `{ noAck: true }` option to the `channel.consume()` API. This tells the broker that messages are considered successfully delivered as soon as they are dispatched, without waiting for confirmation from the consumer. While this can improve performance and reduce complexity, it comes with a trade-off: we lose delivery guarantees. If the consumer crashes before it finishes processing the message (or before persisting it somewhere durable, such as Level), the message is gone for good. This setup gives us at-most-once delivery semantics, meaning that messages might be lost in case of failure, but never duplicated.

Integrating the chat application with AMQP

To integrate the chat servers using AMQP, we can use a setup very similar to the one we implemented in the history service, but with some small variations. So, let's see how the new `index.js` module looks with the introduction of AMQP:

```
import { createServer } from 'node:http'
import staticHandler from 'serve-handler' // v6.1.6
import { WebSocketServer } from 'ws' // v8.18.2
import amqp from 'amqplib' // v0.10.8

const httpPort = process.argv[2] || 8080

// register the server with RabbitMQ and create a queue
const connection = await amqp.connect('amqp://localhost')
const channel = await connection.createChannel()
```

```

await channel.assertExchange('chat', 'fanout')
const { queue } = await channel.assertQueue( // 1
  `chat_srv_${httpPort}`, {
    exclusive: true,
  }
)
await channel.bindQueue(queue, 'chat')
channel.consume( // 2
  queue,
  msg => {
    msg = msg.content.toString()
    console.log(`From queue: ${msg}`)
    broadcast(Buffer.from(msg))
  },
  { noAck: true }
)

// serve static files
const server = createServer((req, res) => {
  return staticHandler(req, res, { public: 'web' })
})

const wss = new WebSocketServer({ server })
wss.on('connection', async client => {
  console.log('Client connected')
  client.on('message', msg => {
    console.log(`Sending message: ${msg}`)
    channel.publish( // 3
      'chat',
      '',
      Buffer.from(
        JSON.stringify({
          text: msg.toString(),
          timestamp: Date.now(),
        })
      )
    )
  })
})

// Load previous messages from the history service
const res = await fetch('http://localhost:8090') // 4

```



```

const messages = await res.json()
for (const message of messages) {
  client.send(Buffer.from(JSON.stringify(message)))
}
})

function broadcast(msg) {
  for (const client of wss.clients) {
    if (client.readyState === WebSocket.OPEN) {
      client.send(msg)
    }
  }
}

server.listen(httpPort)

```

As we can see, AMQP made the code a little bit more verbose on this occasion too, but at this point, we should already be familiar with most of it. There are just a few aspects to be aware of:

1. As we mentioned, our chat server doesn't need to be a durable subscriber: a fire-and-forget paradigm is enough. So, when we create our queue, we pass the `{ exclusive: true }` option, indicating that the queue is scoped to the current connection and therefore it will be destroyed as soon as the chat server shuts down.
2. For the same reason as in the previous point, we don't need to send back any acknowledgement when we read a message from the queue. So, to make things easier, we pass the `{ noAck: true }` option when starting to consume the messages from the queue.
3. Publishing a new message is also very easy. We simply have to specify the target exchange (chat) and a routing key, which in our case is empty (`' '`) because we are using a fan-out exchange, so there is no routing to perform.
4. The other peculiarity of this version of our chat server is that we can now present to the user the full history of the chat, thanks to our history microservice. We do that by querying the history microservice and sending every past message to the client as soon as a new connection is established.

Before we can run our services, there are a couple of small changes we need to make on the frontend. Now, messages arriving to the frontend through WebSocket contain a JSON-serialized object, so we need to deserialize the object before appending the new message to the DOM. This can be done by locating the following line in `web/index.html`:

```
messageContent.textContent = content
```

We replace it with the following:

```
const contentData = JSON.parse(content)
messageContent.textContent = contentData.text
```

Also, we can take this opportunity to display the full timestamp since, now that messages are persisted, it is likely that conversations will span across multiple days. This is easily done by locating:

```
messageTime.textContent = new Date().toLocaleTimeString([], {/* ... */)
```

We replace it with the following:

```
messageTime.textContent = new Date(contentData.timestamp).toLocaleString()
```

We can now run our new and improved chat application. To do that, first make sure to have RabbitMQ running locally on your machine. Then, let's start two chat servers and the history service in three different terminals:

```
node index.js 8080
node index.js 8081
node historySvc.js
```

We should now focus our attention on how our system, and particularly the history service, behaves in case of downtime. If we stop the history server and continue to send messages using the web UI of the chat application, we will see that when the history server is restarted, it will immediately receive all the messages it missed. This is a perfect demonstration of how the Durable Subscriber pattern works!



It is interesting to see how the microservice approach allows our system to survive even without one of its components: the history service. There would be a temporary reduction of functionality (no chat history available), but people would still be able to exchange chat messages in real time. Awesome!

Reliable messaging with streams

At the beginning of this chapter, we mentioned that a possible alternative to MQs is **streams**. The two paradigms are similar in scope but fundamentally different in their approach to messaging. In this section, we are going to unveil the power of streams by leveraging Redis streams to implement our chat application.

Characteristics of a streaming platform

In system integration, a **stream** (or **log**) is an ordered, append-only, durable data structure. **Records** are always appended and, unlike queues, are not deleted when consumed. This makes a stream behave more like a data store, allowing queries for past records or replay from a specific point. Streams are pull-based, so consumers process records at their own pace without being overwhelmed. Because data is retained (unless explicitly removed or subject to retention limits), streams provide reliable delivery out of the box. If a consumer crashes, it can resume from where it left off, reprocessing any missed messages and restoring its state.

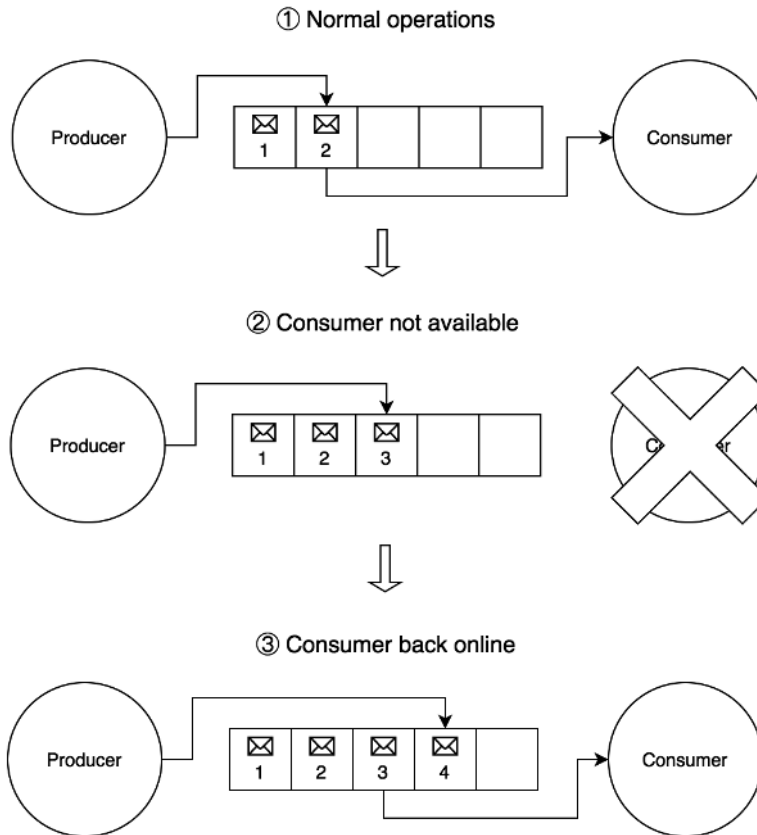


Figure 13.14 – Reliable message delivery with streams

As Figure 13.14 shows, during normal operations (1), the consumer processes the records in the stream as soon as they are added by the producer. When the consumer becomes unavailable (2) because of a problem or a scheduled maintenance, the producer simply continues to add records to the stream as normal. When the consumer comes back online (3), it starts processing the records from the point where it left off. The main aspect of this mechanism is that it's very simple and barebones, but it's quite effective at making sure that no message is lost even when the consumer is not available.

Streams versus message queues

Message queues and streams share similarities but serve different needs. Streams are ideal for processing sequential data, especially when consumers need to batch messages or correlate them with past data. Modern streaming platforms can handle gigabytes per second and distribute both data and processing across many nodes. Both can implement Publish/Subscribe patterns with reliable delivery.

However, message queues are often better for complex system integration, thanks to features like advanced routing and message prioritization, while streams always preserve record order. Similarly, both can be used for task distribution, but queues may be preferable in standard architectures for the same reasons. In short, use streams for high-volume sequential data and *replayability*, and use message queues for task distribution, advanced routing, prioritization, and complex integration scenarios.

Implementing the chat application using Redis streams

At the time of writing, the most popular streaming platforms out there are Apache Kafka (nodejsdp.link/kafka) and Amazon Kinesis (nodejsdp.link/kinesis). However, for simpler tasks, we can rely again on Redis, which implements a log data structure called **Redis streams**.

In the next code sample, we are going to see Redis streams in action by adapting our chat application. The immediate advantage of using a stream over an MQ is that we don't need to rely on a dedicated component to store and retrieve the history of the messages exchanged in a chat room, but we can simply query the stream every time we need to access older messages. As we will see, this simplifies a lot the architecture of our application and certainly makes streams a better choice than MQs, at least for our very simple use case.

So, let's dive into some code. Let's update the `index.js` of our chat application to use Redis streams:

```
import { createServer } from 'node:http'
import staticHandler from 'serve-handler' // v6.1.6
import { WebSocketServer } from 'ws' // v8.18.2
import Redis from 'ioredis' // v5.6.1

const redisClient = new Redis()
const redisClientXread = new Redis()

// serve static files
const server = createServer((req, res) => {
  return staticHandler(req, res, { public: 'web' })
})

const wss = new WebSocketServer({ server })
wss.on('connection', async client => {
  console.log('Client connected')
  client.on('message', msg => {
    console.log(`Sending message: ${msg}`)
  })
})
```

```

    redisClient.xadd( // 1
      'chat_stream',
      '*',
      'message',
      JSON.stringify({
        text: msg.toString(),
        timestamp: Date.now(),
      })
    )
  })
})

// Load previous messages from the history service
const logs = await redisClient.xrange('chat_stream', '-', '+') // 2
for (const [, [, message]] of logs) {
  client.send(Buffer.from(message))
}
})

function broadcast(msg) {
  for (const client of wss.clients) {
    if (client.readyState === WebSocket.OPEN) {
      client.send(msg)
    }
  }
}

let lastRecordId = '$'

async function processStreamMessages() { // 3
  while (true) {
    const [[, records]] = await redisClientXread.xread(
      'BLOCK',
      '0',
      'STREAMS',
      'chat_stream',
      lastRecordId
    )
    for (const [recordId, [, message]] of records) {
      console.log(`Message from stream: ${message}`)
      broadcast(Buffer.from(message))
      lastRecordId = recordId
    }
  }
}

```

```
    }  
  }  
}  
  
processStreamMessages()  
  
server.listen(process.argv[2] || 8080)
```

As always, the overall structure of the application has remained the same; what changed is the API we used to exchange messages with the other instances of the application.

Let's look at those APIs more closely:

1. The first command we want to analyze is `xadd`. This command appends a new record to a stream, and we are using it to add a new chat message as it arrives from a connected client. We pass to `xadd` the following arguments:
 - The name of the stream, which in our case is `chat_stream`.
 - The ID of the record. In our case, we provide an asterisk (*), which is a special ID that asks Redis to generate an ID for us. This is usually what we want, as IDs must be monotonic to preserve the lexicographic order of the records, and Redis takes care of that for us.
 - It follows a list of key-value pairs. In our case, we specify only a 'message' key of the value `msg` (which is the message we receive from the client).
2. This is one of the most interesting aspects of using streams: we query the past records of the stream to retrieve the chat history. We do this every time a client connects. We use the `xrange` command for that, which, as the name implies, allows us to retrieve all the records in the stream within the two specified IDs. In our case, we are using the special IDs '-' (minus) and '+' (plus), which indicate the lowest possible ID and the highest possible ID. This essentially means that we want to retrieve all the records currently in the stream.
3. The last interesting part of our new chat application is where we wait for new records to be added to the stream. This allows each application instance to read new chat messages as they are added to the queue, and it's an essential part for the integration to work. We use an infinite loop and the `xread` command for the task, providing the following arguments:
 - `BLOCK` means that we want the call to block until new messages arrive.
 - Next, we specify the timeout after which the command will simply return with a null result. In our case, `0` means that we want to wait forever.
 - `STREAMS` is a keyword that tells Redis that we are now going to specify the details of the streams we want to read.
 - `chat_stream` is the name of the stream we want to read.

- Finally, we supply the record ID (`lastRecordId`) after which we want to start reading the new messages. Initially, this is set to `$` (dollar sign), which is a special ID indicating the highest ID currently in the stream, which should essentially start to read the stream after the last record currently in the stream. After we read the first record, we update the `lastRecordId` variable with the ID of the last record read.

Within the previous example, we also made use of some clever *destructuring* instructions. Consider, for example, the following code:

```
for (const [, [, message]] of logs) {...}
```

This instruction could be expanded to something like the following:

```
for (const [recordId, [propertyId, message]] of logs) {...}
```

But since we are not interested in getting the `recordId` and `propertyId` values, we are simply keeping them out of the destructuring instruction. This destructuring technique, in combination with the `for...of` loop, is convenient to parse the data returned from the `xrange` command (in just one line of code), which in our case is in the following form:

```
[  
  ["1588590110918-0", ["message", "This is a message"]],  
  ["1588590130852-0", ["message", "This is another message"]]  
]
```

We applied a similar principle to parse the return value of `xread`. Please refer to the API documentation of those instructions for a detailed explanation of their return value.



You can read more about the `xadd` command and the format of record IDs in the official Redis documentation at nodejsdp.link/xadd.

The `xread` command has a slightly more advanced set of arguments and return values, which you can explore in more detail at nodejsdp.link/xread.

Finally, you can check out the documentation for `xrange` at nodejsdp.link/xrange.

Now, you can start a couple of server instances again and test the application to see how the new implementation works.

It's interesting to highlight again the fact that we didn't need to rely on a dedicated component to manage our chat history, but instead, all we needed to do was to retrieve the past records from the stream with `xrange`. This aspect of streams makes them intrinsically reliable as no message is *lost* unless explicitly deleted.



Records can be removed from the stream with the `xdel` (nodejsdp.link/xdel) or `xtrim` command (nodejsdp.link/xtrim) or with the `MAXLEN` option of `xadd` (nodejsdp.link/xadd-maxlen).

This concludes our exploration of the Publish/Subscribe pattern. Now, it's time to discover another important category of messaging patterns: task distribution patterns.

Task distribution patterns

In *Chapter 11, Advanced Recipes*, you learned how to delegate costly tasks to multiple local processes. Even though this was an effective approach, it's inherently limited to the capacity of a single machine. Sooner or later, you'll hit a hard scalability ceiling, whether due to CPU saturation, memory constraints, or I/O bottlenecks. In this section, we'll explore how to apply a similar pattern in a distributed architecture by using remote workers that can run anywhere in the network. This opens the door to a much more scalable solution, where work can be distributed across dozens, hundreds, or even thousands of machines running independently and in parallel.



As part of his cloud consulting work at fourTheorem (nodejsdp.link/4t), one of the authors of this book (Luciano) contributed to a large-scale modernization project in the financial sector. By applying many of the techniques covered in this chapter (including asynchronous communication, distributed messaging, event-driven orchestration, and stream-based coordination), the team was able to dramatically improve system performance and scalability. Key outcomes included reducing “portfolio analytics” computation times from 10 hours to about 1 hour, eliminating resource contention for real-time analytics, and cutting the overall codebase by around 70%, leading to better maintainability and faster innovation. This project demonstrates how effective messaging patterns, combined with the right infrastructure choices, can unlock high performance and resilience even in the most demanding environments. If you are curious to learn more, you can read the case study at nodejsdp.link/fin-mod.

The idea is to have a messaging pattern that allows us to spread tasks across multiple machines. These tasks might be individual chunks of work or pieces of a bigger task split using a *divide-and-conquer* approach.

If we look at the logical architecture represented in the following diagram, we should be able to recognize a familiar pattern:

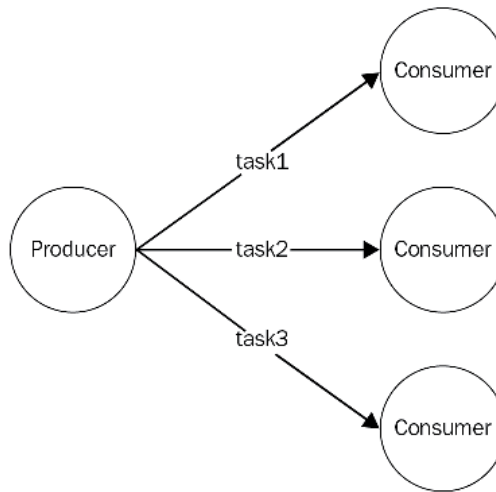


Figure 13.15 – Distributing tasks to a set of consumers

As we can see from the diagram of *Figure 13.15*, the Publish/Subscribe pattern is not suitable for this type of application, as we absolutely don't want a task to be received by multiple workers. What we need instead is a message distribution pattern, such as a load balancer that dispatches each message to a different consumer (also called a **worker**, in this case). In messaging systems terminology, this pattern is also known as **competing consumers**, fan-out distribution, or **ventilator**.

One important difference compared to the HTTP load balancers we saw in the previous chapter is that here, the consumers have a more active role. In fact, as we will see later, most of the time it's not the producer that initiates communication with the consumers. Instead, it's the consumers that actively *poll* the task producer or task queue to check for and retrieve new jobs. This has a significant advantage in scalable systems, as it allows us to increase the number of workers seamlessly, without having to modify the producer or introduce a service registry.

Also, in a generic messaging system, we don't necessarily have request/reply communication between the producer and the workers. Instead, most of the time, the preferred approach is to use one-way asynchronous communication, which enables better parallelism and scalability.

In such an architecture, messages can potentially always travel in one direction, creating **pipelines**, as shown in the following diagram:

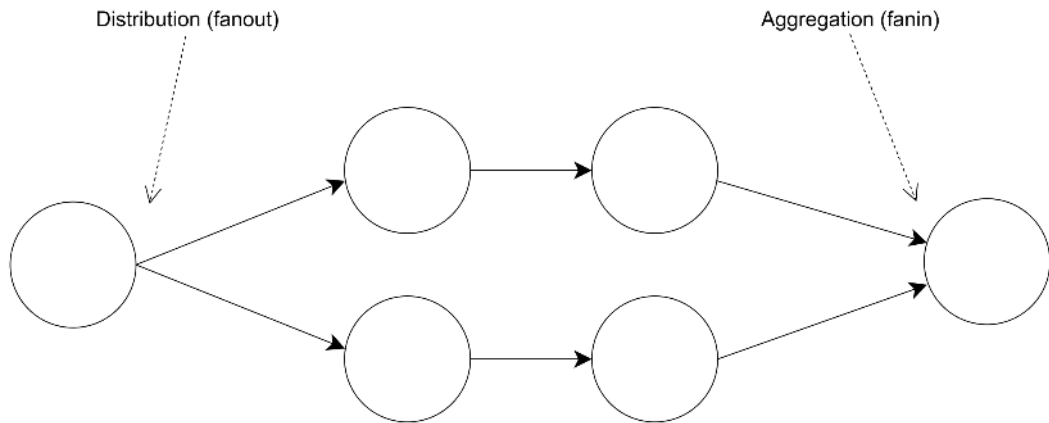


Figure 13.16 – A messaging pipeline

Pipelines allow us to build very complex processing architectures without the overhead of a synchronous request/reply communication, often resulting in lower latency and higher throughput. In *Figure 13.16*, we can see how messages can be distributed across a set of workers (*fan out*), forwarded to other processing units, and then aggregated into a single node (*fan in*), usually called the **sink**.

In this section, we are going to focus on the building blocks of these kinds of architectures, by analyzing the two most important variations: peer-to-peer and broker-based.



The combination of a pipeline with a task distribution pattern is also called a **parallel pipeline**.

The ZeroMQ fan-out/fan-in pattern

We have already discovered some of the capabilities of ZeroMQ for building peer-to-peer distributed architectures. In the previous section, in fact, we used PUB and SUB sockets to disseminate a single message to multiple consumers, and now, we are going to see how it's possible to build parallel pipelines using another pair of sockets called PUSH and PULL.

PUSH/PULL sockets

Intuitively, we can say that the PUSH sockets are made for *sending* messages, while the PULL sockets are meant for *receiving*. It might seem a trivial combination; however, they have some extra features that make them perfect for building one-way communication systems:

- Both can work in either *connect* mode or *bind* mode. In other words, we can create a PUSH socket and bind it to a local port listening for incoming connections from a PULL socket, or vice versa, a PULL socket might listen for connections from a PUSH socket. The messages always travel in the same direction, from PUSH to PULL; it's only the initiator of the connection that can be different. Bind mode is the best solution for *durable* nodes, such as, for example, the task producer and the sink, while connect mode is perfect for *transient* nodes, such as the task workers. This allows the number of transient nodes to vary arbitrarily without affecting the more stable, durable nodes.
- If there are multiple PULL sockets connected to a single PUSH socket, the messages are evenly distributed across all the PULL sockets. In practice, they are load balanced (peer-to-peer load balancing!). On the other hand, a PULL socket that receives messages from multiple PUSH sockets will process the messages using a fair queuing system, which means that they are consumed evenly from all the sources, with a round-robin applied to inbound messages.
- The messages sent over a PUSH socket that doesn't have any connected PULL sockets do not get lost. They are instead queued until a node comes online and starts pulling the messages.

We are now starting to understand how ZeroMQ is different from traditional web services and why it's the perfect tool for building a distributed messaging system.

Building a distributed hashsum cracker with ZeroMQ

Now it's time to build a sample application to see the properties of the PUSH/PULL sockets we just described in action.

A simple and fascinating application to work with would be a *hashsum cracker*: a system that uses a brute-force approach to try to match a given hashsum (such as MD5 or SHA1) to the hashsum of every possible variation of characters of a given alphabet, thus discovering the original string the given hashsum was created from.

This is an *embarrassingly parallel* workload (nodejsdp.link/embarrassingly-parallel), which is perfect for building an example demonstrating the power of parallel pipelines.



Never use plain hashsums to store passwords, as they are easy to crack. Use instead a purpose-built algorithm such as **bcrypt** (nodejsdp.link/bcrypt), **scrypt** (nodejsdp.link/scrypt), **PBKDF2** (nodejsdp.link/pbkdf2), or **Argon2** (nodejsdp.link/argon2).

For our application, we want to implement a typical parallel pipeline where we have the following:

- A node to create and distribute tasks across multiple workers
- Multiple worker nodes (where the actual computation happens)
- A node to collect all the results

The system we just described can be implemented in ZeroMQ using the following architecture:

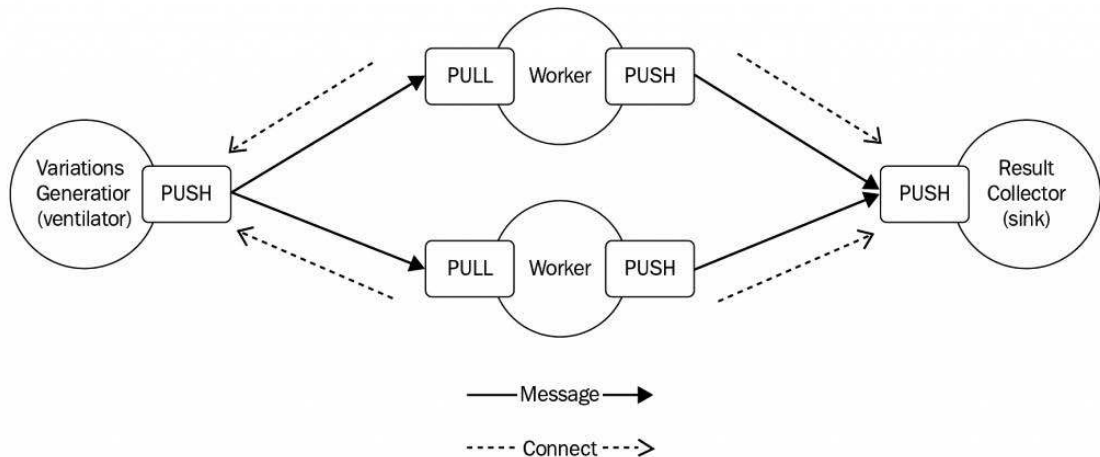


Figure 13.17 – The architecture of a typical pipeline with ZeroMQ

In our architecture, we have a *ventilator* generating intervals of variations of characters in the given alphabet (for example, the interval 'aa' to 'bb' includes the variations 'aa', 'ab', 'ba', 'bb') and distributing those intervals to the workers as tasks. Each worker, then, calculates the hashsum of every variation in the given interval, trying to match each resulting hashsum against the control hashsum given as input. If a match is found, the result is sent to a results collector node (sink).

The durable nodes of our architecture are the ventilator and the sink, while the transient nodes are the workers. This means that each worker connects its PULL socket to the ventilator and its PUSH socket to the sink. This way, we can start and stop as many workers as we want without changing any parameter in the ventilator or the sink.



To keep this example simple, we haven't included a mechanism to stop the ventilator or the workers once a solution is found. As a result, the workers will continue hashing all possible strings (even after the correct one has been discovered) until the entire search space for the given maximum length is exhausted. Naturally, this is wasteful and far from optimal. You'll find an exercise at the end of this chapter that invites you to implement a halting mechanism. Since you're already looking at the current network diagram, this is a good moment to share a few contextual hints to help you design such a solution. There are several ways to approach this problem. One option is to introduce a publish/subscribe channel where the sink acts as the publisher and both the workers and the ventilator subscribe to it. When the sink receives a valid solution, it can broadcast a "solution found" event. The ventilator can then stop generating new tasks, and the workers can stop requesting or processing jobs, allowing the system to shut down gracefully.

Implementing the producer

Our producer needs a way to divide the solution space into manageable chunks that can be distributed to the workers. To do this effectively, we need a mechanism that is both consistent and expressive: one that allows each worker to understand exactly which portion of the solution space it has been assigned to and generate all possible strings within that range.

To represent these intervals of string variations, we'll use an **indexed n-ary tree**. Imagine a tree where each node has exactly n children, and each child represents one of the n characters in a given **alphabet**. By assigning a progressive, unique index to each node using breadth-first order, we can create a direct mapping between strings and their positions in the tree. For example, with the alphabet $[a, b]$, the resulting tree would look like this:

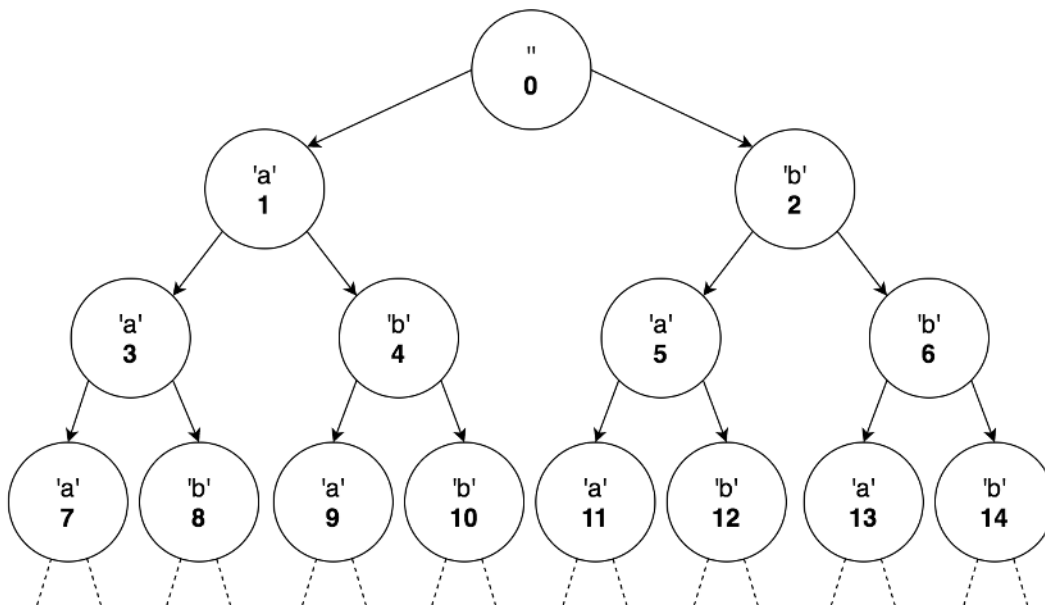


Figure 13.18 – Indexed n-ary tree for alphabet $[a, b]$

With this clever data structure, we can now determine the string variation corresponding to a specific index by traversing the tree from the root to the target node. As we move through the tree, we build the variation by appending the character at each node along the path.

For example, in the tree shown in *Figure 13.18*, the variation at index 13 is 'bba'. We start at the root (index 0, character ''), then move to the right child (index 2, character 'b'), then right again (index 6, character 'b'), and finally to the left child (index 13, character 'a'). The characters collected along the way form the final string.

To help calculate string variations based on their index in the n-ary tree, we'll use the indexed-string-variation package (nodejsdp.link/indexed-string-variation). This library provides an efficient implementation of the data structure we introduced earlier.



indexed-string-variation keeps the tree virtual, rather than generating the entire structure upfront. The library computes branches on the fly to keep memory usage low. If you're curious about the math behind this approach, feel free to explore the source code and see how it works internally.

In our setup, all string generation happens inside the workers, so the ventilator only needs to produce and distribute index intervals. Each worker can then independently compute all the string variations that fall within its assigned range.

Now that we've covered all the necessary theory, it's finally time to start building our system. We'll begin with a core part of the ventilator: the code responsible for generating and distributing tasks to the workers. This logic is implemented in the `generateTasks.js` file:

```
export function* generateTasks(searchHash, alphabet, maxWordLength,
  batchSize) {
  const alphabetLength = BigInt(alphabet.length)
  const maxWordLengthBigInt = BigInt(maxWordLength)
  let nVariations = 0n
  for (let n = 1n; n <= maxWordLengthBigInt; n++) {
    nVariations += alphabetLength ** n
  }
  console.log(
    `Finding the hashsum source string over ${nVariations} `+
    `possible variations`
  )

  let batchStart = 1n
  while (batchStart <= nVariations) {
    const expectedBatchSize = batchStart + BigInt(batchSize) - 1n
    const batchEnd =
      expectedBatchSize > nVariations ? nVariations : expectedBatchSize
```

```

    yield JSON.stringify({
      searchHash,
      alphabet: alphabet,
      // convert BigInt to string for JSON serialization
      batchStart: batchStart.toString(),
      batchEnd: batchEnd.toString(),
    })

    batchStart = batchEnd + 1n
  }
}

```

The `generateTasks()` generator function partitions the entire solution space of possible string variations (based on a given alphabet and `maxWordLength`) into batches of fixed size (`batchSize`). Each batch is a task that can be provided to a worker. We encapsulate all the details of a task in a JSON-encoded object containing:

- The `searchHash` to match against
- The `alphabet` used to build strings
- The `batchStart` and `batchEnd` indexes that define the range of string variations to explore

Every time the generator is invoked, it yields a new task that represents the next chunk of the solution space to explore. This approach will allow us to distribute independent chunks to worker nodes and generate and check string variations in parallel.



In this version of the code, we use `BigInt` ([nodejsdp.link/bigint](https://nodejs.org/docs/latest/builtin-modules/docs/ES2020/bigint)) by default to represent string variation indexes. This allows us to go beyond the limit imposed by `Number.MAX_SAFE_INTEGER` (which is $2^{53} - 1$), ensuring we can handle very large search spaces without loss of precision. However, keep in mind that `BigInt` values cannot be directly serialized to JSON. For this reason, we convert them to strings before serialization, and we will need to manually convert them back to `BigInt` after deserialization.

Now, we need to implement the logic of our producer (the ventilator), which is responsible for distributing the tasks across all workers (in the `producer.js` file):

```

import zmq from 'zeromq' // v6.3.0
import { generateTasks } from './generateTasks.js'

const ALPHABET = 'abcdefghijklmnopqrstuvwxyz'
const BATCH_SIZE = 10000

const [, , maxLength, searchHash] = process.argv

const ventilator = new zmq.Push() // 1

```

```
await ventilator.bind('tcp://*:5016')

const generatorObj = generateTasks(searchHash, ALPHABET, maxLength, BATCH_SIZE)
for (const task of generatorObj) {
  await ventilator.send(task) // 2
}
```

To avoid generating too many variations, our generator uses only the lowercase letters of the English alphabet and sets a limit on the size of the words generated. This limit is provided as an input in the command-line arguments (`maxLength`) together with the hashsum to match (`searchHash`).

But the part that we are most interested in analyzing is how we distribute the tasks across the workers:

1. We first create a PUSH socket and we bind it to the local port 5016, which is where the PULL socket of the workers will connect to receive their tasks.
2. For each generated task, we send it to a worker using the `send()` function of the ventilator socket. Each connected worker will receive a different task following a round-robin approach.



Note that the `send()` function queues the message immediately if possible and returns a resolved promise. However, if the message cannot be queued because an internal high water mark limit has been reached, it waits asynchronously, effectively pausing our generation loop. The promise resolves only when the message has been successfully queued, which allows the loop to resume. This behavior ensures that the rate of task generation on the ventilator side stays in sync with the rate of task consumption on the workers' side. Without this built-in backpressure mechanism, the ventilator could produce tasks faster than they can be processed, causing memory usage to grow unchecked and potentially leading to an out-of-memory crash.

It's also worth noting how elegant and concise this solution is. By combining a `for...of` loop over a synchronous iterator with an awaited promise inside the loop, we express a surprisingly advanced and powerful control flow in just a few lines of clear, readable code. At this point in the book, you should appreciate how all the lessons and patterns we've covered in earlier chapters become even more powerful when carefully combined. This is where design choices start to compound, and the true strength of thoughtful architecture begins to shine.

Implementing the worker

Now it's time to implement the worker, but first, let's create a component to process the incoming tasks (in the `processTask.js` file):

```
import isv from 'indexed-string-variation' // v2.0.1
import { createHash } from 'node:crypto'

export function processTask(task) {
  const strings = isv({
```



```

    alphabet: task.alphabet,
    from: BigInt(task.batchStart),
    to: BigInt(task.batchEnd),
  })

  let first
  let last
  for (const string of strings) {
    if (!first) {
      first = string
    }

    const digest = createHash('sha1').update(string).digest('hex')

    if (digest === task.searchHash) {
      console.log(`>> Found: ${string} => ${digest}`)
      return string
    }
    last = string
  }
  console.log(
    `Processed ${first}..${last} (${task.batchStart}..${task.batchEnd})`
  )
}

```

The logic of the `processTask()` function is quite simple: it iterates over the indexes within the given interval, then for each index it generates the corresponding variation of characters (string). Next, it calculates the SHA1 checksum for the string, and it tries to match it against the `searchHash` passed with the task object. If the two digests match, then it returns the source string to the caller. In the process, this function also prints useful debug information like the first and the last strings tested within the current task.

Now we are ready to implement the main logic of our worker (`worker.js`):

```

import zmq from 'zermq' // v6.3.0
import { processTask } from './processTask.js'

const fromVentilator = new zmq.Pull()
const toSink = new zmq.Push()

fromVentilator.connect('tcp://localhost:5016')
toSink.connect('tcp://localhost:5017')

```

```

for await (const rawMessage of fromVentilator) {
  const found = processTask(JSON.parse(rawMessage.toString()))
  if (found) {
    console.log(`Found! => ${found}`)
    await toSink.send(`Found: ${found}`)
    break
  }
}

```

As we said, a worker represents a transient node in our architecture; therefore, its sockets should connect to a remote node instead of listening for incoming connections. That's exactly what we do in our worker; we create two sockets:

- A PULL socket that connects to the ventilator, for receiving the tasks
- A PUSH socket that connects to the sink, for propagating the results

Besides this, the job done by our worker is very simple: it processes every task received (using a `for...await` loop since `fromVentilator` is an async iterator), and if a match is found, it sends a message to the results collector through the `toSink` socket.

Implementing the results collector

For our example, the results collector (sink) is a very basic program that simply prints the messages received by the workers to the console. The contents of the `collector.js` file are as follows:

```

import zmq from 'zeromq' // v6.3.0

const sink = new zmq.Pull()
await sink.bind('tcp://*:5017')

for await (const rawMessage of sink) {
  console.log('Message from worker: ', rawMessage.toString())
}

```

It's interesting to see that the results collector (as the producer) is also a durable node of our architecture, and therefore, we bind its PULL socket instead of connecting it explicitly to the PUSH socket of the workers.

Running the application

We are now ready to launch our application; let's start a couple of workers and the results collector (each one in a different terminal):

```

node worker.js
node worker.js
node collector.js

```

Then, it's time to start the producer, specifying the maximum length of the words to generate and the SHA1 checksum that we want to match. The following is a sample command line:

```
node producer.js 4 f8e966d1e207d02c44511a58dccff2f5429e9a3b
```

When the preceding command is run, the producer will start generating tasks and distributing them to the set of workers we started. We are telling the producer to generate all possible words with four lowercase letters (because our alphabet comprises only lowercase letters), and we also provide a sample SHA1 checksum that corresponds to a secret four-letter word.

The results of the computation, if any, will appear in the terminal of the results collector application.



Please note that given the low-level nature of PUSH/PULL sockets in ZeroMQ and, in particular, the lack of message acknowledgments, if a node crashes, then all the tasks it was processing will be lost. It's possible to implement a custom acknowledgment mechanism on top of ZeroMQ, but we'll leave that as an exercise for the reader.

Pipelines and competing consumers in AMQP

In the previous section, we saw how a parallel pipeline can be implemented in a peer-to-peer context. Now, we are going to explore this pattern when applied in a broker-based architecture using RabbitMQ.

Point-to-point communications and competing consumers

In a peer-to-peer configuration, a pipeline is a very straightforward concept to imagine. With a message broker in the middle, though, the relationships between the various nodes of the system are a little bit harder to understand: the broker itself acts as an intermediary for our communications and, often, we don't really know who is on the other side listening for messages. For example, when we send a message using AMQP, we don't deliver it directly to its destination, but instead to an exchange and then to a queue. Finally, it will be for the broker to decide where to route the message, based on the rules defined in the exchange, the bindings, and the destination queues.

If we want to implement a pipeline and a task distribution pattern using a system such as AMQP, we must make sure that each message is received by only one consumer, but this is impossible to guarantee if an exchange can potentially be bound to more than one queue. The solution, then, is to send a message directly to the destination queue, bypassing the exchange altogether. This way, we can make sure that only one queue will ever receive the message. This communication pattern is called **point to point**.

Once we can send a set of messages directly to a single queue, we are already halfway to implementing our task distribution pattern. In fact, the next step comes naturally: when multiple consumers are listening on the same queue, the messages will be distributed evenly across them, following a fan-out distribution pattern. As we already mentioned, in the context of message brokers, this is better known as the **Competing Consumers** pattern.

Next, we are going to reimplement our simple hashsum cracker using AMQP, so we can appreciate the differences from the peer-to-peer approach we have discussed in the previous section.

Implementing the hashsum cracker using AMQP

We just learned that exchanges are the point in a broker where a message is multicast to a set of consumers, while queues are the place where messages are load-balanced. With this knowledge in mind, let's now implement our brute-force hashsum cracker on top of an AMQP broker (which in our case is RabbitMQ). The following figure gives you an overview of the system we want to implement:

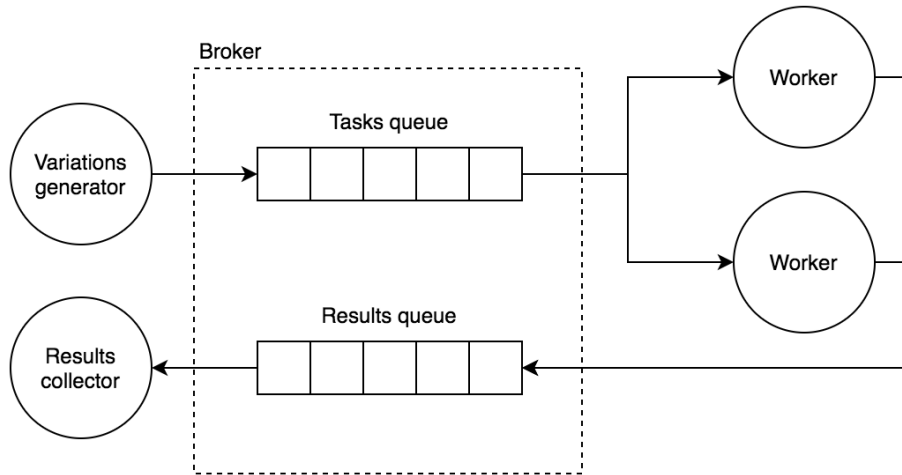


Figure 13.19 – Task distribution architecture using an MQ broker

As we discussed, to distribute a set of tasks across multiple workers, we need to use a single queue. In Figure 13.19, we called this the *tasks queue*. On the other side of the tasks queue, we have a set of workers, which are *competing consumers*: in other words, each one will receive a different message from the queue. The effect is that multiple tasks will execute in parallel on different workers.

The results generated by the workers are published into another queue, which we called the *results queue*, and then consumed by the results collector, which is equivalent to a sink. In the entire architecture, we don't make use of any exchange; we only send messages directly to their destination queue, implementing a point-to-point type of communication.

Implementing the producer

Let's see how to implement such a system, starting from the producer (in the `producer.js` file):

```
import amqp from 'amqplib' // v0.10.8
import { generateTasks } from './generateTasks.js'

const ALPHABET = 'abcdefghijklmnopqrstuvwxyz'
const BATCH_SIZE = 10000

const [, , maxLength, searchHash] = process.argv

const connection = await amqp.connect('amqp://localhost')
```

```

const channel = await connection.createConfirmChannel() // 1
await channel.assertQueue('tasks_queue')

const generatorObj = generateTasks(searchHash, ALPHABET, maxLength,
  BATCH_SIZE)
for (const task of generatorObj) {
  console.log(`Sending task: ${task}`)
  await channel.sendToQueue('tasks_queue', Buffer.from(task)) // 2
}

await channel.waitForConfirms()
channel.close()
connection.close()

```

As we can see, the absence of any exchange or binding makes the setup of an AMQP-based application much simpler. There are, however, a few details to note:

1. Instead of creating a standard channel, we are creating a `confirmChannel`. This is necessary as it creates a channel with some extra functionality; in particular, it provides the `waitForConfirms()` function that we will use later in the code to wait until the broker confirms the reception of all the messages. This is necessary to prevent the application from closing the connection to the broker too soon, before all the messages have been dispatched from the local queue.
2. The core of the producer is the `channel.sendToQueue()` API, which is an AMQP API we haven't seen before. As its name says, that's the API responsible for delivering a message straight to a queue, `tasks_queue` in our example, bypassing any exchange or routing.

Implementing the worker

On the other side of `tasks_queue`, we have the workers listening for the incoming tasks. Let's update the code of our existing `worker.js` module to use AMQP:

```

import amqp from 'amqplib' // v0.10.8
import { processTask } from './processTask.js'

const connection = await amqp.connect('amqp://localhost')
const channel = await connection.createChannel()
const { queue } = await channel.assertQueue('tasks_queue')

channel.prefetch(1) // Ensure only one message is processed at a time
channel.consume(queue, async rawMessage => {
  const found = processTask(JSON.parse(rawMessage.content.toString()))
  await channel.ack(rawMessage)
  if (found) {
    console.log(`Found! => ${found}`)
  }
})

```

```
    await channel.sendToQueue('results_queue',
        Buffer.from(`Found: ${found}`))
    // shuts down the worker
    await channel.close()
    await connection.close()
  }
})
```

Our new worker is also very similar to the one we implemented in the previous section using ZeroMQ, except for the parts related to the exchange of messages. In the preceding code, we can see how we first get a reference to the queue called `tasks_queue` and then we start listening for incoming tasks using `channel.consume()`. Then, every time a match is found, we send the result to the collector via `results_queue`, again using point-to-point communication. It's also important to note how we are acknowledging every message with `channel.ack()` after the message has been completely processed.

If multiple workers are started, they will all listen on the same queue, resulting in the messages being load-balanced between them (they become *competing consumers*).

Implementing the results collector

The results collector is again a trivial module, simply printing any message received to the console. This is implemented in the `collector.js` file, as follows:

```
import amqp from 'amqplib' // v0.10.8

const connection = await amqp.connect('amqp://localhost')
const channel = await connection.createChannel()
const { queue } = await channel.assertQueue('results_queue')

channel.consume(queue, async msg => {
  console.log(`Message from worker: ${msg.content.toString()}`)
  await channel.ack(msg)
})
```

Running the application

Now, everything is ready to give our new system a try. First, make sure that the RabbitMQ server is running. Then, you can launch a couple of workers (in two separate terminals), which will both connect to the same queue (`tasks_queue`) so that every message will be load-balanced between them:

```
node worker.js
node worker.js
```

Then, you can run the collector module and then the producer (by providing the maximum word length and the hash to crack):

```
node collector.js
node producer.js 4 f8e966d1e207d02c44511a58dcccff2f5429e9a3b
```

With this, we have implemented a message pipeline and the Competing Consumers pattern using AMQP.



Our AMQP-based hash-sum cracker runs slightly slower than the ZeroMQ version, showing a common tradeoff: broker-based systems add some latency but provide reliability and features out of the box. For example, if a worker crashes, AMQP ensures the task is re-delivered, while in ZeroMQ, you would have to implement that yourself. However, AMQP can make certain optimizations harder, such as stopping all pending tasks once a match is found. A workaround is to create a unique queue for each run and delete it when finished. In the end, it is about weighing a small performance cost against the simplicity, fault tolerance, and robustness a broker offers.

Now, let's consider another broker-based approach for implementing task distribution patterns, this time built on top of Redis streams.

Distributing tasks with Redis streams

After seeing how the Task Distribution pattern can be implemented using ZeroMQ and AMQP, we are now going to see how we can implement this pattern leveraging Redis streams.

Redis consumer groups

Before diving into some code, we need to learn about a critical feature of Redis that allows us to implement a Task Distribution pattern using Redis streams. This feature is called **consumer groups** and is an implementation of the Competing Consumer pattern (with the addition of some useful accessories) on top of Redis streams.

A consumer group is a stateful entity, identified by a name, that comprises a set of consumers identified by name. When the consumers in the group try to read the stream, they will receive the records in a round-robin configuration.

Each record must be explicitly acknowledged; otherwise, the record will be kept in a *pending* state. Each consumer can only access its own history of pending records unless it explicitly *claims* the records of another consumer. This is useful if a consumer crashes while processing a record. When the consumer comes back online, the first thing it should do is retrieve its list of pending records and process those before requesting new records from the stream. *Figure 13.20* provides a visual representation of how consumer groups work in Redis.

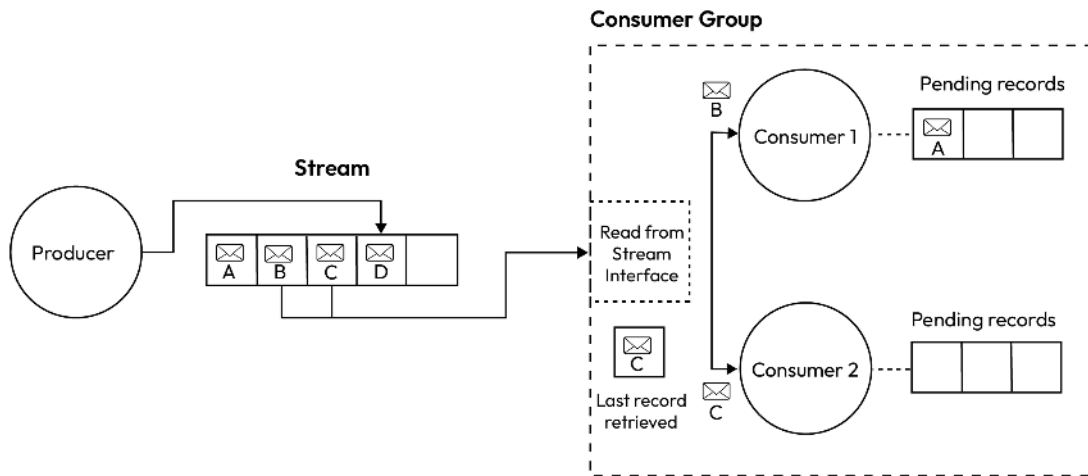


Figure 13.20 – A Redis stream consumer group

We can note how the two consumers in the group receive two different records (B for Consumer 1 and C for Consumer 2) when they try to read from the stream. The consumer group also stores the ID of the last retrieved record (record C), so that at the successive read operation, the consumer group knows what the next record to read is. We can also note how Consumer 1 has a pending record (A), which is a record that it's still processing or couldn't process. Consumer 1 can implement a retry algorithm to make sure to process all the pending records assigned to itself.



A Redis stream can have multiple consumer groups. This way, it's possible to simultaneously apply different types of processing to the same data.

Now, let's put into practice what we have just learned about Redis consumer groups to implement our hashsum cracker.

Implementing the hashsum cracker using Redis streams

The architecture of our hashsum cracker with Redis streams is going to closely resemble that of the previous AMQP example. In fact, we are going to have two different streams (in the AMQP examples, they were queues): one stream to hold the tasks to be processed (`tasks_stream`) and another stream to hold the results coming from the workers (`results_stream`).

Then, we are going to use a consumer group to distribute the tasks from `tasks_stream` to the workers of our application (our workers are the consumers).

Implementing the producer

Let's start by implementing the producer (in the `producer.js` file):

```
import Redis from 'ioredis' // v5.6.1
import { generateTasks } from './generateTasks.js'

const ALPHABET = 'abcdefghijklmnopqrstuvwxyz'
const BATCH_SIZE = 10000

const [, , maxLength, searchHash] = process.argv

const redisClient = new Redis()

const generatorObj = generateTasks(searchHash, ALPHABET, maxLength,
  BATCH_SIZE)
for (const task of generatorObj) {
  console.log(`Sending task: ${task}`)
  await redisClient.xadd('tasks_stream', '*', 'task', task)
}

redisClient.disconnect()
```

As we can see, there is nothing new to us in the implementation of the new `producer.js` module. In fact, we already know very well how to add records to a stream; all we must do is invoke `xadd()`, as discussed in the *Reliable messaging with streams* section.

Implementing the worker

Next, we need to adapt our worker so it can interface with a Redis stream using a consumer group. This is the core of all the architecture, as in here, in the worker, we leverage consumer groups and their features. So, let's implement the new `worker.js` module:

```
import Redis from 'ioredis' // v5.6.1
import { processTask } from './processTask.js'

const redisClient = new Redis()
const [, , consumerName] = process.argv

await redisClient // 1
  .xgroup('CREATE', 'tasks_stream', 'workers_group', '$', 'MKSTREAM')
  .catch(() => console.log('Consumer group already exists'))

const [, , records] = await redisClient.xreadgroup( // 2
  'GROUP',
```

```
'workers_group',
consumerName,
'STREAMS',
'tasks_stream',
'0'
)
for (const [recordId, [, rawTask]] of records) {
  await processAndAck(recordId, rawTask)
}

while (true) {
  const [[, records]] = await redisClient.xreadgroup( // 3
    'GROUP',
    'workers_group',
    consumerName,
    'BLOCK',
    '0',
    'COUNT',
    '1',
    'STREAMS',
    'tasks_stream',
    '>'
  )
  for (const [recordId, [, rawTask]] of records) {
    await processAndAck(recordId, rawTask)
  }
}

async function processAndAck(recordId, rawTask) { // 4
  const found = processTask(JSON.parse(rawTask))
  if (found) {
    console.log(`Found! => ${found}`)
    await redisClient.xadd('results_stream', '*', 'result',
      `Found: ${found}`)
  }

  await redisClient.xack('tasks_stream', 'workers_group', recordId)
}
```

OK, there are a lot of moving parts in the new worker code. So, let's analyze it one step at a time:

1. First, we need to make sure that the consumer group exists before we can use it. We can do that with the `xgroup` command, which we invoke with the following parameters:
 - `'CREATE'` is the keyword to use when we want to create a consumer group. In fact, with the `xgroup` command, we can also destroy the consumer group, remove a consumer, or update the last read record ID, using different subcommands.
 - `'tasks_stream'` is the name of the stream we want to read from.
 - `'workers_group'` is the name of the consumer group.
 - The fourth argument represents the record ID from where the consumer group should start consuming records from the stream. Using `'$'` (dollar sign) means that the consumer group should start reading the stream from the ID of the last record currently in the stream.
 - `'MKSTREAM'` is an extra parameter that instructs Redis to create the stream if it doesn't exist already.



Note that, since this command fails if the consumer group already exists, we are using `catch()` to capture the error and log a useful message. This avoids our application crashing (due to an unhandled promise rejection) if the consumer group already exists.

2. Next, we read all the pending records belonging to the current consumer. Those are the left-over records from a previous run of the consumer that weren't processed because of an abrupt interruption of the application (such as a crash). If the same consumer (with the same name) terminated properly during the last run, without errors, then this list would be empty. As we already mentioned, each consumer has access only to its own pending records. We retrieve this list with an `xreadgroup` command and the following arguments:
 - `'GROUP', 'workers_group', consumerName` is a mandatory trio where we specify the name of the consumer group (`'workers_group'`) and the name of the consumer (`consumerName`) that we read from the command-line inputs.
 - Then we specify the stream we would like to read from with `'STREAMS', 'tasks_stream'`.
 - Finally, we specify `'0'` as the last argument, which is the ID from which we should start reading. Essentially, we are saying that we want to read all pending messages belonging to the current consumer starting from the first message.

3. Then, we have another call to `xreadgroup()`, but this time it has a completely different semantic. In this case, in fact, we want to start reading new records from the stream (and not access the consumer's own history). This is possible with the following list of arguments:
 - As in the previous call of `xreadgroup()`, we specify the consumer group that we want to use for the read operation with the three arguments: `'GROUP'`, `'workers_group'`, `consumerName`.
 - Then, we indicate that the call should block if there are no new records currently available instead of returning an empty list. We do that with the following two arguments: `'BLOCK'`, `'0'`. The last argument is the timeout after which the function returns anyway, even without results. `'0'` means that we want to wait indefinitely.
 - The next two arguments, `'COUNT'` and `'1'`, tell Redis that we are interested in getting one record per call.
 - Next, we specify the stream we want to read from with `'STREAMS'`, `'tasks_stream'`.
 - Finally, with the special ID `'>'` (greater than symbol), we indicate that we are interested in any record not yet retrieved by this consumer group.
4. Finally, in the `processAndAck()` function, which we call for every message consumed from Redis, we run our hashsum cracker against the current task and check whether we have a match. If that's the case, we append a new record to `results_stream`. At last, when all the processing for the record returned by `xreadgroup()` completes, we invoke the Redis `xack` command to acknowledge that the record has been successfully consumed, which results in the record being removed from the pending list for the current consumer.

Phew! There was a lot going on in the `worker.js` module. It's interesting to note that most of the complexity comes from the large number of arguments required by the various Redis commands.

Implementing the results collector

Now, the last component we need to implement is the results collector.

Let's see what the code looks like (in the `collector.js` file):

```
import Redis from 'ioredis' // v5.6.1

const redisClient = new Redis()

let lastRecordId = '$'
while (true) {
  const data = await redisClient.xread(
    'BLOCK',
    '0',
    'STREAMS',
    'results_stream',
    lastRecordId
  )
}
```

```
)  
for (const [, logs] of data) {  
  for (const [recordId, [, message]] of logs) {  
    console.log(`Message from worker: ${message}`)  
    lastRecordId = recordId  
  }  
}  
}  
}
```

The collector continuously reads messages from `results_stream` using the `XREAD` command. It starts from the most recent entry (`'$'`) and blocks until new messages are available. Each time a new message is received, it logs the message content to the console and updates `lastRecordId` so that the next read resumes from where it left off. This effectively creates a simple, persistent stream consumer that listens for messages from workers in real time.



You may be surprised to know that this example just scratches the surface, as there is a lot more to know about Redis streams and consumer groups. Check out the official Redis introduction to streams for more details at nodejsdp.link/redis-streams.

Running the application

Now, everything should be ready for us to try out this new version of the hashsum cracker. Let's start a couple of workers, but this time remember to assign them a name, which will be used to identify them in the consumer group:

```
node worker.js workerA  
node worker.js workerB
```

Then, you can run the collector and the producer as we did in the previous examples:

```
node collector.js  
node producer.js 4 f8e966d1e207d02c44511a58dccff2f5429e9a3b
```

This concludes our exploration of the task distribution patterns, so now, we'll take a closer look at the request/reply patterns.

Request/Reply patterns

One-way communication offers great parallelism and efficiency, but it cannot solve every integration need. Sometimes a request/reply pattern is the better fit. When only an asynchronous one-way channel is available, we can still build abstractions to enable request/reply messaging. That is what we will explore next.

Correlation Identifier

The first Request/Reply pattern that we are going to learn is called **Correlation Identifier**, and it represents the basic block for building a request/reply abstraction on top of a one-way channel.

The pattern involves marking each request with an identifier, which is then attached to the response by the receiver; this way, the sender of the request can correlate the two messages and return the response to the right handler. This elegantly solves the problem in the context of a one-way asynchronous channel, where messages can travel in any direction at any time. Let's look at the example in the following diagram:

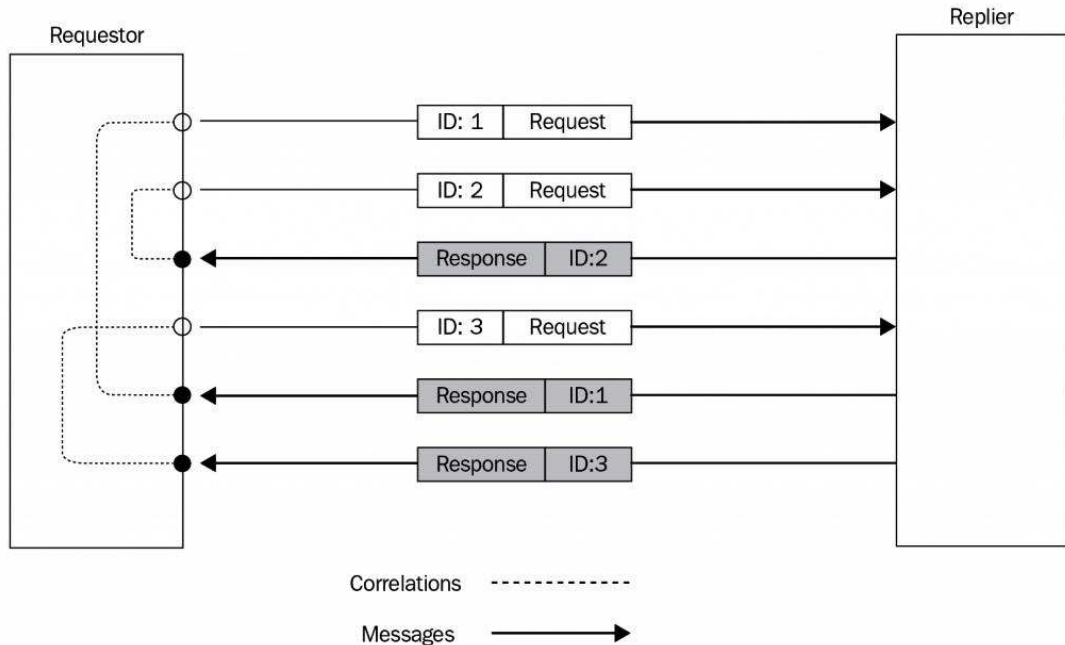


Figure 13.21 – Request/reply message exchange using correlation identifiers

The scenario depicted in *Figure 13.21* shows how using a correlation ID allows us to match each response with the right request, even if those are sent and then received in a different order. The way this works will be much clearer once we start working on our next example.

Implementing a request/reply abstraction using correlation identifiers

In this section, we are going to build a *request/reply abstraction* on top of a simple **point-to-point duplex channel**, which directly connects two nodes in the system and allows messages to flow in both directions.

A common example of this kind of channel is a WebSocket connection, which establishes a point-to-point link between a server and a browser so that messages can travel either way. Another example (one we have already explored in *Chapter 11, Advanced Recipes*) is the communication channel created when a child process is spawned using `child_process.fork()`. This channel is asynchronous, point to point, and duplex, connecting the parent only with the child process and allowing messages to move in both directions. It is probably the most basic example of this category, which makes it ideal for our implementation.

Our goal is to wrap this parent-child process channel with an abstraction that supports asynchronous request/reply communication. The abstraction will automatically tag each outgoing request with a unique **correlation identifier** and then use this identifier to match incoming replies with the correct pending request handler. This ensures that responses are correctly associated with their requests, even when replies arrive out of order.

As we saw in *Chapter 11, Advanced Recipes*, the parent process can send messages to a child using `child.send(message)` and receive them with `child.on('message', callback)`. Similarly, the child process can send messages back with `process.send(message)` and receive them with `process.on('message', callback)`. Because the messaging interface is identical on both sides, we can build a single abstraction that works equally well in the parent and the child.

Abstracting the request

Let's start building this abstraction by considering the part responsible for sending new requests. Let's create a new file called `createRequestChannel.js` with the following content:

```
import { nanoid } from 'nanoid' // v5.1.5

export function createRequestChannel(channel) { // 1
  const correlationMap = new Map()

  function sendRequest(data) { // 2
    console.log('Sending request', data)
    return new Promise((resolve, reject) => {
      const correlationId = nanoid()

      const replyTimeout = setTimeout(() => {
        correlationMap.delete(correlationId)
        reject(new Error('Request timeout'))
      }, 10000)

      correlationMap.set(correlationId, replyData => {
        correlationMap.delete(correlationId)
        clearTimeout(replyTimeout)
        resolve(replyData)
      })

      channel.send({
        type: 'request',
        data,
        id: correlationId,
      })
    })
  }
}
```

```

    }

    channel.on('message', message => { // 3
      const replyCb = correlationMap.get(message.inReplyTo)
      if (replyCb) {
        replyCb(message.data)
      }
    })
    return sendRequest
  }

```

This is how our request abstraction works:

1. `createRequestChannel()` is a factory that wraps the input channel and returns a `sendRequest()` function used to send a request and receive a reply. The magic of the pattern lies in the `correlationMap` variable, which stores the association between the outgoing requests and their reply handlers (effectively a callback to be invoked when the reply arrives).
2. The `sendRequest()` function is used to send new requests. Its job is to generate a unique correlation ID using the `nanoid` package (`nodejsdp.link/nanoid`) and then wrap the request data in an envelope that allows us to specify the correlation ID and the type of the message. The correlation ID and the handler responsible for returning the reply data to the caller (which uses `resolve()` under the hood) are then added to `correlationMap` so that the handler can be retrieved later using the correlation ID. We also implemented a very simple request timeout logic.
3. When the factory is invoked, we also start listening for incoming messages. If the correlation ID of the message (contained in the `inReplyTo` property) matches any of the IDs contained in the `correlationMap` map, we know that we just received a reply, so we obtain the reference to the associated response handler and we invoke it with the data contained in the message.

That's it for the `createRequestChannel.js` module. Let's move on to the next part.

Abstracting the reply

We are just a step away from implementing the full pattern, so let's see how the counterpart of the request channel, which is the reply channel, works. Let's create another file called `createReplyChannel.js`, which will contain the abstraction for wrapping the reply handler:

```

export function createReplyChannel(channel) {
  return function registerHandler(handler) {
    channel.on('message', async message => {
      if (message.type !== 'request') {
        return
      }

      const replyData = await handler(message.data)
      channel.send({

```



```
    type: 'response',
    data: replyData,
    inReplyTo: message.id,
  })
})
}
```

Our `createReplyChannel()` function is again a factory that returns another function used to register new reply handlers. This is what happens when a new handler is registered:

1. When we receive a new request, we immediately invoke the handler by passing the data contained in the message.
2. Once the handler has done its work and returned its reply, we build an envelope around the data and include the type of the message and the correlation ID of the request (the `inReplyTo` property). Then, we put everything back into the channel.

The amazing thing about this pattern is that in Node.js, it comes very easily: everything for us is already asynchronous, so an asynchronous request/reply communication built on top of a one-way channel is not very different from any other asynchronous operation, especially if we build an abstraction to hide its implementation details.

Trying the full request/reply cycle

Now we are ready to try our new asynchronous request/reply abstraction. Let's create a sample *replier* in a file named `replier.js`:

```
import { createReplyChannel } from './createReplyChannel.js'

const registerReplyHandler = createReplyChannel(process)

registerReplyHandler(req => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({ sum: req.a + req.b })
    }, req.delay)
  })
})

process.send('ready')
```

Our replier simply calculates the sum of the two numbers received in the request and returns the result after a certain delay (which is also specified in the request). This will allow us to verify that the order of the responses can be different from the order in which we sent the requests, to confirm that our pattern is working. With the last instruction of the module, we send a message back to the parent process to indicate that the child is ready to accept requests.

The final step to complete the example is to create the requestor in a file named `requestor.js`, which also has the task of starting the replier using `child_process.fork()`:

```
import { fork } from 'node:child_process'
import { join } from 'node:path'
import { once } from 'node:events'
import { createRequestChannel } from './createRequestChannel.js'

const channel = fork(join(import.meta.dirname, 'replier.js')) // 1
const request = createRequestChannel(channel)

try {
  const [message] = await once(channel, 'message') // 2
  console.log(`Child process initialized: ${message}`)
  const p1 = request({ a: 1, b: 2, delay: 500 }).then(res => { // 3
    console.log(`Reply: 1 + 2 = ${res.sum}`)
  })

  const p2 = request({ a: 6, b: 1, delay: 100 }).then(res => { // 4
    console.log(`Reply: 6 + 1 = ${res.sum}`)
  })

  await Promise.all([p1, p2]) // 5
} finally {
  channel.disconnect() // 6
}
```

The requestor starts the replier (1) and then passes its reference to our `createRequestChannel()` abstraction. We then wait for the child process to be available (2) and run a couple of sample requests (3, 4). Finally, we wait for both requests to complete (5) and we disconnect the channel (6) to allow the child process (and therefore the parent process) to exit gracefully. To try out the sample, simply launch the `requestor.js` module. The output should be something like the following:

```
Child process initialized: ready
Sending request { a: 1, b: 2, delay: 500 }
Sending request { a: 6, b: 1, delay: 100 }
Reply: 6 + 1 = 7
Reply: 1 + 2 = 3
```

This confirms that our implementation of the Request/Reply messaging pattern works perfectly and that the replies are correctly associated with their respective requests, no matter in what order they are sent or received.

The technique we've discussed in this section works great when we have a single point-to-point channel. But what happens if we have a more complex architecture with multiple channels or queues? That's what we are going to see next.

Return address

The Correlation Identifier enables request/reply over a one-way channel, but it is not enough when multiple channels, queues, or requestors are involved. In these cases, we also need a return address so the replier knows where to send the response.

Implementing the Return Address pattern in AMQP

In the context of an AMQP-based architecture, the return address is the queue where the requestor is listening for incoming replies. Because the response is meant to be received by only one requestor, it's important that the queue is private and not shared across different consumers. From these properties, we can infer that we are going to need a transient queue scoped to the connection of the requestor, and that the replier must establish a point-to-point communication with the return queue to be able to deliver its responses.

The following diagram gives us an example of this scenario:

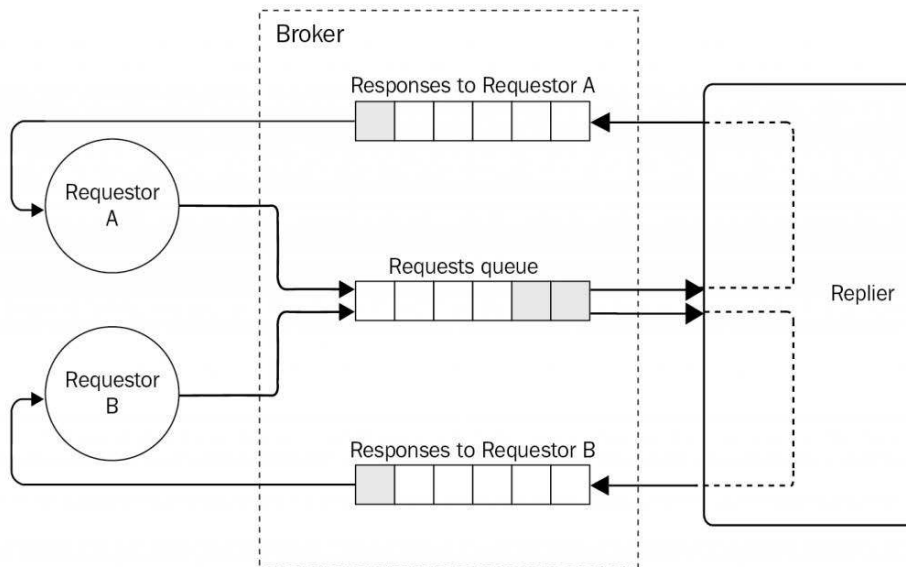


Figure 13.22 – Request/reply messaging architecture using AMQP

Figure 13.22 shows us how each requestor has its own private queue, specifically intended to handle the replies to their requests. All requests are sent instead to a single queue, which is then consumed by the replier. The replier will route the replies to the correct response queue thanks to the *return address* information specified in the request.

In fact, to create a Request/Reply pattern on top of AMQP, all we need to do is to specify the name of the response queue in the message properties, so that the replier knows where the response message must be delivered.

The theory seems very straightforward, so let's see how to implement this in a real application.

Implementing the request abstraction

Let's now build a request/reply abstraction on top of AMQP. We will use RabbitMQ as a broker, but any compatible AMQP broker should do the job. Let's start with the request abstraction, implemented in the `amqpRequest.js` module. We will show the code here one piece at a time to make the explanation easier. Let's start from the constructor of the `AmqpRequest` class:

```
import { nanoid } from 'nanoid' // v5.1.5
import amqp from 'amqplib' // v0.10.8

export class AmqpRequest {
  constructor () {
    this.correlationMap = new Map()
  }
  //...
```

As we can see from the preceding code, we will again be using the Correlation Identifier pattern, so we are going to need a map to hold the association between the message ID and the relative handler.

Then, we need a method to initialize the AMQP connection and its objects:

```
async initialize() {
  this.connection = await amqp.connect('amqp://localhost')
  this.channel = await this.connection.createChannel()
  const { queue } = await this.channel.assertQueue( // 1
    '', { exclusive: true }
  )
  this.replyQueue = queue

  this.channel.consume( // 2
    this.replyQueue,
    msg => {
      const correlationId = msg.properties.correlationId
      const handler = this.correlationMap.get(correlationId)
      if (handler) {
```

```

        handler(JSON.parse(msg.content.toString()))
      }
    },
    { noAck: true }
  )
}

```

The interesting thing to observe here is how we create the queue to hold the replies (1). The peculiarity is that we don't specify any name, which means that a random one will be chosen for us. In addition to this, the queue is *exclusive*, which means that it's bound to the currently active AMQP connection and it will be destroyed when the connection closes. There is no need to bind the queue to an exchange as we don't need any routing or distribution to multiple queues, which means that the messages must be delivered straight into our response queue. In the second part of the function (2), we start to consume the messages from `replyQueue`. Here, we match the ID of the incoming message with the one we have in our `correlationMap` and invoke the associated handler.

Next, let's see how it's possible to send new requests:

```

send(queue, message) {
  return new Promise((resolve, reject) => {
    const id = nanoid() // 1
    const replyTimeout = setTimeout(() => {
      this.correlationMap.delete(id)
      reject(new Error('Request timeout'))
    }, 10000)

    this.correlationMap.set(id, replyData => { // 2
      this.correlationMap.delete(id)
      clearTimeout(replyTimeout)
      resolve(replyData)
    })

    this.channel.sendToQueue( // 3
      queue,
      Buffer.from(JSON.stringify(message)),
      {
        correlationId: id,
        replyTo: this.replyQueue,
      }
    )
  })
}

```

The `send()` method accepts as input the name of the requests queue and the message to send. As we learned in the previous section, we need to generate a correlation ID (1) and associate it with a handler responsible for returning the reply to the caller (2). Finally, we send the message (3), specifying the `correlationId` and the `replyTo` property as metadata. In AMQP, in fact, we can specify a set of properties (or metadata) to be passed to the consumer, together with the main message. The metadata object is passed as the third argument of the `sendToQueue()` method.

This is because we are not interested in implementing a publish/subscribe distribution pattern using exchanges, but a more basic point-to-point delivery straight into the destination queue.

The last piece of our `AmqpRequest` class is where we implement the `destroy()` method, which is used to close the connection and the channel:

```
destroy() {  
  this.channel.close()  
  this.connection.close()  
}
```

That's it for the `amqpRequest.js` module.

Implementing the reply abstraction

Now it's time to implement the reply abstraction in a new module named `amqpReply.js`:

```
import amqp from 'amqplib' // v0.10.8  
  
export class AmqpReply {  
  constructor(requestsQueueName) {  
    this.requestsQueueName = requestsQueueName  
  }  
  
  async initialize() {  
    const connection = await amqp.connect('amqp://localhost')  
    this.channel = await connection.createChannel()  
    const { queue } = await this.channel.assertQueue( // 1  
      this.requestsQueueName  
    )  
    this.queue = queue  
  }  
  
  handleRequests(handler) { // 2  
    this.channel.consume(this.queue, async msg => {  
      const content = JSON.parse(msg.content.toString())  
      const replyData = await handler(content)  
      this.channel.sendToQueue( // 3
```

```

        msg.properties.replyTo,
        Buffer.from(JSON.stringify(replyData)),
        { correlationId: msg.properties.correlationId }
    )
    this.channel.ack(msg)
  })
}
}

```

In the `initialize()` method of the `AmqpReply` class, we create the queue that will receive the incoming requests (1): we can use a simple durable queue for this purpose. The `handleRequest()` method (2) is used to register new request handlers from where new replies can be sent. When sending back a reply (3), we use `channel.sendToQueue()` to publish the message straight into the queue specified in the `replyTo` property of the message (our return address). We also set `correlationId` in the reply, so that the receiver can match the message with the list of pending requests.

Implementing the requestor and the replier

Everything is now ready to give our system a try, but first, let's build a pair sample requestor and replier to see how to use our new abstraction.

Let's start with the `replier.js` module:

```

import { AmqpReply } from './amqpReply.js'

const reply = new AmqpReply('requests_queue')
await reply.initialize()

reply.handleRequests(req => {
  console.log('Request received', req)
  return { sum: req.a + req.b }
})

```

It's nice to see how the abstraction we built allows us to hide all the mechanisms to handle the correlation ID and the return address. All we need to do is initialize a new reply object, specifying the name of the queue where we want to receive our requests ('requests_queue'). The rest of the code is just trivial; in practice, our sample replier simply calculates the sum of the two numbers received as the input and sends back the result in an object.

On the other side, we have a sample requestor implemented in the `requestor.js` file:

```

import { AmqpRequest } from './amqpRequest.js'
import { setTimeout } from 'node:timers/promises'

const request = new AmqpRequest()
await request.initialize()

```

```
async function sendRandomRequest() {
  const a = Math.round(Math.random() * 100)
  const b = Math.round(Math.random() * 100)
  const reply = await request.send('requests_queue', { a, b })
  console.log(`${a} + ${b} = ${reply.sum}`)
}

for (let i = 0; i < 20; i++) {
  await sendRandomRequest()
  await setTimeout(1000)
}

request.destroy()
```

Our sample requestor sends 20 random requests at one-second intervals to the `requests_queue` queue. In this case, also, it's interesting to see that our abstraction is doing its job perfectly, hiding all the details behind the implementation of the asynchronous Request/Reply pattern.

Now, to try out the system, simply run the `replier` module followed by a couple of `requestor` instances:

```
node replier.js
node requestor.js
node requestor.js
```

You will see a set of operations published by the requestors and then received by the replier, which in turn will send back the responses to the right requestor.

Now, we can try other experiments. Once the replier is started for the first time, it creates a durable queue, which means that if we now stop it and then run the replier again, no request will be lost. All the messages will be stored in the queue until the replier is started again!



Note that based on how we implemented the application, a request will time out after 10 seconds. So, in order for a reply to reach the requestor in time, the replier can afford to have only a limited downtime (certainly less than 10 seconds).

Another nice feature that we get for free by using AMQP is the fact that our replier is scalable out of the box. To test this assumption, we can try to start two or more instances of the replier and watch the requests being load-balanced between them. This works because every time a requestor starts, it attaches itself as a listener to the same durable queue, and as a result, the broker will load balance the messages across all the consumers of the queue (remember the Competing Consumers pattern?). Sweet!



ZeroMQ has a pair of sockets specifically meant for implementing request/reply patterns, called REQ/REP; however, they are synchronous (only one request/response at a time). More complex request/reply patterns are possible with more sophisticated techniques. For more information, you can read the official guide at nodejsdp.link/zeromq-reqrep.

A Request/Reply pattern with a return address is also possible on top of Redis streams and very closely resembles the system we implemented with AMQP. We'll leave this to you to implement as an exercise.

Summary

You have reached the end of this chapter, where we explored the most important messaging and integration patterns and their role in designing distributed systems. You should now be confident with the three key message exchange patterns—Publish/Subscribe, Task Distribution, and Request/Reply, and know how to implement them using either a peer-to-peer architecture or a broker. We compared the pros and cons of each approach and saw how brokers, whether based on message queues or data streams, can deliver reliable, scalable applications with minimal effort, at the cost of one more system to maintain.

You also learned how ZeroMQ lets you build distributed systems with complete control, fine-tuning the architecture to your exact needs. Both broker-based and peer-to-peer approaches give you the tools to create anything from simple chat apps to large-scale platforms serving millions.

This chapter also marks the end of the book. By now, you should have a full toolbox of patterns and techniques for real-world projects, along with a deeper understanding of Node.js, its strengths, tradeoffs, and role in modern architectures. You have worked with a rich ecosystem of tools and libraries, built and maintained by passionate developers worldwide. Ultimately, what makes Node.js truly special is its people, a community built on curiosity, generosity, and a shared desire to grow together. Every developer's journey is different, but we all benefit when knowledge is shared, whether through code, documentation, talks, blog posts, or simply helping another developer.

We hope you found value in what we have shared. This is our small contribution to a community that has given us so much, and we look forward to seeing yours.

Sincerely, Luciano Mammino and Mario Casciaro.

Exercises

- **13.1 History service with streams:** In our publish/subscribe example with Redis streams, we didn't need a history service (as we did in the related AMQP example) because all the message history was saved in the stream anyway. Now, implement such a history service, storing all the incoming messages in a separate database, and use this service to retrieve the chat history when a new client connects. Hint: The history service will need to remember the ID of the last message retrieved across restarts.

- **13.2 Multiroom chat:** Update the chat application example we created in this chapter to be able to support multiple chat rooms. The application should also support displaying the message history when the client connects. You can choose the messaging system you prefer, and even mix different ones.
- **13.3 Tasks that stop:** Update the hashsum cracker examples we implemented in this chapter and add the necessary logic to stop the computation on all nodes once a match has been found. There is a note below *Figure 13.17* that gives you some hints.
- **13.4 Reliable task processing with ZeroMQ:** Implement a mechanism to make our hashsum cracker example with ZeroMQ more reliable. As we already mentioned, with the implementation we saw in this chapter, if a worker crashes, all the tasks it was processing are lost. Implement a peer-to-peer queuing system and an acknowledgment mechanism to make sure that the message is always processed at least once (excluding errors due to hypothetical unprocessable tasks).
- **13.5 Data aggregator:** Create an abstraction that can be used to send a request to all the nodes connected to the system, and then returns an aggregation of all the replies received by those nodes. Hint: You can use publish/reply to send the request, and any one-way channel to send back the replies. Use any combination of the technologies you have learned.
- **13.6 Worker status CLI:** Use the data aggregator component defined in *Exercise 13.5* to implement a command-line application that, when invoked, displays the status of all the workers of the hashsum cracker application (for example, which chunk they are processing, whether they found a match, and so on).
- **13.7 Worker status UI:** Implement a web application (from the client to the server) to expose the status of the workers of the hashsum cracker application through a web UI that can report in real time when a match is found.
- **13.8 Pre-initialization queues are back:** In the AMQP request/reply example, we implemented a *Delayed Startup* pattern to deal with the fact that the `initialize()` method is asynchronous. Now, refactor that example by adding pre-initialization queues, as we learned in *Chapter 11, Advanced Recipes*.
- **13.9 Request/reply with Redis streams:** Build a request/reply abstraction on top of Redis streams.
- **13.10 Kafka:** If you are brave enough, try to reimplement all relevant examples in this chapter using Apache Kafka (nodejssdp.link/kafka) instead of Redis streams.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

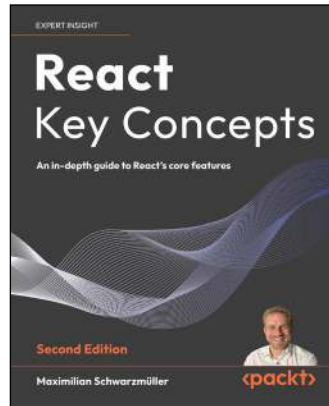
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customer@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

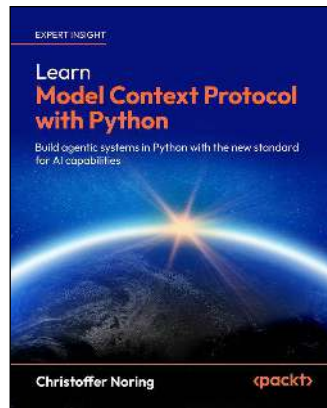


React Key Concepts - Second Edition

Maximilian Schwarzmüller

ISBN: 978-1-83620-227-1

- Build modern, user-friendly, and reactive web apps
- Create components and utilize props to pass data between them
- Handle events, perform state updates, and manage conditional content
- Add styles dynamically and conditionally for modern user interfaces
- Use advanced state management techniques such as React's Context API
- Utilize React Router to render different pages for different URLs
- Understand key best practices and optimization opportunities
- Learn about React Server Components and Server Actions



Learn Model Context Protocol with Python

Christoffer Noring

ISBN: 978-1-80610-323-2

- Understand the MCP protocol and its core components
- Build MCP servers that expose tools and resources to a variety of clients
- Test and debug servers using the interactive inspector tools
- Develop for both LLM and non-LLM clients
- Consume servers using Claude Desktop and Visual Studio Code Agents
- Secure MCP apps, manage and mitigate common threats
- Build and deploy MCP apps using cloud-based strategies

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Node.js Design Patterns — Fourth Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

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

Index

A

- AbortController** 512-514
 - reference link 512
- AbortSignal** 514
- adaptee** 314
- Adapter pattern** 248, 289, 314
 - real-world examples 317
- Advanced Message Queuing Protocol (AMQP)** 621
 - binding 622
 - competing consumers 646
 - exchange 622
 - pipelines 646
 - point-to-point communications 646
 - queue 622
 - reference link 604
 - Return Address pattern, implementing 662, 663
 - used, for implementing hashsum cracker 647
 - used, for implementing history service 624-626
 - used, for integrating chat application 626-629
- Allure**
 - reference link 407
- Amazon Elastic Container Registry**
 - reference link 580
- Amazon Kinesis**
 - reference link 631
- amqplib package**
 - reference link 624
- AMQP model**
 - reference link 622
- Angular**
 - reference link 312
- AngularJS** 22
- Ansible**
 - reference link 561
- Apache Kafka**
 - reference link 631
- API demo server**
 - without caching or batching 502-504
- API gateway** 589
- API orchestration** 590-593
 - reference link 590
- API Orchestration Layer (OL)** 590
- API Orchestrator** 592
- API proxy** 589, 590
- API service** 563
- application scaling** 538
- application structure, E2E tests** 468
 - event page 470
 - home page 469
 - My Reservations page 470
 - sign-in and sign-up forms 469
- archiver package**
 - reference link 214
- Argon2**
 - reference link 638

Arrange-Act-Assert (AAA) pattern 393

arrow functions

versus regular functions 308, 309

assembly 21

async/await 67, 134, 157, 158

concurrent execution 164, 165

error handling 159

limited concurrent execution 166, 167

sequential execution and iteration 162-164

using, with `Array.forEach` for serial execution 164

async generators 368, 369

asynchronicity

with deferred execution 77-80

asynchronous code

testing 424-430

asynchronous communication 602

asynchronous control flow patterns, streams

ordered concurrent execution 227, 228

sequential execution 219-221

unordered concurrent execution 221

unordered limited concurrent execution 225-227

asynchronous CPS 70-72

asynchronously initialized components

dealing with 488

delayed startup 491, 492

issues 488-490

local initialization check 490, 491

pre-initialization queues 492-494

asynchronous module definition (AMD) 22

asynchronous operation 67

asynchronous operations, canceling 509

asynchronous invocations, wrapping 511, 512

cancelable async functions, with `AbortController` 512-516

cancelable functions, creating 509-511

asynchronous programming 67

challenges 100

asynchronous request batching 498-500

asynchronous request caching 500-502

async IIFE (Immediately Invoked Function Expression) 159

async iterables 364

async iterators 364-368

Node.js streams 369, 370

utilities 370-372

Autocannon

reference link 504

Ava

reference link 407

AWS SDK for JavaScript v3

reference link 387

axios

reference link 438

B

Backbone 22

Backend for Frontend (BFF) pattern 593

backpressure 192, 197, 199

batching and caching requests

implementing, with promises 508

batching requests

implementing, with promises 505-507

behavioral design patterns 321

Command 382

Iterator 344

Middleware 373

State 331, 332

Strategy 322-325

Template 341

Behavior-Driven Development (BDD) 392, 396, 397

BigInt

reference link 642

binding 622

Biome

URL 32

BitTorrent 337

black-box tests 401

blocking I/O programming 7, 8

bound function 383

Brotli compression 212

browser automation, E2E tests 472

Browserify

URL 23

browser-level

reference link 317

buffer-based API 178

buffered API

for gzipping 177

buffering

versus streaming 175, 176

Builder pattern 260-264

rules for implementing 264

URL object builder, implementing 265-268

use cases 268-271

built-in Proxy object 296-298

capabilities 298, 299

limitations 299

built-in test mock

HTTP requests mocking 432-435

busy-waiting 9

C

c8

reference link 422

Caddy

reference link 557

callback discipline 106

applying 106-108

callback hell 103, 104

Callback pattern 68

callbacks 67, 68, 133

best practices 105

combining, with events 92-96

conventions 80-84

versus EventEmitter 91, 92

cancelable async functions

with AbortController 512-516

cancelable functions

creating 509-511

cbuffer

reference link 340

Chain of Responsibility pattern 374

reference link 374

Change Observer pattern

with Proxy 301-303

channel 622

chunked encoding

reference link 197

circular dependencies, ECMAScript modules (ES modules)

evaluation 45

instantiation 43, 44

parsing 42

closures 68

reference link 68

cloud-based proxies 557

Cloud design patterns

reference link 549

cluster module 542, 543

application, scaling with 545, 546

availability 547-549

behavior 543

HTTP server, building 544

resiliency 547-549

zero-downtime restart 549-551

code coverage 394

- branch coverage 394
- class coverage 394
- collecting 420, 421
- function coverage 394
- line coverage 394
- statement coverage 394
- visualizing, in code and browser 422

combined stream 229, 230

- implementing 231, 232

combine middleware

- reference link 381

command message 599**Command pattern 382**

- client 382
- command 382
- complex command 383-387
- invoker 382
- target 382

CommonJS 16, 21

- ES modules, importing from 59

CommonJS modules 53

- importing, from ES modules 57, 58

CommonJS specification

- URL 22

competing consumers 636, 646**competitive race 119****complex applications**

- decomposing 583

composability 176

- client-side encryption, adding 184
- server-side decryption, adding 185, 186

composition 305, 306**computing threads**

- reference link 8

concurrency 115, 541

- limiting 122, 123

concurrency, limiting globally 124

- queues, to rescue 124, 125
- TaskQueue, refining 125, 126
- web spider version 4 127-129

concurrent execution 115-117

- pattern 118, 119
- web spider version 3 117, 118

concurrent tasks

- race conditions, fixing 119-121

constructor injection 286**Consul 564**

- reference link 564

consumer 602**consumer groups 650****container image 576****container orchestration tool**

- tasks 578

containers 574

- creating 575

context 322**continuation-passing style (CPS) 68, 99**

- asynchronous 70-72
- synchronous 69, 70

Continuous Delivery (CD) 397**Continuous Deployment 398****Continuous Integration (CI) 397****control flow patterns 109**

- concurrent execution 115-117
- limited concurrent execution 121, 122
- sequential execution 109

Correlation Identifier 656, 657

- used, for implementing request/reply abstraction 657, 658

CouchDB

- reference link 553

CPU-bound tasks 516

- external processes, using 522
- interleaving, with setImmediate 520

- running, in production 534
- subset sum problem, solving 516-520
- worker threads, using 530

crate 21

cross-site request forgery (CSRF) 373

Cross-Site Scripting (XSS) attacks

- reference link 610

csv-parse module

- reference link 206

Cucumber

- URL 397

custom glob patterns, for targeted test execution 417

- tests, filtering by skip and only 418, 419

- tests, filtering by test name 417, 418

Cypress

- reference link 472

D

data aggregation 592

database transaction 461

data partitioning 540

Decorator pattern 289, 304

- implementing, techniques 304

- real-world examples 311, 312

- versus Proxy pattern 313

Decorator pattern implementation techniques 304

- composition 305, 306

- object decoration 306, 307

- using, with Proxy object 309

decoupling 602

default export 29

defensive copies 272

delegates

- reference link 295

delivery semantics 620

- at least once category 620

- at most once category 620

- exactly once category 620

demultiplexer 237

demultiplexing 9, 237

Denial of Service (DoS) attacks 121, 520

dependencies

- mocking 441-445

dependency graph 38

dependency hell problem 3, 35

dependency hoisting 277

dependency injection 446

- versus mocking imports 446-449

Dependency Injection (DI) pattern 282

- used, for wiring modules 282-285

depth-first search (DFS)

- reference link 42

destructuring assignment syntax 58

developer experience (DX) 498

DI container 286

dimensions, scalability

- cloning 539

- data partitioning 540

- decomposing 540

directed graph

- reference link 38

direct style 69

Distributed Denial of Service (DDoS) attacks 520

distributed hashsum cracker, building with ZeroMQ 638

- application, running 645, 646

- building 639, 640

- producer, implementing 640-643

- results collector, implementing 645

- worker, implementing 643-645

Docker

reference link 574

Dockerfile 576**Docker Hub**

reference link 580

Docker registry

reference link 580

Docker Swarm

reference link 558

document message 600**Don't Repeat Yourself (DRY) principle 5****duck typing 138**

reference link 138, 259

duplexer3

reference link 229

duplexify

reference link 229

Duplex stream 201**durable subscriber 619**

history service, designing for chat
application 623, 624
with AMQP and RabbitMQ 623

dynamic horizontal scaling 561**dynamic load balancer**

implementing 563-570

E**early return principle 106****ECMAScript 16**

decorator proposal 312, 313
reference link 16

ECMAScript 2015 (ES2015) 23**ECMAScript modules**

(ES modules) 21, 23, 25, 158

circular dependencies 40, 41
CommonJS modules, importing from 57, 58
default exports 29, 30
default imports 29, 30

dynamic imports 33-35

importing, from CommonJS 59

loading phases 37, 38

mixed exports 30-32

module identifiers 32

named exports 27-29

named imports 27-29

read-only bindings 39

static imports 33-35

syntax 27

using, in Node.js 25, 26

encapsulation 256**end-to-end tests 400, 401**

application structure 468
browser automation 472
challenges 401
user flow 470-472
writing 467

engines section, of package.json

reference link 16

entry point 38**error handling, async/await 159**

return 161, 162
return await 161, 162
try...catch 160, 161

ES modules, and CommonJS

import interoperability 57-59
JSON files, importing 60-62
missing references 56
strict mode 54
this keyword 55
top-level await 55

Event Demultiplexer 12, 13**event demultiplexing 9-11****EventEmitter 84, 85**

using 85, 86
versus callbacks 91, 92

event loop 10, 12, 13

- event message** 600
- event notification interface** 9
- Event Queue** 12, 13
- events** 67
 - callbacks, combining with 92-96
- exchange**
 - direct 622
 - fan-out 622
 - topic 622
- executor function** 140, 153
- Express** 373
 - URL 373
- external processes, for running CPI-bound tasks**
 - child processes 522, 523
 - considerations 529
 - subset sum task, delegating 523

F

- Factory pattern** 254
 - encapsulation 256, 257
 - object creation, decoupling 255, 256
 - object creation, implementing 255, 256
 - simple code profiler, building 257-260
 - use case 260
- fail-fast approach** 84
- Fastify**
 - reference link 457
- fault tolerance** 538
- file modules** 35
- filesystem API**
 - Level, using through 314-317
- fire-and-forget** 620
- first in first out (FIFO)** 508
- fourTheorem**
 - reference link 635
- functional reactive programming (FRP)** 303
- function injection** 286

- function under test (FUT)** 408
- fuzz testing** 402

G

- Gang of Four (GoF) design patterns** 321
- garbage collection**
 - reference link 89
- gem** 21
- generator delegation syntax** 363
- generator functions** 359, 360
 - controlling 361, 362
- generator object** 359
- generators** 359
 - using, in place of iterators 362-364
- Gherkin** 396
- GitHub Container Registry**
 - reference link 580
- god object** 593
- Google Container Registry**
 - reference link 580
- gRPC** 337
- gzipping**
 - with buffered API 177
 - with streams 178

H

- HAProxy** 563
 - reference link 557
- hashsum cracker**
 - implementing, with AMQP 647
 - implementing, with Redis streams 651
- hashsum cracker implementation, with AMQP**
 - application, running 649
 - producer, implementing 647, 648
 - results collector, implementing 649
 - worker, implementing 648, 649

hashsum cracker implementation,
 with Redis streams
 application, running 656
 producer, implementing 652
 results collector, implementing 655, 656
 worker, implementing 652-655

high coupling 585

Hono

 reference link 381

horizontal scaling 541

HTTP client

 peer-to-peer load balancing 572, 573

HTTP requests

 mocking, with built-in test mock 432-435

 mocking, with Undici 435-438

httpxy

 reference link 564

I

idempotency 549

immediately invoked function expression
 (IIFE) 24

immutable objects 272

indexed n-ary tree 640

indexed-string-variation package

 reference link 641

infinite recursive promise resolution chains

 issue 167-170

infrastructure as code (IaC) 561

initialization vector 186

initialized state 495

 implementing 495-498

injector 282, 283

integration 597

integration patterns, microservice
 architecture 588, 589

 API orchestration 590-593

 API proxy 589, 590

integration tests 400

 complexity 400

 web application 457

 with local database 449-456

 writing 449

Intercepting Filter pattern 374

 reference link 374

interleaving

 applying, to subset sum algorithm 520-522

 considerations 522

 with setImmediate 520

Internet of Things (IoT) 17

Inversion of Control pattern 286

inverted testing pyramid 404

I/O (input/output) 7

ioredis package

 reference link 614

I/O starvation 78

is-sorted library

 reference link 4

iterable protocol 348-350

 implementing, on iterators 352, 353

iterables 348

 as native JavaScript interface 350, 351

Iterator pattern 344

 utilities 353-358

iterator protocol 345-347

iterator result 345

iterators

 as native JavaScript interface 350, 351

J

JavaScript, in Node.js 15

 access, to operating system services 16

 module 16

 native code, running 17

 running 15, 16

Jest

reference link 407

jQuery 22**JSON files**

importing 60-62

json-socket module

reference link 312

JSON Web Tokens (JWTs) 555**K****Keep It Simple, Stupid (KISS) principle 5, 100****Knex 260****Koa**

reference link 381

kubect1

reference link 579

Kubernetes 578, 579

reference link 558

rollouts 582

Kubernetes deployment

creating 580, 581

scaling 581

Kubernetes tutorials

reference link 579

L**lazy initialization 299****lazy promises 153-156****lazystream**

reference link 214

Lazy streams 214**lcov**

reference link 422

least recently used (LRU) 508**Level 289, 309**

using, through filesystem API 314-317

LevelDB

reference link 309

Level ecosystem

reference link 309

level-filesystem

reference link 317

LevelGraph

reference link 309

level npm package 502**Level plugin**

implementing 309, 311

level-ttl plugin

reference link 311

libuv 13, 14

reference link 14

Linux container 574**load balancing**

with Nginx 557-561

load distribution 538**Locator API 476, 477**

reference link 477

log 603**long polling 601, 602****long-term support (LTS) release 16****LoopBack**

reference link 304

M**MailHog**

reference link 483

Mastra

reference link 340

Memcached

reference link 508

memory leak 88**message 599**

message broker 602, 604

using, to distribute events in e-commerce application 593, 594

message queues 601

versus stream 631

Message Queue Telemetry Transport (MQTT)

reference link 604

message types

command messages 599
document message 600
event messages 600

messaging pipeline 637**messaging system**

asynchronous communication 602
fundamentals 598
one way, versus request/reply exchange patterns 598, 599
one way, versus request/reply patterns 598
peer-to-peer messaging 604, 605
push, versus pull delivery semantics 601, 602
queues 603
streams 603
synchronous communication 602

microservice architecture 585

advantages 587, 588
example 586
integration patterns 588, 589

microservices 540

challenges 588

microtask queue 137**microtasks 78****middleware framework**

creating, for ZeroMQ 375

middleware manager 374-377

implementing, to process messages 378, 379

Middleware pattern 373-375

in Express 373

Middy

reference link 381

minikube

reference link 579

minimalist real-time chat application

building 606
client side, implementing 607-610
running 610, 611
scaling 610, 611
server side, implementing 606, 607

MobX

reference link 304

Mocha

reference link 407

mock.fn()

spies, creating 430-432

mocking 430**mocking imports**

issues 445, 446
versus dependency injection 446-449

mocks 395**module bundler 16****module identifiers 32****module resolution algorithm 35-37**

file modules 35
Node.js core modules 35
package modules 36

modules 3, 16, 22

using, in TypeScript 62

module specifiers

absolute specifiers 32
bare specifiers 32
deep import specifiers 32
relative specifiers 32

modules wiring 278

with Dependency Injection 282-286
with singleton dependencies 279-282

module system 22

- benefits 22
- in JavaScript 22
- in Node.js 23

MongoDB 498

- reference link 553

Mongoose 498**monit**

- reference link 558

monkey patching 46

- effect of type safety, in TypeScript objects 52-54
- example 46-52

monolithic application 539**monolithic architecture 583**

- challenge 585
- example 584

monolithic kernels 583**multiple checksum generator**

- implementing 234

multiplexer 237**multiplexing 9, 237****multistream**

- reference link 237

mutation testing 402**mux/demux application**

- running 242

N**namespace import 28****nanoid package**

- reference link 659

NestJS

- reference link 312

Nginx 563

- for load balancing 557-561
- reference link 557

nock

- reference link 438

Node.js

- components 14
- unit test 404-407
- working 6

Node.js applications

- scaling 538

Node.js-based proxies 557**Node.js core modules 35**

- mocking 438-441

Node.js philosophy 2

- pragmatism 6
- simplicity 5
- small core 3
- small modules 3, 4
- small surface area 5

Node.js release cycle

- reference link 16

Node.js streams

- converting, to Web Streams 247
- Web Streams, converting to 247, 248

Node.js test runner 407

- running, with TypeScript 423
- test script 408, 409

Nomad

- reference link 558

non-blocking I/O 8, 9**non-CPS callbacks 73****npm 3****O****object augmentation**

- (monkey patching) 295, 296

object composition 292-294**object decoration 306, 307**

- with Proxy 307, 308

object streams

demultiplexing 243

multiplexing 242

Observer pattern 67, 84

asynchronous events 90, 91

errors, propagating 86

EventEmitter 84, 85

EventEmitter, using 85, 86

memory leaks, risks 88, 89

objects, making observable 87, 88

synchronous events 90, 91

one way communication

versus request/reply patterns 598

one-way communication 598**Open Container Initiative (OCI)**

reference link 574

OpenTelemetry 557**optimal asynchronous caching algorithm 502****P****package 21****package modules 36****Packer**

reference link 561

packet switching 238**parallelism 115****parallel pipeline 637****parameterized test cases 413-415**

test suites 415, 416

Passport.js 330, 331

reference link 330

PassThrough stream 210

late piping 211-213

observability 210

path aliases

reference link 63

path traversal attack

reference link 180

PBKDF2

reference link 638

peer-to-peer load balancing 570, 571

in HTTP client 572, 573

peer-to-peer messaging 604, 605**peer-to-peer Publish/Subscribe**

architecture, designing for chat server 615, 616

with ZeroMQ 615

performance testing 402**pg package 498****pino**

reference link 48

pino-colada

reference link 48

pipeline()

error handling 217, 218

pipes

error handling 216

used, for connecting streams 215, 216

piping patterns 229**Playwright 472**

assertions 478

Locator API 476, 477

navigation 476

reference link 472

timeouts 479

user flow, testing 479, 480

Playwright API 475, 476**Playwright project**

setting up 473, 474

p-lazy

reference link 156

p-limit package

reference link 152

pm2

reference link 551

- p-map package**
 - reference link 148
- pnpm 3**
- point to point communication 646**
- point-to-point duplex channel 657**
- polyfills 15, 299**
- portfinder**
 - reference link 564
- PostgreSQL**
 - reference link 553
- PouchDB**
 - reference link 309
- Prisma ORM**
 - reference link 317
- proactor pattern 13**
 - reference link 13
- process identifier (PID) 544**
- processing pipeline 374**
- producer 602**
- product-market fit 540**
- project setup, web application 458-460**
 - business logic 460, 461
 - integration testing 464-467
 - routes code, writing 461-464
- Prometheus 557**
- promise API 138-140**
- promises 67, 134, 505**
 - batching and caching requests, implementing with 507, 508
 - batching requests, implementing with 505-507
 - chain execution flow 136, 137
 - concurrent execution 147
 - creating 140, 141
 - lazy promises 153-156
 - limited concurrent execution 148
 - sequential execution 143-146
 - sequential iteration 143-146
 - TaskQueue class, implementing 148-150
 - web spider, updating 150-152
- Promises/A+ 134, 137**
 - reference link 137
- promisification 141, 142**
- promisify() function 143**
- property injection 286**
- property testing 402**
- proxy 289**
- proxying techniques**
 - comparing 299
- Proxy pattern 289, 290**
 - benefits 290
 - Change Observer pattern 301-303
 - implementing, techniques 290
 - logging Writable stream, creating 300, 301
 - versus Decorator pattern 313
- Proxy pattern implementation techniques 290, 291**
 - built-in Proxy object 296-298
 - object augmentation 295, 296
 - object composition 292-295
- publisher 605**
- Publish/Subscribe pattern 605, 606**
 - minimalist real-time chat application, building 606
- pull delivery (consumer-initiated) model 601**
 - examples 601
- pumpify**
 - reference link 230
- Puppeteer**
 - reference link 472
- push delivery (producer-initiated) model 601**
 - examples 601
 - versus pull delivery (producer-initiated) 601
- push notifications 602**
- PUSH/PULL sockets 638**
- pyramid of doom 104**

Q

queues 603

- auto-delete 622
- durable 622
- exclusive 622
- used, for reliable message delivery 619-621

queuing state 495

R

RabbitMQ broker

- reference link 604, 623

race conditions

- fixing, with concurrent tasks 119-121

React

- URL 32

Reactive Manifesto

- reference link 303

reactive programming (RP) 289, 303

reactor pattern 11-13

Readable stream 187

- async iterators 189
- flowing mode 189
- from iterables 193, 194
- implementing 190-192
- non-flowing mode 187-189
- simplified construction 193

Readable stream, utilities 243

- filtering and iteration 244
- limiting and reducing 244-246
- mapping and transformation 243
- searching and evaluation 244

records 630

Red-Green-Refactor 396

Redis

- reference link 508
- using, as simple message broker 611-614

Redis protocol (RESP) 337

Redis streams 631

- chat application, implementing with 631-635
- hashsum cracker, implementing with 651
- used, for distributing tasks 650

regression testing 402

regular functions

- versus arrow functions 308, 309

reliability 538

reliable message delivery

- with queues 619-621

remote logger

- building 238
- demultiplexing 240-242
- multiplexing 238-240

Remote Procedure Call (RPC) 382, 600

request/reply abstraction implementation, with correlation

- full request/reply cycle, trying 660-662
- reply, abstracting 659, 660
- request, abstracting 658, 659

request/reply pattern 656

- Correlation Identifier 656, 657

request/reply patterns

- reference link 668
- versus one way communication 598

RequireJS

- URL 22

resiliency 547

results queue 647

Return Address pattern implementation, with AMQP 662, 663

- replier, implementing 666, 667
- reply abstraction, implementing 665, 666
- request abstraction, implementing 663-665
- requestor, implementing 666

Return on Investment (ROI) 404

Revealing Constructor pattern 271
immutable buffer, building 272-274
use cases 274

revealing module pattern 23-25

reverse proxy 555
application, scaling with 556, 557

reverse proxy, versus forward proxy
reference link 556

round-robin algorithm 543

row-level locking 461

runit
reference link 558

S

Saga 549

scalability 537, 538
dimensions 538-540

scale cube 539
strategies, combining across 540

sCrypt
reference link 638

search-index plugin
reference link 311

security testing 402

Selenium
reference link 472

semicoroutines 359

sequential execution flow 109
known set of tasks, executing 109, 110

sequential iteration 110
crawling of links 112, 113
pattern 114, 115
web spider version 2 110, 111

serve-handler
reference link 607

server-sent events
reference link 598

service-level agreements (SLAs) 550

service locator 286

service registry 562
using 563

Simple/Streaming Text Orientated Messaging Protocol (STOMP)
reference link 604

singleton dependencies 279
used, for wiring modules 279-281

Singleton pattern 275-277

sink 637

Socket.IO
reference link 555

software development philosophies
reference link 3

software testing 392

spies 395
creating, with `mock.fn()` 430-432

state 331

stateful communications
handling 552
state, sharing across multiple instances 553
sticky load balancing 554, 555

State pattern 331, 332
applying 495-498
basic failsafe socket, implementing 332-340

state transition 332

sticky load balancing 554, 555

strategies 322

Strategy pattern 322-325
multi-format configuration objects 325-330

Stream consumer
utilities 249, 250

streaming
versus buffering 175, 176

streaming platform
characteristics 630

streams 67, 175, 186, 603, 630

- anatomy 187
- combining 229, 230
- composability 183
- connecting, with pipes 215, 216
- Duplex stream 201
- for gzipping 178
- forking 233
- Lazy stream 214
- merging 235
- PassThrough stream 210
- Readable stream 187
- reading from 187
- spatial efficiency 176
- time efficiency 178-183
- Transform stream 202
- used, for reliable messaging 629
- versus message queues 631
- Writable stream 195
- writing to 195, 196

structural design patterns

- Adapter 289, 314
- Decorator 289, 304
- Proxy 289, 290

stubs 394**subject 289****subscribers 605****subset sum problem 516**

- solving 516-520

subset sum task delegation

- child process, communicating with 526, 527
- process pool, implementing 524-526
- to external process 523
- worker, implementing 528

subtests 411, 412

- concurrency 412, 413

supervisor

- reference link 558

supply chain vulnerabilities

- reference link 5

surrogate 289**synchronous APIs**

- using 76

synchronous communication 602**synchronous CPS 69, 70****synchronous event demultiplexer 9****synchronous programming 67****syntactic sugar**

- reference link 139

systemd

- reference link 558

System Under Test (SUT) 393**T****tail variable 204****task distribution patterns 635-637****Task pattern 383****TaskQueue class**

- implementing, with promises 148-150

tasks queue 647**tee**

- reference link 233

template methods 341**Template pattern 341**

- configuration manager template 342-344

ternary-stream package

- reference link 243

Terraform

- reference link 561

Test Anything Protocol (TAP)

- reference link 420

testcontainers

- reference link 457

test doubles 394

- mocks 395
- spies 395
- stubs 394

Test-Driven Development (TDD) 392, 395, 396**testing 392**

- benefits 393

testing framework 407

- features 407

testing pyramid 403, 404**testing trophy**

- reference link 404

test reporters 419

- dot 420
- junit 420
- lcov 420
- spec 420
- tap 420

test runner, tips and tricks

- targeted test execution, with custom glob patterns 417
- watch mode 416

tests 398

- end-to-end tests 400-402
- fuzz testing 402
- integration tests 400
- mutation testing 402
- organizing 409-411
- performance testing 402
- property testing 402
- regression testing 402
- security testing 402
- unit tests 399
- usability testing 402

test suites 415, 416**text files**

- merging 235-237

thenables 138**tight coupling 445****timeouts 479****top-level await 158, 159****tracked-queue**

- reference link 340

transcompilation 299**Transform streams 202**

- data, aggregating 206-209
- data, filtering 206-209
- implementing 203-205
- simplified construction 205

transpilers 15**trap methods 297****tree traversal algorithm**

- reference link 345

TypeScript 18

- input module syntax 64
- module output format, configuring 64
- module resolution 64
- modules, using 62
- Node.js test runner, using 423
- output emission 64
- using, with Node.js 18, 19

TypeScript compiler 62, 63**@types/node package 19****U****Undici**

- HTTP requests mocking 435-438

unit 21**unit tests 399**

- qualities 399
- with Node.js 404-407
- writing 423, 424

Universally Unique Lexicographically Sortable Identifier (ULID)

- reference link 625

universal module definition (UMD)

URL 22

Unlimited Concurrent Execution pattern 119**unordered concurrent stream**

implementing 221-223

unpredictable function

writing 73, 74

URL status monitoring application

implementing 223-225

usability testing 402**user flow, E2E tests 470, 472****user flow, Playwright**

considerations 482, 483

dashboard, checking 482

event, booking 481

event booking, verifying 481, 482

registration form, filling 481

Sign Up 480

starting 480

testing 479, 480

V**V8 14****Valkey**

reference link 508

ventilator 636**vertical scaling 541****virtual machines (VMs) 17****Vitest**

reference link 407

Vue.js

reference link 304

W**watch mode 416**

default test discovery 417

subset, executing of tests 416

Watt

reference link 557

web application

project setup 458-460

testing 457

WebApp service 563**WebAssembly (Wasm) 17**

reference link 17

Webpack

URL 23

web scraper 100**web spider**

creating 100-103

Web Streams 246

converting, to Node.js streams 247, 248

Node.js streams, converting to 247

WHATWG Streams Standard 246

reference link 246

white-box tests 401**worker 636****worker threads 530**

subset sum task, running 531-533

wrapper functions 511**Writable stream 195**

implementing 199, 200

simplified construction 201

ws package

reference link 606

X**xadd command**

reference link 634

xdell command

reference link 635

xrange command

reference link 634

xread command

reference link 634

XState 340

reference link 340

xtrim command

reference link 635

Y

yarn 3

Z

Zalgo 74, 75

zero-downtime restarts 550

ZeroMQ 615

middleware framework, creating 375

PUB/SUB sockets, using 616-619

reference link 340

using, with peer-to-peer Publish/Subscribe 615

ZeroMQ fan-out/fan-in pattern 637

distributed hashsum cracker, building with
ZeroMQ 638-640

PUSH/PULL sockets 638

ZeroMQ middleware framework

client 380, 381

server 379, 380

using 379

zeromq package

reference link 372

zod

reference link 328