

EXPERT INSIGHT

Node.js Design Patterns

Design and implement production-grade Node.js applications using proven patterns and techniques

Third Edition



**Mario Casciaro
Luciano Mammino**

Packt>

Node.js Design Patterns

Third Edition

Design and implement production-grade Node.js applications using proven patterns and techniques

Mario Casciaro

Luciano Mammino



BIRMINGHAM - MUMBAI

Node.js Design Patterns

Third Edition

Copyright © 2020 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.

Producer: Tushar Gupta

Acquisition Editor – Peer Reviews: Suresh Jain

Project Editor: Tom Jacob

Content Development Editors: Joanne Lovell, Bhavesh Amin

Copy Editor: Safis Editing

Technical Editor: Saby D'silva

Proofreader: Safis Editing

Indexer: Manju Arasan

Presentation Designer: Sandip Tadge

First published: December 2014

Second Edition: July 2016

Third Edition: July 2020

Production reference: 1240720

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-83921-411-0

www.packt.com



packt.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
- Learn better 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 www.Packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.Packt.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.

Contributors

About the authors

Mario Casciaro is a software engineer and entrepreneur. Since he was a child he's been in love with building things, from LEGO spaceships to programs written on his Commodore 64, his first computer. When in college, he used to work more on side projects than on assignments and he published his first open source project on SourceForge back in 2006, it was around 30,000 lines of C++ code. After graduating with a master's degree in software engineering, Mario worked at IBM for a number of years, first in Rome, then in the Dublin Software Lab. He currently splits his time between Var7 Technologies — his own software company — and his role as lead engineer at D4H Technologies where he creates software for emergency response teams. He is a big supporter of pragmatism and simplicity.

The story of this book starts with you all who are reading this book. You make all our efforts worthwhile. Thanks also to the readers who contributed to the success of the first two editions, providing invaluable feedback, writing reviews, and spreading the word about the book.

Thanks to the Packt team, who worked hard to make this book a reality; thanks to Tom Jacob, Jonathan Malysiak, Saby D'silva, Bhavesh Amin, Tushar Gupta, Kishor Rit, Joanne Lovell.

For this book, I had the honor to work with a team of top-class technical reviewers: Roberto Gambuzzi, Minwoo Jung, Kyriakos Markakis, Romina Miraballes, Peter Poliwoda, Liran Tal, and Tomas Della Vedova. Thanks for lending your expertise to make this book perfect.

Thanks to Hiroyuki Musha who translated the second edition of Node.js Design Patterns to Japanese and improved it in the process.

However, the top of the podium goes to Luciano. It has been a tremendous experience and also an honor coauthoring this book with him. Thank you Luciano for being both a great professional and an all-around great person. Hopefully, we'll have the chance to work again together in the future.

Thanks to my Dad, Mom, Alessandro, and Elena for being there for me even if we are far away.

Finally, I'd like to say a heartfelt thank you to Miriam, the love of my life, for inspiring and supporting me in all my endeavors. There are still many more adventures awaiting us. And to Leonardo, thanks for filling our life with joy. Your dad loves you very much.

Luciano Mammino was born in 1987, the same year *Super Mario Bros.* was released in Europe, which, by chance, is his favorite game! He started coding at the age of 12, hacking away with his father's old i386 armed only with MS-DOS and the QBasic interpreter. Since then he has been a professional software developer for more than 10 years. Luciano is currently wearing the hat of principal software engineer at FabFitFun in Dublin where he builds microservices and scaling applications for millions of users.

Luciano loves the cloud, full-stack web development, Node.js, and serverless. Among other things, Luciano runs Fullstack Bulletin (fstack.link), a weekly newsletter for ambitious full-stack developers and Serverlesslab.com, bespoke training courses to foster serverless adoption.

The biggest thanks of all goes to Mario Casciaro for involving me in such an amazing project. It has been a fantastic journey and I have definitely learned and grown a ton while working together. I do hope there will be many other chances to work together!

This book was possible, thanks to the hard work of the team at Packt, especially Saby, Tushar, Tom, Joanne, Kishor, Jonathan, and Bhavesh. Thanks for bearing with us for almost a year, and thanks to everyone else at Packt for supporting 3 editions of this book!

A huge thanks go to our talented reviewers. Without their true-hearted supervision and their invaluable recommendations, this book wouldn't be something I could be proud about: thanks Romina, Kyriakos, Roberto, Peter, Tomas, Liran and Minwoo. I will be forever grateful for your help.

A special thanks to Pdraig O'Brien, Domagoj Katavic, Michael Twomey, Eugen Serbanescu, Stefano Abalsamo, and Gianluca Arbezzano for providing a great deal of support along the way, and for letting me borrow their expertise when I needed some extra piece of feedback on the content of this book.

My gratitude goes to my family, who raised me and supported me in every possible way along my journey. Thanks, Mom, for being a constant source of inspiration and strength in my life. Thanks, Dad, for all the lessons, the encouragement, and the pieces of advice. I do miss them in my life. Thanks to Davide and Alessia for being present in all the painful and the joyful events of life.

Thanks to Franco, Silvana, and their family for supporting many of my initiatives and for sharing their wisdom with me.

Kudos to all the readers of the second edition, especially, the ones that went a step further and left reviews, reported issues, submitted patches, or suggested new topics. Special praise goes to Vu Nhat Tan, Danilo Carrabino, Aaron Hatchard, Angelo Gulina, Bassem Ghoniem, Vasyl Boroviak, and Willie Maddox. Thanks also to Hiroyuki Musha for translating this book to Japanese and for finding many opportunities to improve the content of the book. You are my hero!

Thanks to my friends for promoting this book and supporting me: Andrea Mangano, Ersel Aker, Joe Karlsson, Francesco Sciuti, Arthur Thevenet, Anton Whalley, Duncan Healy, Heitor Lessa, Francesco Ciula, John Brett, Alessio Biancalana, Tommaso Allevi, Chris Sevilleja, David Gonzalez, Nicola del Gobbo, Davide De Guz, Aris Markogiannakis, and Simone Gentili.

Last, but not least, thanks to my partner Francesca. Thanks for the unconditioned love and for supporting me on every adventure, even the craziest ones. I look forward to writing the next chapters of our book of life together!

About the reviewers

Roberto Gambuzzi was born in 1978 and started coding at the age of 8 on a Commodore 16. His first PC was an 8086 with 1 MB of RAM and a 20 MB hard drive. He coded in BASIC, assembly, Pascal, C, C++, COBOL, Delphi, Java, PHP, Python, Go, JavaScript, and other lesser-known languages. He worked with Amazon in Dublin, then moved to the world of startups. He likes simple, effective code.

He was the reviewer of *Magento Best Practices* (<https://leanpub.com/magebp>).

Thanks to Luciano Mammino and Mario Casciaro for giving me the chance to review their beautiful book.

Minwoo Jung is a Node.js core collaborator and works for NodeSource as a full-time software engineer. He specializes in web technologies with more than 10 years of experience and used to publish the weekly updates on the official Node.js website. When he isn't glued to a computer screen, he spends time hiking with his friends.

Kyriakos Markakis has worked as a software engineer and application architect. He acquired experience in developing web applications by working on a wide range of projects in e-government, online voting, digital education, media monitoring, digital advertising, travel, and banking industries for almost 10 years.

During that time, he had the opportunity to work with some leading technologies, such as Java, asynchronous messaging, AJAX, REST APIs, PostgreSQL, Redis, Docker, and others. He has dedicatedly worked on Node.js for the past 3 years.

He contributes his knowledge by mentoring new technologies to beginners through e-learning programs and by reviewing books.

Romina Miraballes is a software engineer from Uruguay, currently living in Ireland. She started working as a firmware developer programming in C for a company that creates medical devices. After that, she began working as a full-stack developer, working on several projects for a wide range of startups, creating cloud solutions in AWS with Node.js. Currently, she works at Vectra AI, which is a cybersecurity company and mostly works with Node.js applications in AWS and Azure.

I would like to thank, first of all, Luciano for giving me the opportunity to be part of this project as a reviewer. Secondly, to my family and my boyfriend Nicolás who are always supporting me.

Peter Poliwoda is a senior software engineer and a technical lead at the IBM Technology Campus in Ireland. He has over a decade of software development experience in a wide range of industry sectors from banking and financial systems to healthcare and research. Peter graduated from University College Cork in business information systems. He has a keen interest in artificial intelligence, cognitive solutions, as well as IoT. He is an advocate of solving problems using technology. Peter is an avid contributor to the tech community through open source and by speaking at conferences, seminars, and workshops across Europe. He enjoys giving back to the educational system by working together with schools and universities.

To Larissa – you inspire me to do great things.

Liran Tal is a developer advocate at Snyk and a member of the Node.js Security working group. Among his security activities, Liran has also authored *Essential Node.js Security* and coauthored O'Reilly's *Serverless Security* book, and is a core contributor to the OWASP NodeGoat project. He is passionate about the open source movement, web technologies, and testing and software philosophy.

Tomas Della Vedova is an enthusiastic software engineer, who spends most of his time programming in JavaScript and Node.js. He works for Elastic as a senior software engineer in the clients team, focusing on the JavaScript client. Tomas is also the author of the Fastify web framework and part of its ecosystem. He constantly advances the enrichment of his knowledge and the exploration of new technologies; moreover, he is a strong open source supporter and he will always be passionate about technology, design, and music.

Table of Contents

Preface	xi
Chapter 1: The Node.js Platform	1
The Node.js philosophy	2
Small core	2
Small modules	2
Small surface area	3
Simplicity and pragmatism	4
How Node.js works	5
I/O is slow	5
Blocking I/O	5
Non-blocking I/O	6
Event demultiplexing	7
The reactor pattern	9
Libuv, the I/O engine of Node.js	11
The recipe for Node.js	12
JavaScript in Node.js	13
Run the latest JavaScript with confidence	13
The module system	14
Full access to operating system services	14
Running native code	15
Summary	16
Chapter 2: The Module System	17
The need for modules	18
Module systems in JavaScript and Node.js	19
The module system and its patterns	20
The revealing module pattern	20

CommonJS modules	22
A homemade module loader	22
Defining a module	24
module.exports versus exports	25
The require function is synchronous	26
The resolving algorithm	26
The module cache	28
Circular dependencies	29
Module definition patterns	33
Named exports	33
Exporting a function	34
Exporting a class	35
Exporting an instance	36
Modifying other modules or the global scope	37
ESM: ECMAScript modules	38
Using ESM in Node.js	39
Named exports and imports	39
Default exports and imports	42
Mixed exports	43
Module identifiers	45
Async imports	45
Module loading in depth	48
Loading phases	48
Read-only live bindings	49
Circular dependency resolution	50
Modifying other modules	56
ESM and CommonJS differences and interoperability	60
ESM runs in strict mode	60
Missing references in ESM	60
Interoperability	61
Summary	62
Chapter 3: Callbacks and Events	63
The Callback pattern	64
The continuation-passing style	64
Synchronous CPS	65
Asynchronous CPS	65
Non-CPS callbacks	67
Synchronous or asynchronous?	67
An unpredictable function	68
Unleashing Zalgo	68
Using synchronous APIs	70
Guaranteeing asynchronicity with deferred execution	72

Node.js callback conventions	73
The callback comes last	73
Any error always comes first	74
Propagating errors	74
Uncaught exceptions	75
The Observer pattern	77
The EventEmitter	78
Creating and using the EventEmitter	79
Propagating errors	80
Making any object observable	80
EventEmitter and memory leaks	82
Synchronous and asynchronous events	83
EventEmitter versus callbacks	85
Combining callbacks and events	86
Summary	88
Exercises	88
Chapter 4: Asynchronous Control Flow Patterns with Callbacks	89
The difficulties of asynchronous programming	90
Creating a simple web spider	90
Callback hell	93
Callback best practices and control flow patterns	94
Callback discipline	95
Applying the callback discipline	95
Sequential execution	98
Executing a known set of tasks in sequence	99
Sequential iteration	100
Parallel execution	104
Web spider version 3	106
The pattern	108
Fixing race conditions with concurrent tasks	108
Limited parallel execution	110
Limiting concurrency	112
Globally limiting concurrency	113
The async library	119
Summary	120
Exercises	121
Chapter 5: Asynchronous Control Flow Patterns with Promises and Async/Await	123
Promises	124
What is a promise?	125
Promises/A+ and thenables	127
The promise API	128

Creating a promise	130
Promisification	131
Sequential execution and iteration	133
Parallel execution	136
Limited parallel execution	137
Implementing the TaskQueue class with promises	138
Updating the web spider	139
Async/await	141
Async functions and the await expression	141
Error handling with async/await	143
A unified try...catch experience	143
The "return" versus "return await" trap	144
Sequential execution and iteration	145
Antipattern – using async/await with Array.forEach for serial execution	147
Parallel execution	147
Limited parallel execution	149
The problem with infinite recursive promise resolution chains	152
Summary	156
Exercises	157
Chapter 6: Coding with Streams	159
Discovering the importance of streams	160
Buffering versus streaming	160
Spatial efficiency	161
Gzipping using a buffered API	162
Gzipping using streams	163
Time efficiency	163
Composability	167
Adding client-side encryption	167
Adding server-side decryption	169
Getting started with streams	170
Anatomy of streams	170
Readable streams	171
Reading from a stream	171
Implementing Readable streams	174
Writable streams	179
Writing to a stream	179
Backpressure	181
Implementing Writable streams	182
Duplex streams	185
Transform streams	185
Implementing Transform streams	186
Filtering and aggregating data with Transform streams	189
PassThrough streams	193
Observability	193

Late piping	194
Lazy streams	197
Connecting streams using pipes	198
Pipes and error handling	200
Better error handling with pipeline()	201
Asynchronous control flow patterns with streams	203
Sequential execution	203
Unordered parallel execution	206
Implementing an unordered parallel stream	206
Implementing a URL status monitoring application	208
Unordered limited parallel execution	210
Ordered parallel execution	212
Piping patterns	214
Combining streams	214
Implementing a combined stream	217
Forking streams	219
Implementing a multiple checksum generator	220
Merging streams	221
Merging text files	221
Multiplexing and demultiplexing	223
Building a remote logger	224
Multiplexing and demultiplexing object streams	229
Summary	230
Exercises	230
Chapter 7: Creational Design Patterns	233
Factory	234
Decoupling object creation and implementation	235
A mechanism to enforce encapsulation	236
Building a simple code profiler	238
In the wild	241
Builder	241
Implementing a URL object builder	244
In the wild	248
Revealing Constructor	249
Building an immutable buffer	250
In the wild	253
Singleton	253
Wiring modules	257
Singleton dependencies	258
Dependency Injection	261
Summary	266
Exercises	267

Chapter 8: Structural Design Patterns	269
Proxy	269
Techniques for implementing proxies	271
Object composition	272
Object augmentation	275
The built-in Proxy object	277
A comparison of the different proxying techniques	280
Creating a logging Writable stream	281
Change observer with Proxy	282
In the wild	285
Decorator	285
Techniques for implementing decorators	286
Composition	286
Object augmentation	288
Decorating with the Proxy object	289
Decorating a LevelUP database	290
Introducing LevelUP and LevelDB	290
Implementing a LevelUP plugin	291
In the wild	293
The line between proxy and decorator	294
Adapter	294
Using LevelUP through the filesystem API	295
In the wild	298
Summary	299
Exercises	300
Chapter 9: Behavioral Design Patterns	301
Strategy	302
Multi-format configuration objects	304
In the wild	308
State	308
Implementing a basic failsafe socket	310
Template	315
A configuration manager template	316
In the wild	318
Iterator	319
The iterator protocol	319
The iterable protocol	322
Iterators and iterables as a native JavaScript interface	324
Generators	326
Generators in theory	327
A simple generator function	327
Controlling a generator iterator	328
How to use generators in place of iterators	330

Async iterators	331
Async generators	334
Async iterators and Node.js streams	335
In the wild	336
Middleware	337
Middleware in Express	337
Middleware as a pattern	338
Creating a middleware framework for ZeroMQ	340
The Middleware Manager	340
Implementing the middleware to process messages	342
Using the ZeroMQ middleware framework	344
In the wild	347
Command	347
The Task pattern	349
A more complex command	349
Summary	353
Exercises	354
Chapter 10: Universal JavaScript for Web Applications	357
Sharing code with the browser	358
JavaScript modules in a cross-platform context	359
Module bundlers	360
How a module bundler works	363
Using webpack	369
Fundamentals of cross-platform development	371
Runtime code branching	372
Challenges of runtime code branching	373
Build-time code branching	374
Module swapping	377
Design patterns for cross-platform development	378
A brief introduction to React	379
Hello React	381
Alternatives to react.createElement	383
Stateful components	385
Creating a Universal JavaScript app	391
Frontend-only app	392
Server-side rendering	399
Asynchronous data retrieval	405
Universal data retrieval	411
Two-pass rendering	412
Async pages	414
Implementing async pages	416
Summary	425
Exercises	426

Chapter 11: Advanced Recipes	427
Dealing with asynchronously initialized components	428
The issue with asynchronously initialized components	428
Local initialization check	429
Delayed startup	430
Pre-initialization queues	431
In the wild	435
Asynchronous request batching and caching	435
What's asynchronous request batching?	436
Optimal asynchronous request caching	437
An API server without caching or batching	439
Batching and caching with promises	441
Batching requests in the total sales web server	442
Caching requests in the total sales web server	443
Notes about implementing caching mechanisms	445
Canceling asynchronous operations	445
A basic recipe for creating cancelable functions	446
Wrapping asynchronous invocations	447
Cancelable async functions with generators	449
Running CPU-bound tasks	453
Solving the subset sum problem	453
Interleaving with setImmediate	457
Interleaving the steps of the subset sum algorithm	457
Considerations on the interleaving approach	459
Using external processes	460
Delegating the subset sum task to an external process	461
Considerations for the multi-process approach	467
Using worker threads	468
Running the subset sum task in a worker thread	469
Running CPU-bound tasks in production	472
Summary	473
Exercises	473
Chapter 12: Scalability and Architectural Patterns	475
An introduction to application scaling	476
Scaling Node.js applications	477
The three dimensions of scalability	477
Cloning and load balancing	479
The cluster module	480
Notes on the behavior of the cluster module	481
Building a simple HTTP server	482
Scaling with the cluster module	484
Resiliency and availability with the cluster module	486
Zero-downtime restart	488

Dealing with stateful communications	490
Sharing the state across multiple instances	491
Sticky load balancing	492
Scaling with a reverse proxy	494
Load balancing with Nginx	496
Dynamic horizontal scaling	501
Using a service registry	501
Implementing a dynamic load balancer with http-proxy and Consul	503
Peer-to-peer load balancing	510
Implementing an HTTP client that can balance requests across multiple servers	511
Scaling applications using containers	513
What is a container?	513
Creating and running a container with Docker	514
What is Kubernetes?	517
Deploying and scaling an application on Kubernetes	519
Decomposing complex applications	523
Monolithic architecture	524
The microservice architecture	526
An example of a microservice architecture	526
Microservices – advantages and disadvantages	528
Integration patterns in a microservice architecture	530
The API proxy	531
API orchestration	532
Integration with a message broker	536
Summary	538
Exercises	539
Chapter 13: Messaging and Integration Patterns	541
Fundamentals of a messaging system	542
One way versus request/reply patterns	542
Message types	544
Command Messages	544
Event Messages	545
Document Messages	545
Asynchronous messaging, queues, and streams	545
Peer-to-peer or broker-based messaging	547
Publish/Subscribe pattern	549
Building a minimalist real-time chat application	550
Implementing the server side	550
Implementing the client side	551
Running and scaling the chat application	553
Using Redis as a simple message broker	554
Peer-to-peer Publish/Subscribe with ZeroMQ	557
Introducing ZeroMQ	557
Designing a peer-to-peer architecture for the chat server	558
Using the ZeroMQ PUB/SUB sockets	559

Reliable message delivery with queues	562
Introducing AMQP	564
Durable subscribers with AMQP and RabbitMQ	566
Reliable messaging with streams	571
Characteristics of a streaming platform	571
Streams versus message queues	573
Implementing the chat application using Redis Streams	573
Task distribution patterns	577
The ZeroMQ Fanout/Fanin pattern	579
PUSH/PULL sockets	580
Building a distributed hashsum cracker with ZeroMQ	580
Pipelines and competing consumers in AMQP	587
Point-to-point communications and competing consumers	588
Implementing the hashsum cracker using AMQP	588
Distributing tasks with Redis Streams	592
Redis consumer groups	593
Implementing the hashsum cracker using Redis Streams	594
Request/Reply patterns	598
Correlation Identifier	598
Implementing a request/reply abstraction using correlation identifiers	599
Return address	605
Implementing the Return Address pattern in AMQP	605
Summary	611
Exercises	612
Other Books You May Enjoy	615
Index	619

Preface

Node.js is considered by many a game-changer — possibly the biggest innovation of the decade in web development. It is loved not just for its technical capabilities, but also for the paradigm shift that it introduced in web development and, in general, in the software development ecosystem.

First, Node.js applications are written in JavaScript, the most adopted language on the web and the only programming language supported natively by every web browser. This aspect enables scenarios such as single-language application stacks and the sharing of code between the server and the client. A single language also helps to reduce the gap between frontend and backend engineers, making backend programming extremely approachable and intuitive for frontend developers. Once you are acquainted with Node.js and JavaScript, you can easily build software for a wide variety of platforms and contexts.

Node.js itself is contributing to the rise and evolution of the JavaScript language. People realize that using JavaScript on the server brings a lot of value, and they are loving it for its pragmatism, for its flexibility, its event-driven approach, and for its hybrid nature, halfway between object-oriented and functional programming.

The second revolutionizing factor is Node.js' single-threaded programming model and its asynchronous architecture. Besides obvious advantages from a performance and scalability point of view, this characteristic changed the way developers approach concurrency and parallelism. Mutexes are replaced by queues, threads by callbacks, and synchronization by causality. These abstractions are generally simpler to adopt than their traditional counterparts, but they are still extremely powerful, allowing developers to be very productive while solving day-to-day challenges.

The last and most important aspect of Node.js lies in its ecosystem: the npm package manager, its constantly growing database of modules, its enthusiastic and helpful community, and most importantly, its very own culture based on simplicity, pragmatism, and extreme modularity.

However, because of these peculiarities, Node.js development gives you a very different feel compared to other server-side platforms, and any developer new to this paradigm will often feel unsure about how to tackle even the most common design and coding problems effectively. Common questions include: *How do I organize my code? What's the best way to design this? How can I make my application more modular? How do I handle a set of asynchronous calls effectively? How can I make sure that my application will not collapse while it grows? Or more simply, what's the right way to implement this?* Fortunately, Node.js has become a mature enough platform and most of these questions can now be easily answered with a design pattern, a proven coding technique, or a recommended practice. The aim of this book is to guide you through this emerging world of patterns, techniques, and practices, showing you what the proven solutions to the most common problems are and teaching you how to use them as the starting point to building the solution to your particular problem.

By reading this book, you will learn the following:

- **The "Node way":**
How to use the right point of view when approaching Node.js development. You will learn, for example, how different traditional design patterns look in Node.js, or how to design modules that do only one thing.
- **A set of patterns to solve common Node.js design and coding problems:**
You will be presented with a "Swiss Army knife" of patterns, ready to use in order to efficiently solve your everyday development and design problems.
- **How to write scalable and efficient Node.js applications:**
You will gain an understanding of the basic building blocks and principles of writing large and well-organized Node.js applications that can scale. You will be able to apply these principles to novel problems that don't fall within the scope of existing patterns.
- **Code in "modern JavaScript":**
JavaScript has been around since 1995, but a lot has changed since its first inception, especially in these last few years. This book will take advantage of the most modern JavaScript features, like the class syntax, promises, generator functions, and `async/await`, giving you a properly up-to-date experience.

Throughout the book, you will be presented with real-life libraries and technologies, such as LevelDB, Redis, RabbitMQ, ZeroMQ, Express, and many others. They will be used to demonstrate a pattern or technique, and besides making the example more useful, these will also give you great exposure to the Node.js ecosystem and its set of solutions.

Whether you use or plan to use Node.js for your work, your side project, or for an open source project, recognizing and using well-known patterns and techniques will allow you to use a common language when sharing your code and design, and on top of that, it will help you get a better understanding of the future of Node.js and how to make your own contributions a part of it.

What you need for this book

To experiment with the code, you will need a working installation of Node.js version 14 (or greater) and npm version 6 (or greater). 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 line, know how to install an npm 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.

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 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, serves as an introduction to the world of Node.js application design by showing the patterns at the core of the platform itself. It covers the Node.js ecosystem and its philosophy, and provides a quick introduction to the Node.js internals and the reactor pattern.

Chapter 2, The Module System, dives into the module systems available in Node.js, underlining the differences between CommonJS and the more modern ES modules from the ECMAScript 2015 specification.

Chapter 3, Callbacks and Events, introduces the first steps towards learning asynchronous coding and its patterns, discussing and comparing callbacks and the event emitter (observer pattern).

Chapter 4, Asynchronous Control Flow Patterns with Callbacks, introduces a set of patterns and techniques for efficiently handling asynchronous control flow in Node.js using callbacks. This chapter teaches you some traditional ways to mitigate the "callback hell" problem using plain JavaScript.

Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await, progresses with the exploration of more sophisticated and modern asynchronous control flow techniques.

Chapter 6, Coding with Streams, dives deep into one of the most important tools in Node.js: streams. It shows you how to process data with transform streams and how to combine them into different patterns.

Chapter 7, Creational Design Patterns, starts to explore the traditional design patterns in Node.js. In this chapter, you will learn about some of the most popular creational design patterns, namely the *Factory* pattern, the *Revealing Constructor* pattern, the *Builder* pattern, and the *Singleton* pattern.

Chapter 8, Structural Design Patterns, continues the exploration of traditional design patterns in Node.js, covering structural design patterns such as *Proxy*, *Decorator*, and *Adapter*.

Chapter 9, Behavioral Design Patterns, concludes the conversation around traditional design patterns in Node.js by introducing behavioral design patterns like *Strategy*, *State*, *Template*, *Middleware*, *Command*, and *Iterator*.

Chapter 10, Universal JavaScript for Web Applications, explores one of the most interesting capabilities of modern JavaScript web applications: being able to share code between the frontend and the backend. Throughout this chapter, you will learn the basic principles of Universal JavaScript by building a simple web application using modern tools and libraries.

Chapter 11, Advanced Recipes, takes a problem-solution approach to show you how some common coding and design intricacies can be approached with ready-to-use solutions.

Chapter 12, Scalability and Architectural Patterns, teaches you the basic techniques and patterns for scaling a Node.js application.

Chapter 13, Messaging and Integration Patterns, presents the most important messaging patterns, teaching you how to build and integrate complex distributed systems using Node.js and its ecosystem.

To get the most out of this book

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

Download the example code files

You can download the example code files for this book from your account at www.packt.com/. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at <http://www.packt.com>.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for macOS
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at [nodejsdp.link/repo](https://github.com/nodejsdp/link/repo). In case there's an update to the code, it will be updated on 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://static.packt-cdn.com/downloads/9781839214110_ColorImages.pdf.

Conventions used

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning:

- **Code words in text:** `server.listen(handle)`
- **Pathname:** `src/app.js`
- **Dummy URL:** `http://localhost:8080`

A block of code is generally formatted using StandardJS conventions (nodejsdp.link/standard) and it 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 highlighted 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
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "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 coming through 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, mention the book title in the subject of your message and email us at customer@packtpub.com.

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 could report this to us. Please visit www.packtpub.com/support/errata, select your book, click on the **Errata Submission Form** link, and enter the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you could 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 authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

The Node.js Platform

Some principles and design patterns literally define the developer experience with the Node.js platform and its ecosystem. The most peculiar one is probably its asynchronous nature, which makes heavy use of asynchronous constructs such as callbacks and promises. In this introductory chapter, we will explore where Node.js gets its asynchronous behavior from. This is not just good-to-know theoretical information: knowing how Node.js works at its core will give you a strong foundation for understanding the reasoning behind more complex topics and patterns that we will cover later in the book.

Another important aspect that characterizes Node.js is its philosophy. Approaching Node.js is, in fact, far more than simply learning a new technology: it's also embracing a culture and a community. You will see how this greatly influences the way we design our applications and components, and the way they interact with those created by the community.

In this chapter, you will learn about the following:

- 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

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.

None of these rules are imposed and they should always be applied with common sense; however, they can prove to be tremendously useful when we are looking for a source of inspiration while designing our software.



You can find an extensive list of software development philosophies on Wikipedia at nodejsdp.link/dev-philosophies.

Small core

The Node.js core — understood as the Node.js runtime and built-in modules — has its foundations built on a few principles. One of these is having the smallest possible set of functionalities, while leaving the rest to the so-called **userland** (or **userspace**), which is the ecosystem of modules living outside the core. This principle has an enormous impact on the Node.js culture, as it gives freedom to the community to experiment and iterate quickly on a broader set of solutions within the scope of the userland modules, instead of having one slowly evolving solution that is built into the more tightly controlled and stable core. Keeping the core set of functionalities to the bare minimum, then, is not only convenient in terms of maintainability, but also in terms of the positive cultural impact that it brings to the evolution of the entire ecosystem.

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, 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** 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. 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 couple of lines of code – for example, a regular expression for matching emails such as `nodejsdp.link/email-regex`.

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

- Easier to understand and use
- Simpler to test and maintain
- Small in size and perfect for use in the browser

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.

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 the fact that they are created to be used, rather than extended. Locking down the internals of a module by forbidding any possibility of an extension might sound inflexible, but it actually has the advantage of reducing use cases, simplifying implementation, facilitating maintenance, and increasing usability. In practice, this means preferring to expose functions instead of classes, and being careful not to expose any internals 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, and, finally, 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, the adoption of this principle is also facilitated by JavaScript, which is a very pragmatic language. In fact, it's common to see simple classes, functions, and closures replacing complex class hierarchies. Pure object-oriented designs often try to replicate the real world using the mathematical terms of a computer system without considering the imperfection and complexity of the real world itself. Instead, the truth is that our software is always an approximation of reality, and we will probably have more success by trying to get something working sooner and with reasonable complexity, instead of trying to create near-perfect software with huge effort and tons of code to maintain.

Throughout this book, you will see this principle in action many times. For example, a considerable number of traditional design patterns, such as Singleton or Decorator, can have a trivial, even if sometimes not bulletproof, implementation, and you will see how an uncomplicated, practical approach is (most of the time) preferred to a pure, flawless design.

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 will gain an understanding of how Node.js works internally and be introduced to the reactor pattern, which is the heart of the asynchronous nature of Node.js. We will go through the main concepts behind the pattern, such as the single-threaded architecture and the non-blocking I/O, and you will see how this creates the foundation for the entire Node.js platform.

I/O is slow

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 ($10\text{E-}9$ seconds), while accessing data on the disk or the network is in the order of milliseconds ($10\text{E-}3$ 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 doesn't only depend on technical aspects, and it can be many orders of magnitude slower than the disk or network.

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 trivial to notice 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 illustrates this scenario:

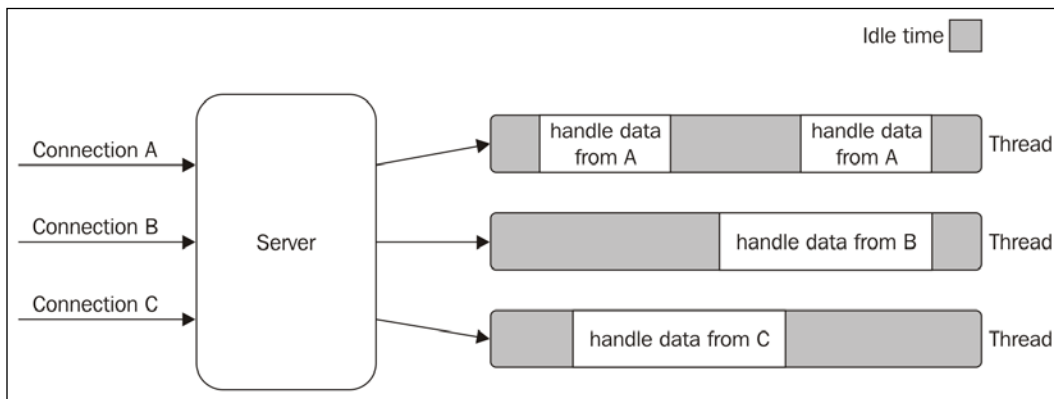


Figure 1.1: Using multiple threads to process multiple connections

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 has to 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.

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 manipulate an existing file descriptor (which in Unix represents the reference used to access a local file or a network socket) to change its operating mode to non-blocking (with the `O_NONBLOCK` flag). Once the resource is in non-blocking mode, any read operation will fail with the return code `EAGAIN` if the resource doesn't have any data ready to be read.

The most basic pattern for dealing with this type of non-blocking I/O is to actively poll the resource within a loop until some actual data is returned. This is called **busy-waiting**. The following pseudocode shows you how it's possible 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 only consume precious CPU for iterating over resources that are unavailable most of the time. Polling algorithms usually result in a huge amount of wasted CPU time.

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.

The synchronous event demultiplexer that we were talking about watches multiple resources and returns a new event (or set of events) when a read or write operation executed over one of those resources completes. The advantage here is that the synchronous event demultiplexer is, of course, synchronous, so it blocks until there are new events to process. The following is the pseudocode of an algorithm that uses a generic synchronous event demultiplexer to read from two different resources:

```
watchedList.add(socketA, FOR_READ) // (1)
watchedList.add(fileB, FOR_READ)
while (events = demultiplexer.watch(watchedList)) { // (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 to 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 clearer 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:

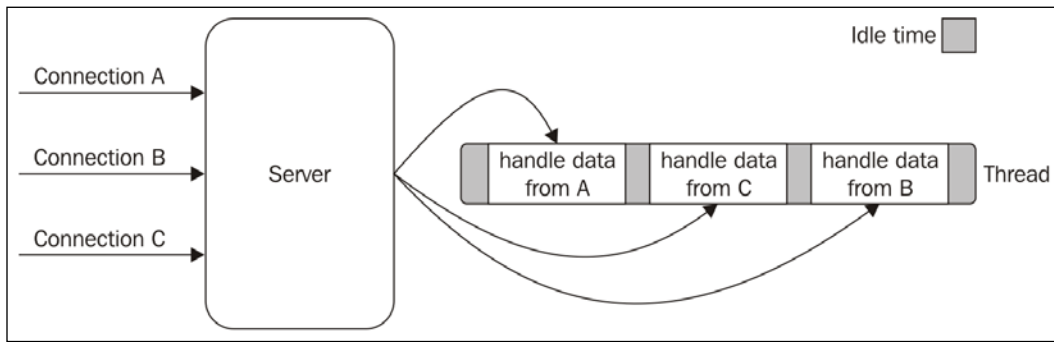


Figure 1.2: Using a single thread to process multiple connections

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 specialization of the algorithms presented in the previous sections. 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:

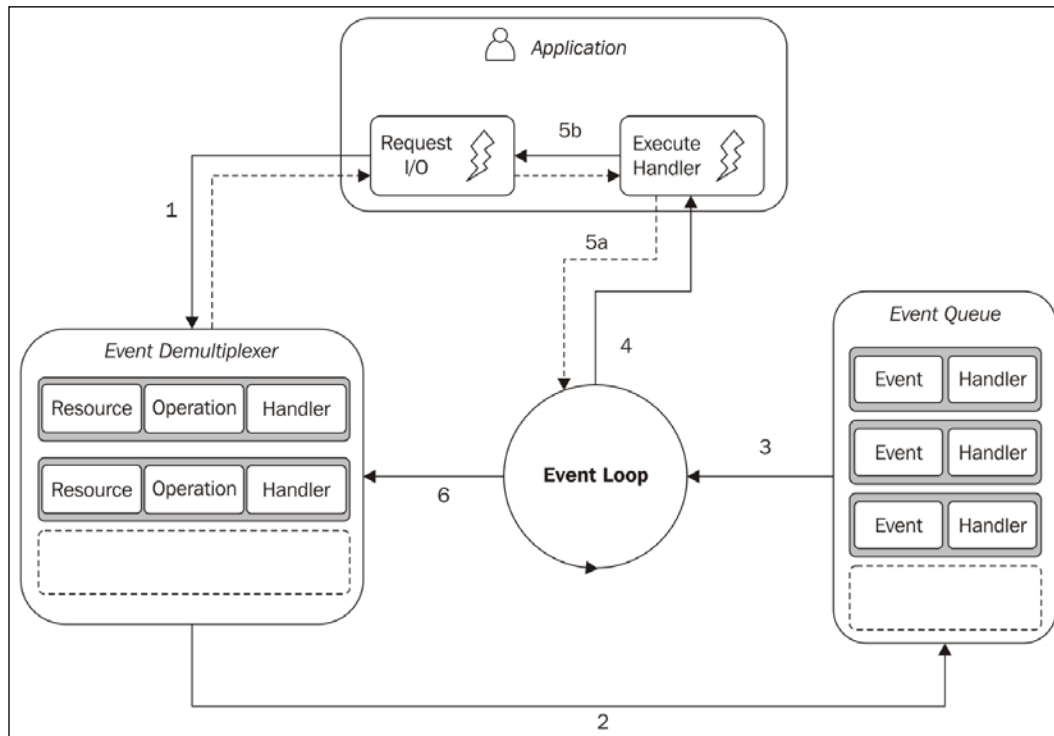


Figure 1.3: The reactor pattern

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 will exit when there are no more pending operations in the event demultiplexer, and no more events to be processed inside the event queue.

We can now define the pattern at the heart of Node.js:



The reactor pattern

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.

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 to make Node.js compatible with all the major operating systems and normalize the non-blocking behavior of the different types of resource. 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 task.



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 recipe for Node.js

The reactor pattern and libuv are the basic building blocks of Node.js, but we need three more components to build the full platform:

- 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 for 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:

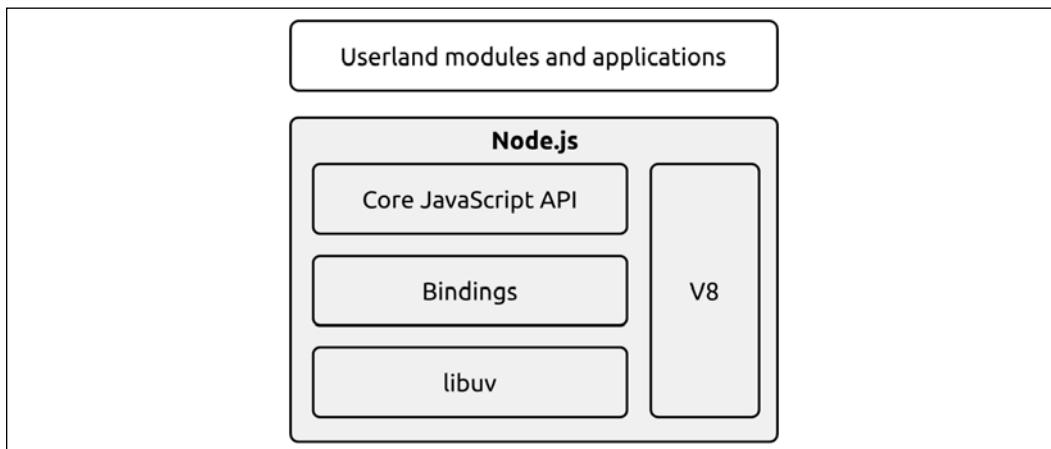


Figure 1.4: The Node.js internal components

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 most obvious difference is that in Node.js we don't have a DOM and we don't have a window or a document. On the other hand, Node.js has access to a set of services offered by the underlying operating system that are not available in the browser. In fact, the browser has to implement a set of safety measures to make sure that the underlying system is not compromised by a rogue web application. The browser provides a higher-level abstraction over the operating system resources, which makes it easier to control and contain the code that runs in it, which will also inevitably limit its capabilities. In turn, in Node.js we can virtually have access to all the services exposed by the operating system.

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 pain points of using JavaScript in the browser is that our code will likely run on a variety of devices and browsers. Dealing with different browsers means dealing with JavaScript runtimes that may miss some of the newest features of both the language or the web platform. Luckily, today this problem can be somewhat mitigated by the use of transpilers and polyfills. Nonetheless, this brings its own set of disadvantages and not everything can be polyfilled.

All these inconveniences don't apply when developing applications on Node.js. In fact, our Node.js applications will most likely run on a system and a Node.js runtime that are well known in advance. This makes a huge difference as it allows us to target our code for a specific JavaScript and Node.js version, with the absolute guarantee that we won't have any surprises when we run it on production.

This factor, in combination with the fact that Node.js ships with very recent versions of V8, means that we can use with confidence most of the features of the latest ECMAScript specification (ES for short; this is the standard on which the JavaScript language is based) without the need for any extra transpilation step.

Please bear in mind, though, that if we are developing a library meant to be used by third parties, we still have to take into account that our code may run on different versions of Node.js. The general pattern in this case is to target the oldest active **long-term support (LTS)** release and specify the engines section in our `package.json`, so that the package manager will warn the user if they are trying to install a package that is not 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 shipped with a module system, even when JavaScript still had no official support for any form of it. The original Node.js module system is called CommonJS and it uses the `require` keyword to import functions, variables, and classes exported by built-in modules or other modules located on the device's filesystem.

CommonJS was a revolution for the JavaScript world in general, as it started to get popular even in the client-side world, where it is used in combination with a module bundler (such as Webpack or Rollup) to produce code bundles that are easily executable by the browser. CommonJS was a necessary component for Node.js to allow developers to create large and better organized applications on a par with other server-side platforms.

Today, JavaScript has the so-called ES modules syntax (the `import` keyword may be more familiar) from which Node.js inherits just the syntax, as the underlying implementation is somewhat different from that of the browser. In fact, while the browser mainly deals with remote modules, Node.js, at least for now, can only deal with modules located on the local filesystem.

We'll talk about modules 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 code. This gives to the platform a tremendous advantage as it allows us to reuse existing or new components written in C/C++. Node.js officially provides great support for implementing native modules thanks to the N-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 tons 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 "understandable" 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.

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 three pillars: V8, libuv, and the core JavaScript library.

Finally, we analyzed some of the main characteristics of using JavaScript in Node.js compared to the browser.

Besides the obvious technical advantages enabled by its internal architecture, Node.js is attracting so much interest because of the principles you have just discovered and the community orbiting around it. For many, grasping the essence of this world feels like returning to the origins, to a more humane way of programming in both size and complexity, and that's why developers end up 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. But what do we actually 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 codebase into small units that can be developed and tested independently. Modules are also the main mechanism to enforce information hiding by keeping private all the functions and variables that are not explicitly marked to be exported.

If you come from other languages, you have probably seen similar concepts being referred to with different names: **package** (Java, Go, PHP, Rust, or Dart), **assembly** (.NET), **library** (Ruby), or **unit** (Pascal dialects). The terminology is not perfectly interchangeable because every language or ecosystem comes with its own unique characteristics, but there is a significant overlap between these concepts.

Interestingly enough, Node.js currently comes with two different module systems: **CommonJS (CJS)** and **ECMAScript modules (ESM or ES modules)**. In this chapter, we will discuss why there are two alternatives, we will learn about their pros and cons, and, finally, we will analyze several common patterns that are relevant when using or writing Node.js modules. By the end of this chapter, you should be able to make pragmatic choices about how to use modules effectively and how to write your own custom modules.

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 and the different module systems available in Node.js
- CommonJS internals and module patterns
- ES modules (ESM) in Node.js
- Differences and interoperability between CommonJS and ESM

Let's begin with why we need modules.

The need for modules

A good module system should help with addressing some fundamental needs of software engineering:

- *Having a way to split the codebase into multiple files.* This helps with keeping the code more organized, making it easier to understand but also helps with developing and testing various pieces of functionality independently from each other.
- *Allowing code reuse across different projects.* A module can, in fact, implement a generic feature that can be useful for different projects. Organizing such functionality within a module can make it easier to bring it into the different projects that may want to use it.
- *Encapsulation (or information hiding).* It is generally a good idea to hide implementation complexity and only expose simple interfaces with clear responsibilities. Most module systems allow to selectively keep the *private* part of the code hidden, while exposing a *public* interface, such as functions, classes, or objects that are meant to be used by the consumers of the module.
- *Managing dependencies.* A good module system should make it easy for module developers to build on top of existing modules, including third-party ones. A module system should also make it easy for module users to import the chain of dependencies that are necessary for a given module to run (transient dependencies).

It is important to clarify the distinction between *a module* and *a module system*. We can define a module as the actual unit of software, while a module system is the syntax and the tooling that allows us to define modules and to use them within our projects.

Module systems in JavaScript and Node.js

Not all programming languages come with a built-in module system, and JavaScript had been lacking this feature for a long time.

In the browser landscape, it is possible to split the codebase 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 complicated and frameworks like *jQuery*, *Backbone*, and *AngularJS* took over the ecosystem, the JavaScript community came 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 (sometimes also referred to as *CJS*, nodejsdp.link/commonjs), which was designed to provide a module system for JavaScript in browserless environments.

CommonJS has been the dominant module system in Node.js since its inception and it has become very prominent also in the browser landscape thanks to *module bundlers* like Browserify (nodejsdp.link/browserify) and webpack (nodejsdp.link/webpack).

In 2015, with the release of *ECMAScript 6* (also called *ECMAScript 2015* or *ES2015*), there was finally an official proposal for a standard module system: *ESM* or *ECMAScript modules*. ESM brings a lot of innovation in the JavaScript ecosystem and, among other things, it tries to bridge the gap between how modules are managed on browsers and servers.

ECMAScript 6 defined only the formal specification for ESM in terms of syntax and semantics, but it didn't provide any implementation details. It took different browser companies and the Node.js community several years to come up with solid implementations of the specification. Node.js ships with stable support for ESM starting from version 13.2.

At the time of writing, the general feeling is that ESM is going to become the de facto way to manage JavaScript modules in both the browser and the server landscape. The reality today, though, is that the majority of projects are still heavily relying on CommonJS and it will take some time for ESM to catch up and eventually become the dominant standard.

To provide a comprehensive overview of module-related patterns in Node.js, in the first part of this chapter, we will discuss them in the context of CommonJS, and then, in the second part of the chapter, we will revisit our learnings using ESM.

The goal of this chapter is to make you comfortable with both module systems, but in the rest of the book, we will only be using ESM for our code examples. The idea is to encourage you to leverage ESM as much as possible so that your code will be more future-proof.

If you are reading this chapter a few years after its publication, you are probably not too worried about CommonJS, and you might want to jump straight into the ESM part. This is probably fine, but we still encourage you to go through the entire chapter, because understanding CommonJS and its characteristics will certainly be beneficial in helping you to understand ESM and its strengths in much more depth.

The module system and its patterns

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

Before getting into the specifics of CommonJS, let's discuss a generic pattern that helps with information hiding and that we will be using for building a simple module system, which is the **revealing module pattern**.

The revealing module pattern

One of the bigger problems with JavaScript in the browser is the lack of namespacing. Every script runs in the global scope; therefore, internal application code or third-party dependencies can pollute the scope while exposing their own pieces of functionality. This can be extremely harmful. Imagine, for instance, that a third-party library instantiates a global variable called `utils`. If any other library, or the application code itself, accidentally overrides or alters `utils`, the code that relies on it will likely crash in some unpredictable way. Unpredictable side effects can also happen if other libraries or the application code accidentally invoke a function of another library meant for internal use only.

In short, relying on the global scope is a very risky business, especially as your application grows and you have to rely more and more on functionality implemented by other individuals.

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 = []

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

  return exported
})(); // once the parenthesis here are parsed, the function
    // will be invoked

console.log(myModule)
console.log(myModule.privateFoo, myModule.privateBar)
```

This pattern leverages a self-invoking function. This type of function is sometimes also referred to as **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 is essentially exploiting 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 outside.

The log statement is going to print something like this:

```
{ publicFoo: [Function: publicFoo],
  publicBar: [Function: publicBar] }
undefined undefined
```

This demonstrates that only the exported properties are directly accessible from `myModule`.

As we will see in a moment, the idea behind this pattern is used as a base for the CommonJS module system.

CommonJS modules

CommonJS is the first module system originally built into Node.js. Node.js' CommonJS implementation respects the CommonJS specification, with the addition of some custom extensions.

Let's summarize two of the main concepts of the CommonJS specification:

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

This information is sufficient for now; we will learn more details and some of the nuances of the CommonJS specification in the next few sections.

A homemade module loader

To understand how CommonJS works in Node.js, let's build a similar system from scratch. The code that follows creates a function that mimics a subset of the functionality of the original `require()` function of Node.js.

Let's start by creating a function that loads the content of a module, wraps it into a private scope, and evaluates it:

```
function loadModule (filename, module, require) {  
  const wrappedSrc =  
    `(function (module, exports, require) {  
      ${fs.readFileSync(filename, 'utf8')}  
    })(module, module.exports, require)`  
  eval(wrappedSrc)  
}
```

The source code of a module is essentially wrapped into a function, as it was for the revealing module pattern. The difference here is that we pass a list of variables to the module, in particular, `module`, `exports`, and `require`. Make a note of how the `exports` argument of the wrapping function is initialized with the content of `module.exports`, as we will talk about this later.

Another important detail to mention is that we are using `readFileSync` to read the module's content. While it is generally not recommended to use the synchronous version of the filesystem APIs, here it makes sense to do so. The reason for that is that loading modules in CommonJS are deliberately synchronous operations. This approach makes sure that, if we are importing multiple modules, they (and their dependencies) are loaded in the right order. We will talk more about this aspect later in the chapter.



Bear in mind that this is only an example, and you will rarely need to evaluate some source code in a real application. Features such as `eval()` or the functions of the `vm` module ([nodejsdp.link/vm](https://nodejs.org/docs/latest/api/vm)) can be easily used in the wrong way or with the wrong input, thus opening a system to code injection attacks. They should always be used with extreme care or avoided altogether.

Let's now implement the `require()` function:

```
function require (moduleName) {
  console.log(`Require invoked for module: ${moduleName}`)
  const id = require.resolve(moduleName) // (1)
  if (require.cache[id]) { // (2)
    return require.cache[id].exports
  }

  // module metadata
  const module = { // (3)
    exports: {},
    id
  }
  // Update the cache
  require.cache[id] = module // (4)

  // Load the module
  loadModule(id, module, require) // (5)

  // return exported variables
  return module.exports // (6)
}
require.cache = {}
require.resolve = (moduleName) => {
  /* resolve a full module id from the moduleName */
}
```


The previous function simulates the behavior of the original `require()` function of Node.js, which is used to load a module. Of course, this is just for educational purposes and does not accurately or completely reflect the internal behavior of the real `require()` function, but it's great to understand the internals of the Node.js module system, including how a module is defined and loaded.

What our homemade module system does is explained as follows:

1. A module name is accepted as input, and the very first thing that we do is resolve the full path of the module, which we call `id`. This task is delegated to `require.resolve()`, which implements a specific resolving algorithm (we will talk about it later).
2. If the module has already been loaded in the past, it should be available in the cache. If this is the case, we just return it immediately.
3. If the module has never been loaded before, we set up the environment for the first load. In particular, we create a `module` object that contains an `exports` property initialized with an empty object literal. This object will be populated by the code of the module to export its public API.
4. After the first load, the `module` object is cached.
5. The module source code is read from its file and the code is evaluated, as we saw before. We provide the module with the `module` object that we just created, and a reference to the `require()` function. The module exports its public API by manipulating or replacing the `module.exports` object.
6. Finally, the content of `module.exports`, which represents the public API of the module, is returned to the caller.

As we can see, there is nothing magical behind the workings of the Node.js module system. The trick is all in the wrapper we create around a module's source code and the artificial environment in which we run it.

Defining a module

By looking at how our custom `require()` function works, we should now be able to understand how to define a module. The following code gives us an example:

```
// Load another dependency
const dependency = require('./anotherModule')

// a private function
function log() {
  console.log(`Well done ${dependency.username}`)
```

```
}  
  
// the API to be exported for public use  
module.exports.run = () => {  
  log()  
}
```

The essential concept to remember is that everything inside a module is private unless it's assigned to the `module.exports` variable. The content of this variable is then cached and returned when the module is loaded using `require()`.

module.exports versus exports

For many developers who are not yet familiar with Node.js, a common source of confusion is the difference between using `exports` and `module.exports` to expose a public API. The code of our custom `require()` function should again clear any doubt. The `exports` variable is just a reference to the initial value of `module.exports`. We have seen that such a value is essentially a simple object literal created before the module is loaded.

This means that we can only attach new properties to the object referenced by the `exports` variable, as shown in the following code:

```
exports.hello = () => {  
  console.log('Hello')  
}
```

Reassigning the `exports` variable doesn't have any effect, because it doesn't change the content of `module.exports`. It will only reassign the variable itself. The following code is therefore wrong:

```
exports = () => {  
  console.log('Hello')  
}
```

If we want to export something other than an object literal, such as a function, an instance, or even a string, we have to reassign `module.exports` as follows:

```
module.exports = () => {  
  console.log('Hello')  
}
```

The require function is synchronous

A very important detail that we should take into account is that our homemade `require()` function is synchronous. In fact, it returns the module contents using a simple direct style, and no callback is required. This is true for the original Node.js `require()` function too. As a consequence, any assignment to `module.exports` must be synchronous as well. For example, the following code is incorrect and it will cause trouble:

```
setTimeout(() => {  
  module.exports = function() {...}  
}, 100)
```

The synchronous nature of `require()` has important repercussions on the way we define modules, as it limits us to mostly using synchronous code during the definition of a module. This is one of the most important reasons why the core Node.js libraries offer synchronous APIs as an alternative to most of the asynchronous ones.

If we need some asynchronous initialization steps for a module, we can always define and export an uninitialized module that is initialized asynchronously at a later time. The problem with this approach, though, is that loading such a module using `require()` does not guarantee that it's ready to be used. In *Chapter 11, Advanced Recipes*, we will analyze this problem in detail and present some patterns to solve this issue elegantly.

For the sake of curiosity, you might want to know that in its early days, Node.js used to have an asynchronous version of `require()`, but it was soon removed because it was overcomplicating a functionality that was actually only meant to be used at initialization time and where asynchronous I/O brings more complexities than advantages.

The resolving algorithm

The term *dependency hell* describes a situation whereby two or more dependencies of a program in turn depend on a shared dependency, but require different incompatible versions. Node.js solves this problem elegantly by loading a different version of a module depending on where the module is loaded from. All the merits of this feature go to the way Node.js package managers (such as npm or yarn) organize the dependencies of the application, and also to the resolving algorithm used in the `require()` function.

Let's now give a quick overview of this algorithm. As we saw, the `resolve()` function takes a module name (which we will call `moduleName`) as input and it returns the full path of the module. This path is then used to load its code and also to identify the module uniquely. The resolving algorithm can be divided into the following three major branches:

- **File modules:** If `moduleName` starts with `/`, it is already considered an absolute path to the module and it's returned as it is. If it starts with `./`, then `moduleName` is considered a relative path, which is calculated starting from the directory of the requiring module.
- **Core modules:** If `moduleName` is not prefixed with `/` or `./`, the algorithm will first try to search within the core Node.js modules.
- **Package modules:** If no core module is found matching `moduleName`, 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 requiring 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.

For file and package modules, both files and directories can match `moduleName`. In particular, the algorithm will try to match the following:

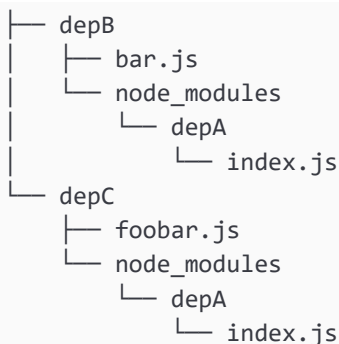
- `<moduleName>.js`
- `<moduleName>/index.js`
- The directory/file specified in the `main` property of `<moduleName>/package.json`



The complete, formal documentation of the resolving algorithm can be found at [nodejsdp.link/resolve](https://nodejs.org/en/docs/guides/modules/).

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
```



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, using `require('depA')` will load a different file depending on the module that requires it, for example:

- Calling `require('depA')` from `/myApp/foo.js` will load `/myApp/node_modules/depA/index.js`
- Calling `require('depA')` from `/myApp/node_modules/depB/bar.js` will load `/myApp/node_modules/depB/node_modules/depA/index.js`
- Calling `require('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.



The resolving algorithm is applied transparently for us when we invoke `require()`. However, if needed, it can still be used directly by any module by simply invoking `require.resolve()`.

The module cache

Each module is only loaded and evaluated the first time it is required, since any subsequent call of `require()` will simply return the cached version. This should be clear by looking at the code of our homemade `require()` function. Caching is crucial for performance, but it also has some important functional implications:

- It makes it possible to have cycles within module dependencies
- It guarantees, to some extent, that the same instance is always returned when requiring the same module from within a given package

The module cache is exposed via the `require.cache` variable, so it is possible to directly access it if needed. A common use case is to invalidate any cached module by deleting the relative key in the `require.cache` variable, a practice that can be useful during testing but very dangerous if applied in normal circumstances.

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 at least how this works with CommonJS. If we look again at our homemade `require()` function, we immediately get a glimpse of how this might work and what its caveats are.

But let's walk together through an example to see how CommonJS behaves when dealing with circular dependencies. Let's suppose we have the scenario represented in *Figure 2.1*:

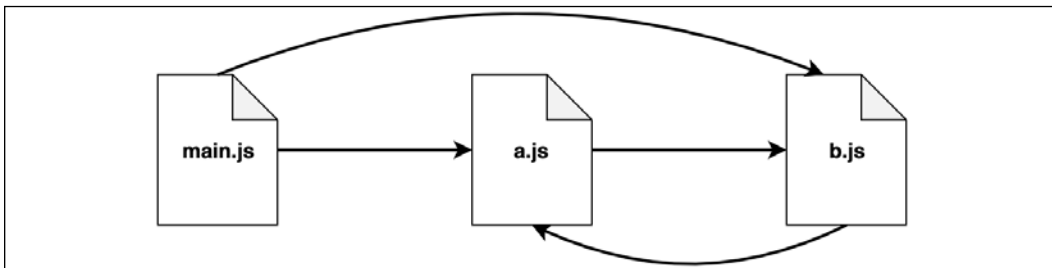


Figure 2.1: An example of circular dependency

A module called `main.js` requires `a.js` and `b.js`. In turn, `a.js` requires `b.js`. But `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 have a look at the code of these two modules:

- Module `a.js`:

```
exports.loaded = false
const b = require('./b')
module.exports = {
  b,
  loaded: true // overrides the previous export
}
```

- Module `b.js`:

```
exports.loaded = false
const a = require('./a')
module.exports = {
```

```
    a,  
    loaded: true  
  }
```

Now, let's see how these modules are required by `main.js`:

```
const a = require('./a')  
const b = require('./b')  
console.log('a ->', JSON.stringify(a, null, 2))  
console.log('b ->', JSON.stringify(b, null, 2))
```

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

```
a -> {  
  "b": {  
    "a": {  
      "loaded": false  
    },  
    "loaded": true  
  },  
  "loaded": true  
}  
  
b -> {  
  "a": {  
    "loaded": false  
  },  
  "loaded": true  
}
```

This result reveals the caveats of circular dependencies with CommonJS, that is, different parts of our application will have a different view of what is exported by module `a.js` and module `b.js`, depending on the order in which those dependencies are loaded. While both the modules are completely initialized as soon as they are required from the module `main.js`, the `a.js` module will be incomplete when it is loaded from `b.js`. In particular, its state will be the one that it reached the moment `b.js` was required.

In order to understand in more detail what happens behind the scenes, let's analyze step by step how the different modules are interpreted and how their local scope changes along the way:

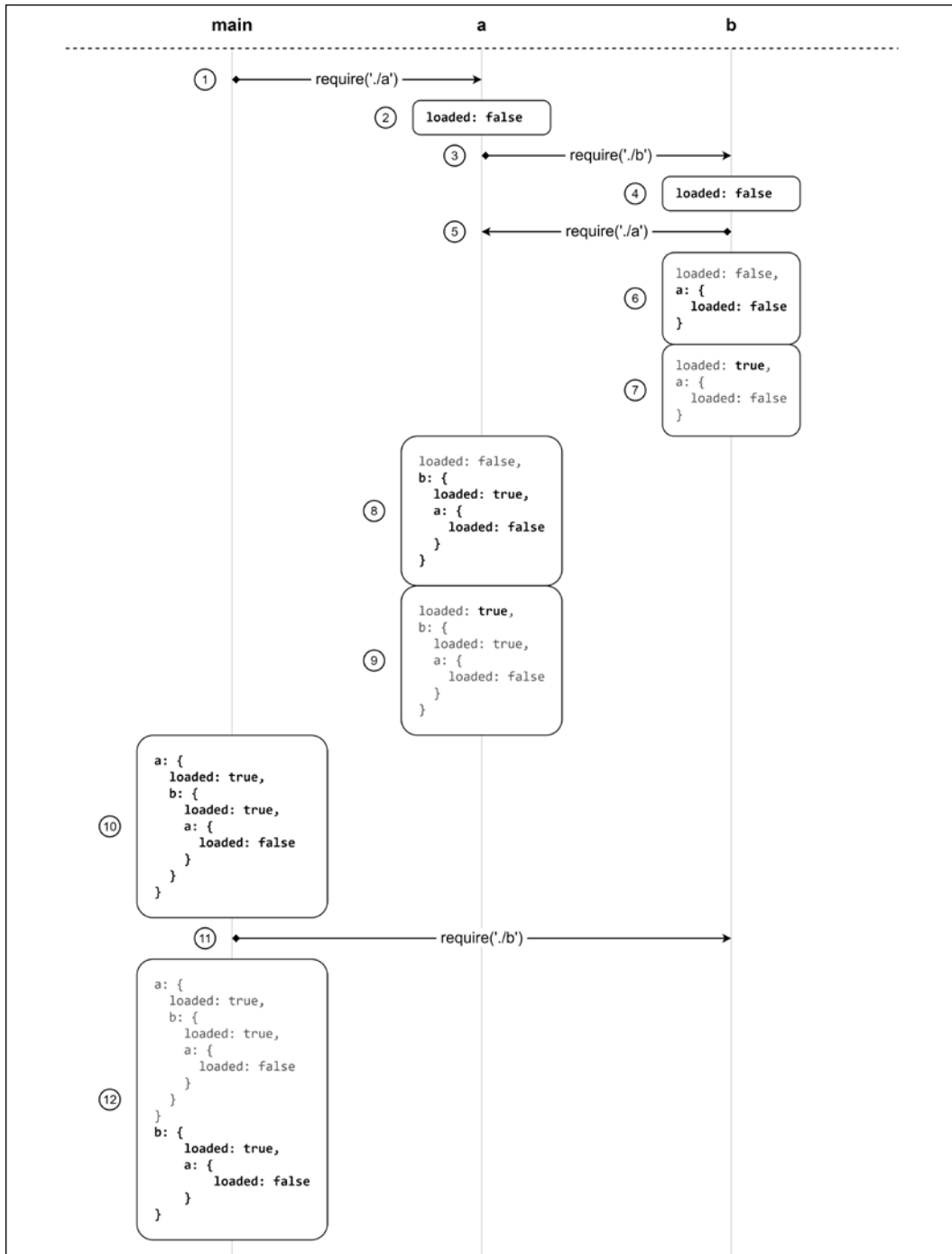


Figure 2.2: A visual representation of how a dependency loop is managed in Node.js

The steps are as follows:

1. The processing starts in `main.js`, which immediately requires `a.js`
2. The first thing that module `a.js` does is set an exported value called `loaded` to `false`
3. At this point, module `a.js` requires module `b.js`
4. Like `a.js`, the first thing that module `b.js` does is set an exported value called `loaded` to `false`
5. Now, `b.js` requires `a.js` (cycle)
6. Since `a.js` has already been traversed, its currently exported value is immediately copied into the scope of module `b.js`
7. Module `b.js` finally changes the `loaded` value to `true`
8. Now that `b.js` has been fully executed, the control returns to `a.js`, which now holds a copy of the current state of module `b.js` in its own scope
9. The last step of module `a.js` is to set its `loaded` value to `true`
10. Module `a.js` is now completely executed and the control returns to `main.js`, which now has a copy of the current state of module `a.js` in its internal scope
11. `main.js` requires `b.js`, which is immediately loaded from cache
12. The current state of module `b.js` is copied into the scope of module `main.js` where we can finally see the complete picture of what the state of every module is

As we said, the issue here is that module `b.js` has a partial view of module `a.js`, and this partial view gets propagated over when `b.js` is required in `main.js`. This behavior should spark an intuition which can be confirmed if we swap the order in which the two modules are required in `main.js`. If you actually try this, you will see that this time it will be the `a.js` module that will receive an incomplete version of `b.js`.

We understand now that this can become quite a fuzzy business if we lose control of which module is loaded first, which can happen quite easily if the project is big enough.

Later in this chapter, we will see how ESM can deal with circular dependencies in a more effective way. Meanwhile, if you are using CommonJS, be very careful about this behavior and the way it can affect your application.

In the next section, we will discuss some patterns to define modules in Node.js.

Module definition patterns

The module system, besides being a mechanism for loading dependencies, is also a tool for defining APIs. Like any other problem related to API design, the main factor to consider is the balance between private and public functionality. The aim is to maximize information hiding and API usability, while balancing these with other software qualities, such as *extensibility* and *code reuse*.

In this section, we will analyze some of the most popular patterns for defining modules in Node.js, such as named exports, exporting functions, classes and instances, and monkey patching. Each one has its own balance of information hiding, extensibility, and code reuse.

Named exports

The most basic method for exposing a public API is using **named exports**, which involves assigning the values we want to make public to properties of the object referenced by exports (or `module.exports`). In this way, the resulting exported object becomes a container or namespace for a set of related functionalities.

The following code shows a module implementing this pattern:

```
// file logger.js
exports.info = (message) => {
  console.log(`info: ${message}`)
}

exports.verbose = (message) => {
  console.log(`verbose: ${message}`)
}
```

The exported functions are then available as properties of the loaded module, as shown in the following code:

```
// file main.js
const logger = require('./logger')
logger.info('This is an informational message')
logger.verbose('This is a verbose message')
```

Most of the Node.js core modules use this pattern. However, the CommonJS specification only allows the use of the `exports` variable to expose public members. Therefore, the named exports pattern is the only one that is really compatible with the CommonJS specification. The use of `module.exports` is an extension provided by Node.js to support a broader range of module definition patterns, which we are going to see next.

Exporting a function

One of the most popular module definition patterns consists of reassigning the whole `module.exports` variable to a function. The main strength of this pattern is the fact that it allows you to expose only a single functionality, which provides a clear entry point for the module, making it simpler to understand and use; it also honors the principle of *small surface area* very well. This way of defining modules is also known in the community as the **substack pattern**, after one of its most prolific adopters, James Halliday (nickname substack – <https://github.com/substack>). Have a look at this pattern in the following example:

```
// file logger.js
module.exports = (message) => {
  console.log(`info: ${message}`)
}
```

A possible extension of this pattern is using the exported function as a namespace for other public APIs. This is a very powerful combination because it still gives the module the clarity of a single entry point (the main exported function) and at the same time it allows us to expose other functionalities that have secondary or more advanced use cases. The following code shows us how to extend the module we defined previously by using the exported function as a namespace:

```
module.exports.verbose = (message) => {
  console.log(`verbose: ${message}`)
}
```

This code demonstrates how to use the module that we just defined:

```
// file main.js
const logger = require('./logger')
logger('This is an informational message')
logger.verbose('This is a verbose message')
```

Even though exporting just a function might seem like a limitation, in reality, it's a perfect way to put the emphasis on a single functionality, the most important one for the module, while giving less visibility to secondary or internal aspects, which are instead exposed as properties of the exported function itself. The modularity of Node.js heavily encourages the adoption of the **single-responsibility principle (SRP)**: every module should have responsibility over a single functionality and that responsibility should be entirely encapsulated by the module.

Exporting a class

A module that exports a class is a specialization of a module that exports a function. The difference is that with this new pattern we allow the user to create new instances using the constructor, but we also give them the ability to extend its prototype and forge new classes. The following is an example of this pattern:

```
class Logger {
  constructor (name) {
    this.name = name
  }

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

  info (message) {
    this.log(`info: ${message}`)
  }

  verbose (message) {
    this.log(`verbose: ${message}`)
  }
}

module.exports = Logger
```

And, we can use the preceding module as follows:

```
// file main.js
const Logger = require('./logger')
const dbLogger = new Logger('DB')
dbLogger.info('This is an informational message')
const accessLogger = new Logger('ACCESS')
accessLogger.verbose('This is a verbose message')
```

Exporting a class still provides a single entry point for the module, but compared to the substack pattern, it exposes a lot more of the module internals. On the other hand, it allows much more power when it comes to extending its functionality.

Exporting an instance

We can leverage the caching mechanism of `require()` to easily define stateful instances created from a constructor or a factory, which can be shared across different modules. The following code shows an example of this pattern:

```
// file logger.js
class Logger {
  constructor (name) {
    this.count = 0
    this.name = name
  }
  log (message) {
    this.count++
    console.log('[ ' + this.name + ' ] ' + message)
  }
}
module.exports = new Logger('DEFAULT')
```

This newly defined module can then be used as follows:

```
// main.js
const logger = require('./logger')
logger.log('This is an informational message')
```

Because the module is cached, every module that requires the `logger` module will actually always retrieve the same instance of the object, thus sharing its state. This pattern is very much like creating a **singleton**. However, it does not guarantee the uniqueness of the instance across the entire application, as it happens in the traditional singleton pattern. When analyzing the resolving algorithm, we have seen that a module might be installed multiple times inside the dependency tree of an application. This results in multiple instances of the same logical module, all running in the context of the same Node.js application. We will analyze the Singleton pattern and its caveats in more detail in *Chapter 7, Creational Design Patterns*.

One interesting detail of this pattern is that it does not preclude the opportunity to create new instances, even if we are not explicitly exporting the class. In fact, we can rely on the constructor property of the exported instance to construct a new instance of the same type:

```
const customLogger = new logger.constructor('CUSTOM')
customLogger.log('This is an informational message')
```

As you can see, by using `logger.constructor()`, we can instantiate new `Logger` objects. Note that this technique must be used with caution or avoided altogether. Consider that, if the module author decided not to export the class explicitly, they probably wanted to keep this class private.

Modifying other modules or the global scope

A module can even export nothing. This can seem a bit out of place; however, we should not forget that a module can modify the global scope and any object in it, including other modules in the cache. Please note that these are in general considered bad practices, but since this pattern can be useful and safe under some circumstances (for example, for testing) and it's sometimes used in real-life projects, it's worth knowing.

We said that a module can modify other modules or objects in the global scope; well, this is called **monkey patching**. It generally refers to the practice of modifying the existing objects at runtime to change or extend their behavior or to apply temporary fixes.

The following example shows us how we can add a new function to another module:

```
// file patcher.js

// ../logger is another module
require('../logger').customMessage = function () {
  console.log('This is a new functionality')
}
```

Using our new patcher module is as easy as writing the following code:

```
// file main.js

require('./patcher')
const logger = require('../logger')
logger.customMessage()
```

The technique described here can be very dangerous to use. The main concern is that having a module that modifies the global namespace or other modules is an operation with *side effects*. In other words, it affects the state of entities outside their scope, which can have consequences that aren't easily predictable, especially when multiple modules interact with the same entities. Imagine having two different modules trying to set the same global variable, or modifying the same property of the same module. The effects can be unpredictable (which module wins?), but most importantly it would have repercussions on the entire application.

So, again use this technique with care and make sure you understand all the possible side effects while doing so.



If you want a real-life example of how this can be useful, have a look at `nock` (nodejsdp.link/nock), a module that allows you to mock HTTP responses in your tests. The way `nock` works is by monkey patching the Node.js `http` module and by changing its behavior so that it will provide the mocked response rather than issuing a real HTTP request. This allows our unit test to run without hitting the actual production HTTP endpoints, something that's very convenient when writing tests for code that relies on third-party APIs.

At this point, we should have a quite complete understanding of CommonJS and some of the patterns that are generally used with it. In the next section, we will explore ECMAScript modules, also known as ESM.

ESM: ECMAScript modules

ECMAScript modules (also known as ES modules or ESM) were introduced as part of the ECMAScript 2015 specification with the goal to give JavaScript an official module system suitable for different execution environments. The ESM 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.

The most important differentiator between ESM and CommonJS is that 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'  
}
```

While in CommonJS, it is perfectly fine to write something like this:

```
let module = null  
if (condition) {
```

```
module = require('module1')
} else {
  module = require('module2')
}
```

At a first glance, this characteristic of ESM might seem an unnecessary limitation, but in reality, having static imports opens up a number of interesting scenarios that are not practical with the dynamic nature of CommonJS. For instance, static imports allow the static analysis of the dependency tree, which allows optimizations such as dead code elimination (tree shaking) and more.

Using ESM in Node.js

Node.js will consider every `.js` file to be written using the CommonJS syntax by default; therefore, if we use the ESM syntax inside a `.js` file, the interpreter will simply throw an error.

There are several ways to tell the Node.js interpreter to consider a given module as an ES module rather than a CommonJS module:

- Give the module file the extension `.mjs`
- Add to the nearest parent `package.json` a field called `"type"` with a value of `"module"`



Throughout the rest of this book and in the code examples provided, we will keep using the `.js` extension to keep the code more easily accessible to most text editors, 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 ESM syntax.

Named exports and imports

ESM allows us to export functionality from a module through the `export` keyword.



Note that ESM uses the singular word `export` as opposed to the plural (`exports` and `module.exports`) used by CommonJS.

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 to 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 very important to note that, as opposed to CommonJS, with ESM we have to specify the file extension of the imported modules. With CommonJS we can use either `./logger` or `./logger.js`, with ESM we are forced to use `./logger.js`.

If we are using a large module, most often we don't want to import all of its functionality, but only one or few entities from it:

```
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 { log, Logger } 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.

Default exports and imports

One widely used feature of CommonJS is the ability to export a single unnamed entity through the assignment of `module.exports`. We saw that this is very convenient as it encourages module developers to follow the single-responsibility principle and expose only one clear interface. With ESM, we can do something similar through what's called 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 is handled in a special way, and it 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 ESM 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. This is very similar to what we do with CommonJS modules. 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)
```

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

```
[Module] { default: [Function: 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 preceding example, we are importing the default export from `logger.js` as `mylog` and also the named export `info`.

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

- 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 '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.
- 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.
- 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.

For these reasons, it is generally considered good practice to stick with named exports, especially when you want to expose more than one functionality, and only use default exports if it's one clear functionality you want to 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 seen only 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`. They are used to refer to a path relative to the location of the importing file.
- *Absolute specifiers* like `file:///opt/nodejs/config.js`. They refer directly and explicitly to a full path. Note that this is the only way with ESM to refer to an absolute path for a module, using a `/` or a `//` prefix won't work. This is a significant difference with CommonJS.
- *Bare specifiers* are identifiers like `fastify` or `http`, and 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.
- *Deep import specifiers* like `fastify/lib/logger.js`, which 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.

Async imports

As we have seen in the previous section, the `import` statement is static and therefore subject to two important limitations:

- A module identifier cannot be constructed at runtime
- Module imports are declared at the top level of every file and they cannot be nested within control flow statements

There are some use cases when these limitations can become a little bit too restrictive. Imagine, for instance, if we have to import a specific translation module for the current user language, or a variation of a module that depends on the user's operating system.

Also, what if we want to load a given module, which might be particularly heavy, only if the user is accessing the piece of functionality that requires that module?

To allow us to overcome these limitations ES modules provides *async imports* (also called *dynamic imports*).

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 and 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:

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

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

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

// strings-it.js
export const HELLO = 'Ciao mondo'

// strings-pl.js
export const HELLO = 'Witaj świecie'
```

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 (!SUPPORTED_LANGUAGES.includes(selectedLanguage)) { // (3)
  console.error('The specified language is not supported')
  process.exit(1)
}

const translationModule = `./strings-${selectedLanguage}.js` // (4)
import(translationModule) // (5)
  .then((strings) => { // (6)
    console.log(strings.HELLO)
  })
```

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

1. Define a list of supported languages.
2. Read the selected language from the first argument passed in the command line.
3. 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:

4. 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 module file, that's why we are prepending `./` to the filename.
5. We use the `import()` operator to trigger the dynamic import of the module.
6. The dynamic import happens asynchronously, so we can use the `.then()` hook on the returned promise to get notified when the module is ready to be used. The function passed to `then()` will be executed when the module is fully loaded and `strings` will be the module namespace imported dynamically. After that, we can access `strings.HELLO` and print its value to the console.

Now we can execute this script like this:

```
node main.js it
```

And we should see *Ciao mondo* being printed to our console.

Module loading in depth

To understand how ESM actually works and how it can deal effectively with circular dependencies, we have to deep dive a little bit more into how JavaScript code is parsed and evaluated when using ES modules.

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

Loading phases

The goal of the interpreter is to build a graph of all the necessary modules (a **dependency graph**).



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 ECMAScript 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.

More specifically, this process happens in three separate phases:

- **Phase 1 - Construction (or parsing):** Find all the imports and recursively load the content of every module from the respective file.

- **Phase 2 - Instantiation:** For every exported entity, keep a named reference in memory, but don't assign any value just yet. Also, references are created for all the import and export statements tracking the dependency relationship between them (**linking**). No JavaScript code has been executed at this stage.
- **Phase 3 - Evaluation:** Node.js finally executes the code so that all the previously instantiated entities can get an actual value. Now running the code from the entry point is possible because all the blanks have been filled.

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

At first glance, this approach doesn't seem very different from what CommonJS does, but there's a fundamental difference. Due to its dynamic nature, CommonJS will execute all the files while the dependency graph is explored. We have seen that every time a new `require` statement is found, all the previous code has already been executed. This is why you can use `require` even within `if` statements or loops, and construct module identifiers from variables.

In ESM, these three phases are totally separate from each other, no code can be executed until the dependency graph has been fully built, and therefore module imports and exports have to be static.

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 string is changed at a later time, the requiring module won't be able to see those changes.

Circular dependency resolution

Now to close the circle, let's reimplement the circular dependency example we saw in the *CommonJS modules* section using the ESM syntax:

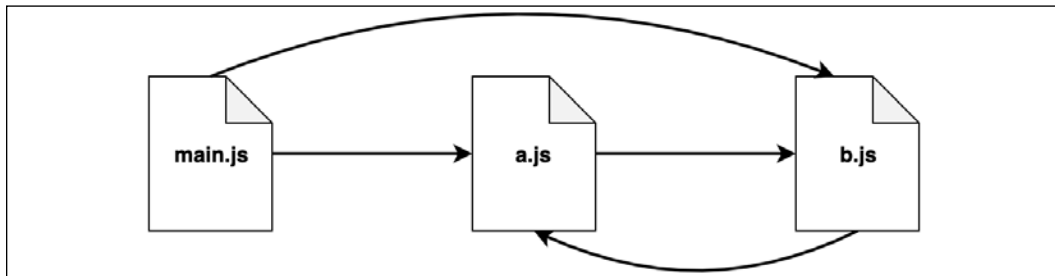


Figure 2.3: An example scenario with circular dependencies

Let's have a look at the 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
```

And 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)
```

Note that this time we are not using `JSON.stringify` because that will fail with a `TypeError: Converting circular structure to JSON`, since there's an actual circular reference between `a.js` and `b.js`.

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

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

The interesting bit here is that the modules `a.js` and `b.js` have a complete picture of each other, unlike what would happen with `CommonJS`, where they would only hold partial information of each other. We can see that because all the `loaded` values are set to `true`. Also, `b` within `a` is an actual reference to the same `b` instance available in the current scope, and the same goes for `a` within `b`. That's the reason why we cannot use `JSON.stringify()` to serialize these modules. Finally, if we swap the order of the imports for the modules `a.js` and `b.js`, the final outcome does not change, which is another important difference in comparison with how `CommonJS` works

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 that looks like a tree structure, as shown in *Figure 2.4*:

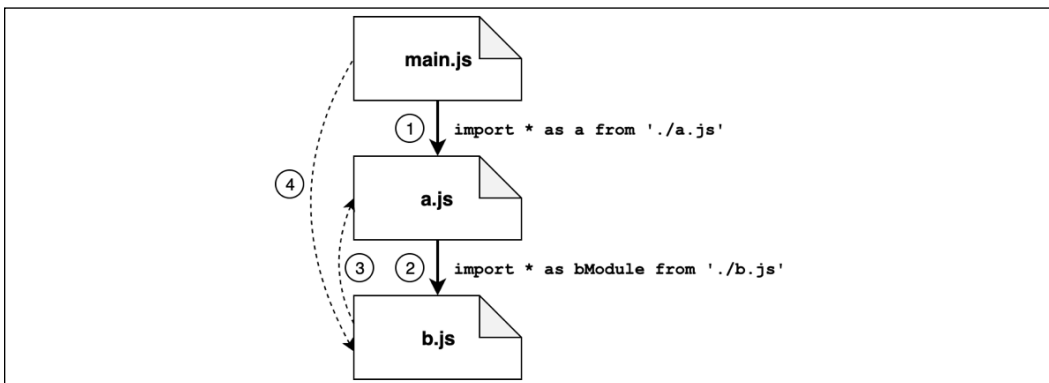


Figure 2.4: Parsing of cyclic dependencies with ESM

Given the example in *Figure 2.4*, 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.

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 2.5*:

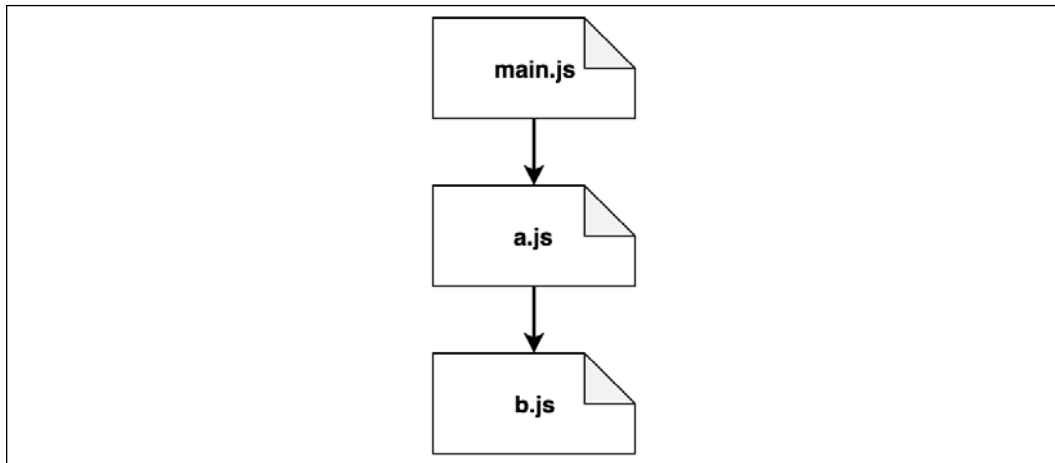


Figure 2.5: A linear view of the module graph where cycles have been removed

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 build out a map of the exported names in memory:

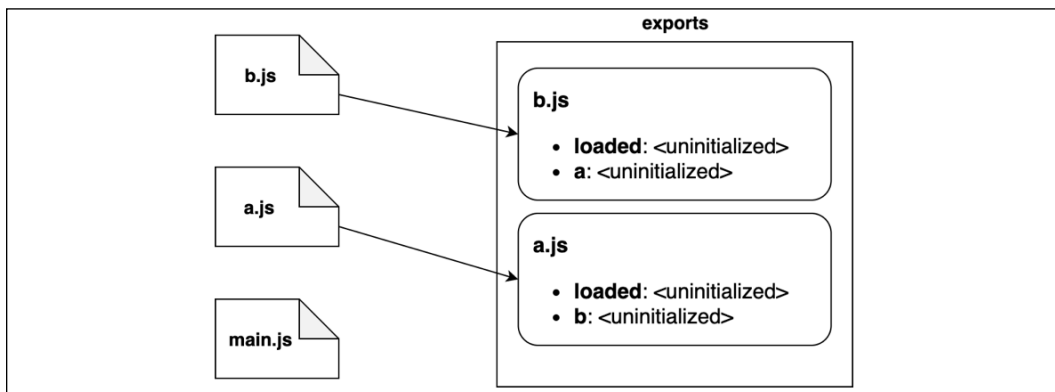


Figure 2.6: A visual representation of the instantiation phase

Figure 2.6 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.
4. Note that, in this phase, the exports map keeps track of the exported names only; 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 2.7:

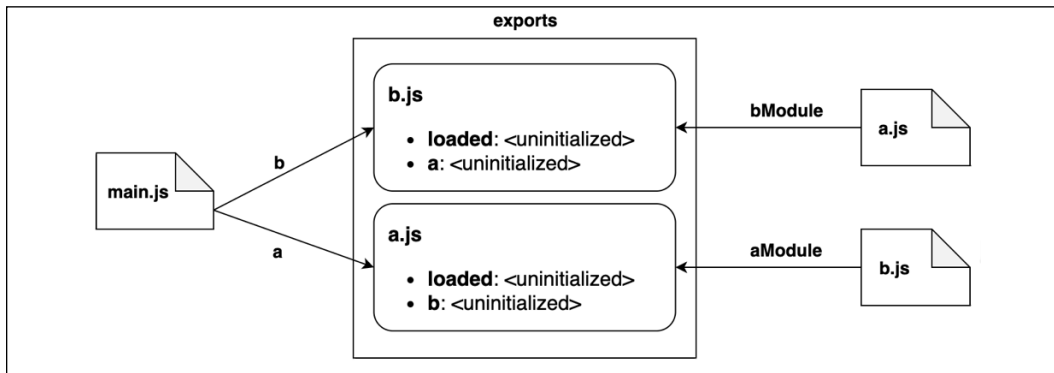


Figure 2.7: Linking exports with imports across modules

We can describe what we see in Figure 2.7 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`.
4. 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, `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:

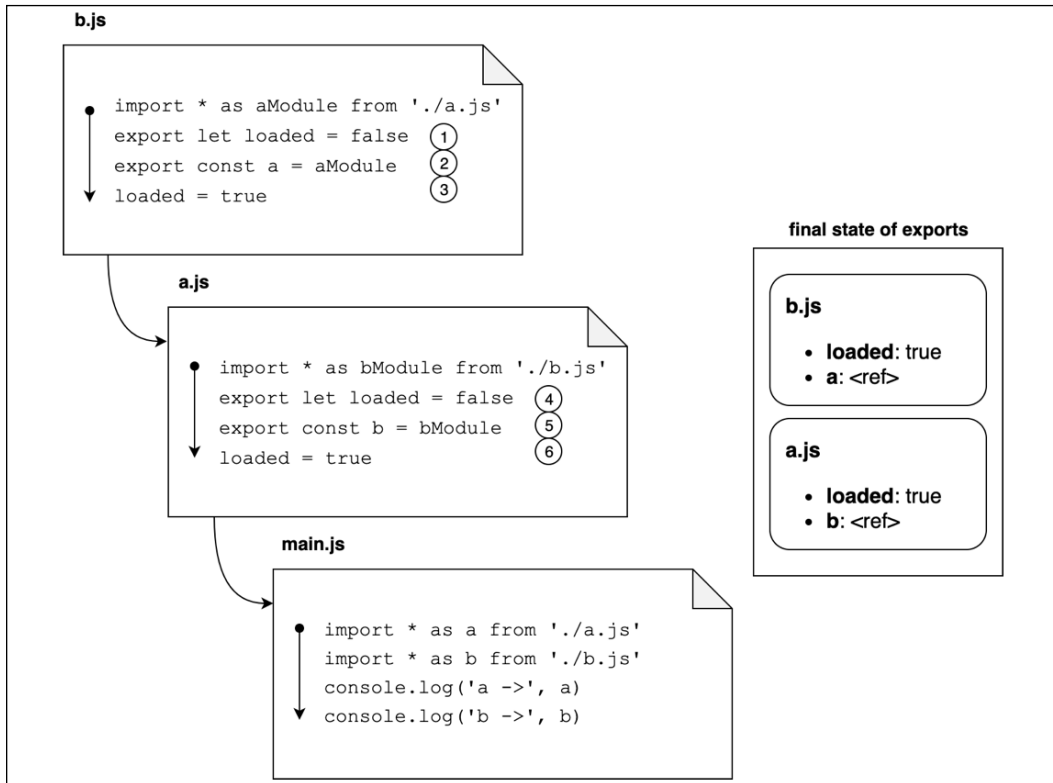


Figure 2.8: A visual representation of the evaluation phase

Following along from the diagram in *Figure 2.8*, 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, here the exported property `a` gets evaluated. This time, it will be evaluated to 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.

Modifying other modules

We saw that entities imported through ES modules are *read-only live bindings*, and therefore we cannot reassign them from an external module.

There's a caveat, though. It is true that we can't change the bindings of the default export or named exports of an existing module from another module, but, if one of these bindings is an object, we can still mutate the object itself by reassigning some of the object properties.

This caveat can give us enough freedom to alter the behavior of other modules. To demonstrate this idea, let's write a module that can alter the behavior of the core `fs` module so that it prevents the module from accessing the filesystem and returns mocked data instead. This kind of module is something that could be useful while writing tests for a component that relies on the filesystem:

```
// mock-read-file.js
import fs from 'fs' // (1)

const originalReadFile = fs.readFile // (2)
let mockedResponse = null

function mockedReadFile (path, cb) { // (3)
  setImmediate(() => {
    cb(null, mockedResponse)
  })
}

export function mockEnable (respondWith) { // (4)
  mockedResponse = respondWith
  fs.readFile = mockedReadFile
}

export function mockDisable () { // (5)
  fs.readFile = originalReadFile
}
```

Let's review the preceding code:

1. The first thing we do is import the default export of the `fs` module. We will get back to this in a second, for now, just keep in mind that the default export of the `fs` module is an object that contains a collection of functions that allows us to interact with the filesystem.
2. We want to replace the `readFile()` function with a mock implementation. Before doing that, we save a reference to the original implementation. We also declare a `mockedResponse` value that we will be using later.
3. The function `mockedReadFile()` is the actual mocked implementation that we want to use to replace the original implementation. This function invokes the callback with the current value of `mockedResponse`. Note that this is a simplified implementation; the real function accepts an optional `options` argument before the callback argument and is able to handle different types of encoding.
4. The exported `mockEnable()` function can be used to activate the mocked functionality. The original implementation will be swapped with the mocked one. The mocked implementation will return the same value passed here through the `respondWith` argument.
5. Finally, the exported `mockDisable()` function can be used to restore the original implementation of the `fs.readFile()` function.

Now let's see a simple example that uses this module:

```
// main.js
import fs from 'fs' // (1)
import { mockEnable, mockDisable } from './mock-read-file.js'

mockEnable(Buffer.from('Hello World')) // (2)

fs.readFile('fake-path', (err, data) => { // (3)
  if (err) {
    console.error(err)
    process.exit(1)
  }
  console.log(data.toString()) // 'Hello World'
})

mockDisable()
```

Let's discuss step by step what happens in this example:

1. The first thing that we do is import the default export of the `fs` module. Again, note that we are importing specifically the default export exactly as we did in our `mock-read-file.js` module, but more on this later.
2. Here we enable the mock functionality. We want, for every file read, to simulate that the file contains the string "Hello World."
3. Finally, we read a file using a fake path. This code will print "Hello World" as it will be using the mocked version of the `readFile()` function. Note that, after calling this function, we restore the original implementation by calling `mockDisable()`.

This approach works, but it is very fragile. In fact, there are a number of ways in which this may not work.

On the `mock-read-file.js` side, we could have tried the two following imports for the `fs` module:

```
import * as fs from 'fs' // then use fs.readFile
```

or

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

Both of them are valid imports because the `fs` module exports all the filesystem functions as named exports (other than a default export which is an object with the same collection of functions as attributes).

There are certain issues with the preceding two `import` statements:

- We would get a read-only live binding into the `readFile()` function, and therefore, we would be unable to mutate it from an external module. If we try these approaches, we will get an error when trying to reassign `readFile()`.
- Another issue is on the consumer side within our `main.js`, where we could use these two alternative import styles as well. In this case, we won't end up using the mocked functionality, and therefore the code will trigger an error while trying to read a nonexistent file.

The reason why using one of the two `import` statements mentioned above would not work is because our mocking utility is altering only the copy of the `readFile()` function that is registered inside the object exported as the default export, but not the one available as a named export at the top level of the module.

This particular example shows us how monkey patching could be much more complicated and unreliable in the context of ESM. For this reason, testing frameworks such as Jest (nodejsdp.link/jest) provide special functionalities to be able to mock ES modules more reliably (nodejsdp.link/jest-mock).



Another approach that can be used to mock modules is to rely on the hooks available in a special Node.js core module called `module` (nodejsdp.link/module-doc). One simple library that takes advantage of this module is `mocku` (nodejsdp.link/mocku). Check out its source code if you are curious.

We could also use the `syncBuiltinESMExports()` function from the `module` package. When this function is invoked, the value of the properties in the default exports object gets mapped again into the equivalent named exports, effectively allowing us to propagate any external change applied to the module functionality even to named exports:

```
import fs, { readFileSync } from 'fs'
import { syncBuiltinESMExports } from 'module'

fs.readFileSync = () => Buffer.from('Hello, ESM')
syncBuiltinESMExports()

console.log(fs.readFileSync === readFileSync) // true
```

We could use this to make our small filesystem mocking utility a little bit more flexible by invoking the `syncBuiltinESMExports()` function after we enable the mock or after we restore the original functionality.



Note that `syncBuiltinESMExports()` works only for built-in Node.js modules like the `fs` module in our example.

This concludes our exploration of ESM. At this point, we should be able to appreciate how ESM works, how it loads modules, and how it deals with cyclic dependencies. To close this chapter, we are now ready to discuss some key differences and some interesting interoperability techniques between CommonJS and ECMAScript modules.

ESM and CommonJS differences and interoperability

We already mentioned several important differences between ESM and CommonJS, such as having to explicitly specify file extensions in imports with ESM, while file extensions are totally optional with the CommonJS `require` function.

Let's close this chapter by discussing some other important differences between ESM and CommonJS and how the two module systems can work together when necessary.

ESM runs in strict mode

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, but 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 (<https://nodejsdp.link/strict-mode>).

Missing references in ESM

In ESM, some important CommonJS references are not defined. These include `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
```

We already discussed at length the meaning of `exports` and `module` in CommonJS; `__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 ESM, it is possible to get a reference to the current file URL by using the special object `import.meta`. Specifically, `import.meta.url` is a reference to the current module file in a format similar to `file:///path/to/current_module.js`. This value can be used to reconstruct `__filename` and `__dirname` in the form of absolute paths:

```
import { fileURLToPath } from 'url'
import { dirname } from 'path'
const __filename = fileURLToPath(import.meta.url)
const __dirname = dirname(__filename)
```

It is also possible to recreate the `require()` function as follows:

```
import { createRequire } from 'module'
const require = createRequire(import.meta.url)
```

Now we can use `require()` to import functionality coming from CommonJS modules in the context of ES modules.

Another interesting difference 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`:

```
// this.js - ESM
console.log(this) // undefined
```

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

Interoperability

We discussed in the previous section how to import CommonJS modules in ESM by using the `module.createRequire` function. It is also possible to import CommonJS modules from ESM by using the standard `import` syntax. This is only limited to default exports, though:

```
import packageMain from 'commonjs-package' // Works
import { method } from 'commonjs-package' // Errors
```

Unfortunately, it is not possible to import ES modules from CommonJS modules.

Also, ESM cannot import JSON files directly as modules, a feature that is used quite frequently with CommonJS. The following `import` statement will fail:

```
import data from './data.json'
```

It will produce a `TypeError` (Unknown file extension: `.json`).

To overcome this limitation, we can use again the `module.createRequire` utility:

```
import { createRequire } from 'module'
const require = createRequire(import.meta.url)
const data = require('./data.json')
console.log(data)
```

There is ongoing work to support JSON modules natively even in ESM, so we may not need to rely on `createRequire()` in the near future for this functionality.

Summary

In this chapter, we explored in depth what modules are, why they are useful, and why we need a module system. We also learned about the history of modules in JavaScript and about the two module systems available today in Node.js, namely CommonJS and ESM. We also explored some common patterns that are useful when creating modules or when using third-party modules.

You should now be comfortable with understanding and writing code that takes advantage of the features of both CommonJS and ESM.

In the rest of the book, we will rely mostly on ES modules, but you should now be equipped to be flexible with your choices and be able to deal with CommonJS effectively if necessary.

In the next chapter, we will start to explore the idea of asynchronous programming with JavaScript, and we will examine callbacks, events, and their patterns in depth.

3

Callbacks and Events

In synchronous programming, we conceptualize code as a series of consecutive computing steps that solve a specific problem. Every operation is blocking, which means that only when an operation is completed, it is possible to execute the next one. This approach makes the code very easy to read, understand, and debug.

On the other side, in asynchronous programming, some operations, such as reading from a file or performing a network request, are launched and then executed "in the background." When we invoke an asynchronous operation, the instruction that follows is executed immediately, even if the previous asynchronous operation has not finished yet. In this scenario, we need a way to get notified when an asynchronous operation completes, and then continue the execution flow using the results from the operation. The most basic mechanism to get notified about the completion of an asynchronous operation in Node.js is the **callback**, which is nothing more than a function invoked by the runtime with the result of an asynchronous operation.

The callback is the most basic building block on which all other asynchronous mechanisms are based. In fact, without callbacks, we wouldn't have promises, and therefore not even `async/await`; we also wouldn't have streams or events. This is why it's important to know how callbacks work.

In this chapter, you will learn more about the Node.js Callback pattern and understand what it means, in practice, to write asynchronous code. We will make our way through conventions, patterns, and pitfalls, and by the end of this chapter, you will have mastered the basics of the Callback pattern.

You will also learn about the Observer pattern, which can be considered a close relative of the Callback pattern. The Observer pattern—embodied by the `EventEmitter`—uses callbacks to deal with multiple heterogeneous events and is one of the most extensively 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

The Callback pattern

Callbacks are the materialization of the handlers of the Reactor pattern (introduced in the previous chapter). They are one of those imprints that give Node.js its distinctive programming style.

Callbacks are functions that are invoked to propagate the result of an operation, and this is exactly what we need when dealing with asynchronous operations. In the asynchronous world, they replace the use of the `return` instruction, which, in turn, always executes synchronously. JavaScript is the ideal language for callbacks because functions are first-class objects and can be easily assigned to variables, passed as arguments, returned from another function invocation, or stored in data structures. Another ideal construct for implementing callbacks is **closures**. With closures, we can reference the environment in which a function was created; this way, we can always maintain the context in which the asynchronous operation was requested, no matter when or where its callback is 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 will analyze this particular style of programming, which uses callbacks instead of `return` instructions.

The continuation-passing style

In JavaScript, a callback is a function that is passed as an argument to another function and is invoked with the result when the operation 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 this concept, let's take a look at a simple synchronous function:

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

If you are wondering, there is nothing special going on here. The result is passed back to the caller using the `return` instruction. This is also called **direct style**, and it represents the most common way of returning a result in synchronous programming.

The equivalent CPS of the preceding function would be as follows:

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

The `addCps()` function is a synchronous CPS function. It's synchronous because it will complete its execution only when the callback completes its execution too. The following code demonstrates this statement:

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

Since `addCps()` is synchronous, the previous code will trivially print the following:

```
before  
Result: 3  
after
```

Now, let's see how asynchronous CPS works.

Asynchronous CPS

Let's consider a case where the `addCps()` function is asynchronous:

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

In the previous code, we used `setTimeout()` to simulate an asynchronous invocation of the callback. `setTimeout()` adds a task to the event queue that is executed after the given number of milliseconds. This is clearly an asynchronous operation. Now, let's try to use `addAsync()` and see how the order of the 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 shows how this works:

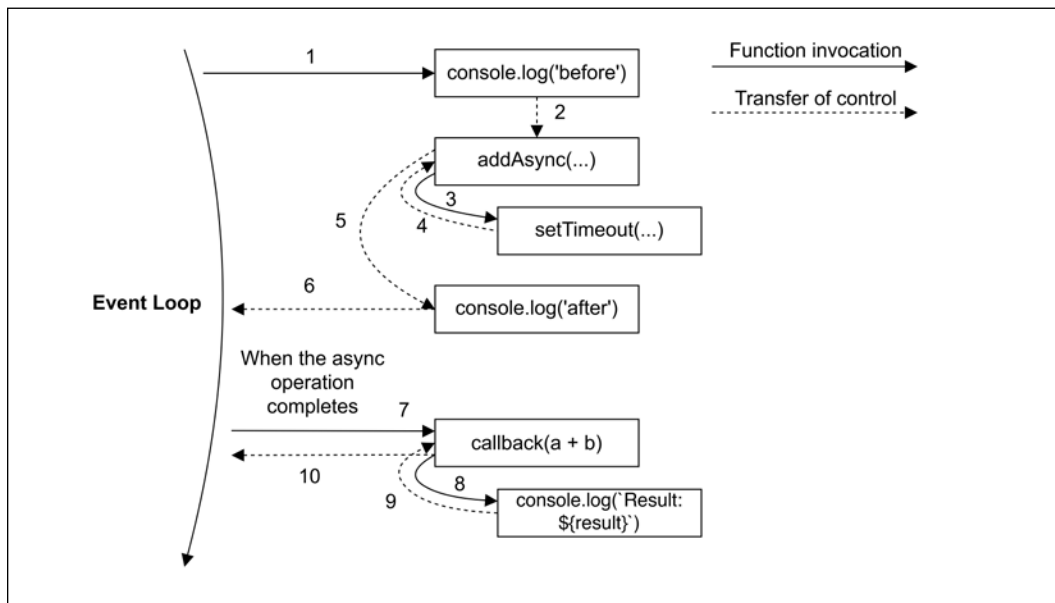


Figure 3.1: Control flow of an asynchronous function's invocation

When the asynchronous operation completes, the execution is then resumed, starting from the callback provided to the asynchronous function that caused the unwinding. The execution starts from the event loop, so it has a fresh stack. This is where JavaScript comes in really handy. Thanks to closures, it is trivial to maintain the context of the caller of the asynchronous function, even if the callback is invoked at a different point in time and from a different location.

To sum this up, a synchronous function blocks until it completes its operations. An asynchronous function returns immediately, and its result is passed to a handler (in our case, a callback) at a later cycle of the event loop.

Non-CPS callbacks

There are several circumstances in which the presence of a callback argument might make us think that a function is asynchronous or is using a CPS. That's not always true. Let's take, for example, the `map()` method of an Array object:

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

Clearly, the callback is used just to iterate over the elements of the array, and not to pass the result of the operation. In fact, the result is returned synchronously using a direct style. There's no syntactic difference between non-CPS callbacks and CPS ones. Therefore, the intent of a callback should be clearly stated in the documentation of the API.

In the next section, we are going to discuss one of the most important pitfalls of callbacks that every Node.js developer should 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.

An unpredictable function

One of the most dangerous situations is to have an API that behaves synchronously under certain conditions and asynchronously under others. Let's take the following code as an example:

```
import { readFile } from '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)
    })
  }
}
```

The preceding function uses the cache map to store the results of different file read operations. Bear in mind that this is just an example; it does not have error management, and the caching logic itself is suboptimal (in *Chapter 11, Advanced Recipes*, you'll learn how to handle asynchronous caching properly). But besides all this, the preceding function is dangerous because it behaves asynchronously until the file is read for the first time and the cache is set, but it is synchronous for all the subsequent requests once the file's content is already in the cache.

Unleashing Zalgo

Now, let's discuss how the use of an unpredictable function, such as the one that we just defined, can easily break an application. Consider the following code:

```
function createFileReader (filename) {
  const listeners = []
  inconsistentRead(filename, value => {
    listeners.forEach(listener => 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:

```

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:

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

As you can see, the callback of the second reader is never invoked. Let's see why:

- During the creation of `reader1`, our `inconsistentRead()` function behaves asynchronously because there is no cached result available. 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 the requested file 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.

The bug that you've just seen 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. This can definitely be considered a nasty defect.

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

Zalgo is an internet legend about an ominous entity believed to cause insanity, death, and the destruction of the world. If you're not familiar with Zalgo, you are invited to find out what it is.



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

Using synchronous APIs

The lesson to learn from the unleashing Zalgo example is that it is imperative for an API to clearly define its nature: either synchronous or asynchronous.

One possible fix for our `inconsistentRead()` function is to make it completely synchronous. This is possible because Node.js provides a set of synchronous direct style APIs for most basic I/O operations. For example, we can use the `fs.readFileSync()` function in place of its asynchronous counterpart. The code would become as follows:

```
import { readFileSync } from 'fs'

const cache = new Map()

function consistentReadSync (filename) {
  if (cache.has(filename)) {
    return cache.get(filename)
  } else {
    const data = readFileSync(filename, 'utf8')
    cache.set(filename, data)
    return data
  }
}
```

You can see that the entire function was also converted into direct style. There is no reason for a function to have a CPS if it is synchronous. In fact, it is always best practice to implement a synchronous API using a direct style. This will eliminate any confusion around its nature and will also be more efficient from a performance perspective.

**Pattern**

Always choose a direct style for purely synchronous functions.

Bear in mind that changing an API from CPS to a direct style, or from asynchronous to synchronous or vice versa, might also require a change to the style of all the code using it. For example, in our case, we will have to totally change the interface of our `createFileReader()` API and adapt it so that it always works synchronously.

Also, 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 will break the Node.js concurrency model, slowing down the whole application. You will see later in this book what this really means for our applications.

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 strongly discouraged in many circumstances, but in some situations, this might be the easiest and most efficient solution. Always evaluate your specific use case in order to choose the right alternative. As an example, it makes perfect sense to use a synchronous blocking API to load a configuration file while bootstrapping an application.

**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 our `inconsistentRead()` function, as follows:

```
import { readFile } from '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) => {
      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()`.

Another API for deferring the execution of code is `setImmediate()`. While its purpose is very similar to that of `process.nextTick()`, its semantics are quite different. 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. Since `process.nextTick()` runs before any already scheduled I/O, it will be executed faster, but under certain circumstances, it might also delay the running of any I/O callback indefinitely (also known as **I/O starvation**), such as in the presence of a recursive invocation. This can never happen with `setImmediate()`.

Using `setTimeout(callback, 0)` has a behavior comparable to that of `setImmediate()`, but in typical circumstances, callbacks scheduled with `setImmediate()` are executed faster than those scheduled with `setTimeout(callback, 0)`. To see why, we have to consider that the event loop executes all the callbacks in different phases; for the type of events we are considering, we have timers (`setTimeout()`) that are executed before I/O callbacks, which are, in turn, executed before `setImmediate()` callbacks. This means that if we queue a task with `setImmediate()` in a `setTimeout()` callback, in an I/O callback, or in a microtask queued after these two phases, then the callback will be executed in a phase that comes right after the phase we are currently in. `setTimeout()` callbacks have to wait for the next cycle of the event loop.

You will better appreciate the difference between these APIs when we analyze the use of deferred invocation for running synchronous CPU-bound tasks later in this book.

Next, we are going to explore the conventions used to define callbacks in Node.js.

Node.js callback conventions

In Node.js, CPS APIs and callbacks follow a set of specific conventions. These conventions apply to the Node.js core API, but they are also followed by the vast majority of the userland modules and applications. So, it's very important that you understand them and make sure that you comply whenever you need to design an asynchronous API that makes use of callbacks.

The callback comes last

In all core Node.js functions, the standard convention is that when a function accepts a callback as input, this has to 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. 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 take into account is that the error must always be of type `Error`. This means that simple strings or numbers should never be passed as error objects.

Propagating errors

Propagating errors 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.

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 'fs'  
  
function readJSON (filename, callback) {
```

```
readFile(filename, 'utf8', (err, data) => {  
  let parsed  
  if (err) {  
    // 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 propagate the error received by the `readFile()` operation. We do not throw it or return it; instead, we just use the callback as if it were any other result. Also, notice how we use the `try...catch` statement to catch any error thrown by `JSON.parse()`, which is a synchronous function and therefore uses the traditional throw instruction to propagate errors to the caller. Lastly, 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.

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 had forgotten to surround `JSON.parse()` with a `try...catch` statement in the `readJSON()` function we defined previously. 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 demonstrate this, 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()`. If we try to parse an invalid JSON file with the following code:

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

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/08-uncaught-errors/index.
js:8:25
    at FSReqCallback.readFileAfterClose [as oncomplete] (internal/fs/
read_file_context.js:61:3)
```

Now, if you look at the preceding stack trace, 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 work, 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 this happens, 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.



We'll discuss supervisors 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 actually 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:

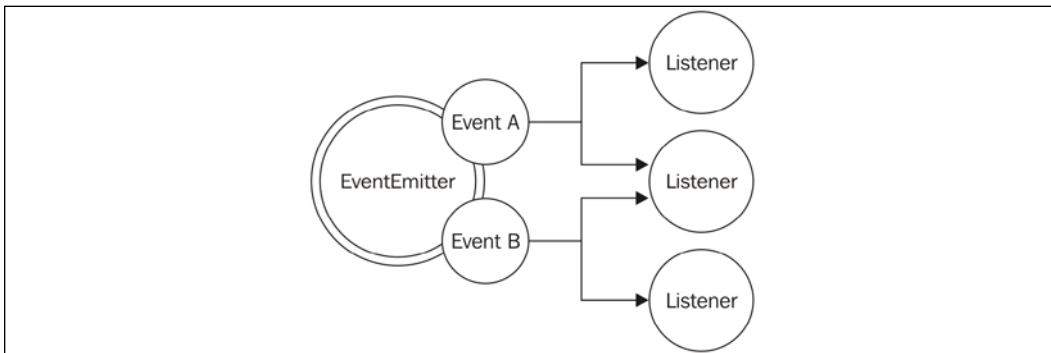


Figure 3.2: Listeners receiving events from an EventEmitter

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

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

The essential methods of the `EventEmitter` are as follows:

- `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.

All the preceding methods will return the `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 is matched in a list of files:

```
import { EventEmitter } from 'events'
import { readFile } from '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) {
        match.forEach(elem => 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+/g
)
.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.

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 were doing in the `findRegex()` function that we defined earlier.



The `EventEmitter` 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 'events'
import { readFile } from '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) {
          match.forEach(elem => this.emit('found', file, elem))
        }
      })
    }
    return this
  }
}
```

The `FindRegex` class that we just defined extends `EventEmitter` to become a fully fledged observable class. Always remember to use `super()` in the constructor to initialize the `EventEmitter` internals.

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 pretty 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*.

EventEmitter and memory leaks

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).

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. 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. To prevent such a 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`.



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 advised 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 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) {
      match.forEach(elem => 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 NodeJS"
```

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

A common dilemma when defining an asynchronous API is deciding whether to use an `EventEmitter` or simply accept a callback. The general differentiating rule is semantic: callbacks should be used when a result must be returned in an asynchronous way, while events should be used when there is a need to communicate that something has happened.

But besides this simple principle, a lot of confusion is generated from the fact that the two paradigms are, most of the time, equivalent and allow us to achieve the same results. Consider the following code as an example:

```
import { EventEmitter } from 'events'

function helloEvents () {
  const eventEmitter = new EventEmitter()
  setTimeout(() => eventEmitter.emit('complete', 'hello world'), 100)
  return eventEmitter
}
```

```
function helloCallback (cb) {  
  setTimeout(() => cb(null, 'hello world'), 100)  
}  
  
helloEvents().on('complete', message => console.log(message))  
helloCallback((err, message) => console.log(message))
```

The two functions `helloEvents()` and `helloCallback()` can be considered equivalent in terms of functionality. The first communicates the completion of the timeout using an event, while the second uses a callback. But what really differentiates them is the readability, the semantics, and the amount of code that is required for them to be implemented or used.

While a deterministic set of rules for you to choose between one style or the other can't be given, here are some hints to help you make a decision on which method to use:

- Callbacks have some limitations when it comes to supporting different types of events. In fact, we can still differentiate between multiple events by passing the type as an argument of the callback, or by accepting several callbacks, one for each supported event. However, this can't exactly be considered an elegant API. In this situation, the `EventEmitter` can give a better interface and leaner code.
- The `EventEmitter` should be used when the same event can occur multiple times, or may not occur at all. A callback, in fact, is expected to be invoked exactly once, whether the operation is successful or not. Having a possibly repeating circumstance should make us think again about the semantic nature of the occurrence, which is more similar to an event that has to be communicated, rather than a result to be returned.
- An API that uses callbacks can notify only one particular callback, while using an `EventEmitter` allows us to register multiple listeners for the same event.

Combining callbacks and events

There are some particular circumstances where the `EventEmitter` can be used in conjunction with a callback. This pattern is extremely powerful as it allows us to pass a result asynchronously using a traditional callback, and at the same time return an `EventEmitter`, which can be used to provide a more detailed account on the status of an asynchronous process.

One example of this pattern is offered by the `glob` package (nodejsdp.link/npm-glob), a library that performs glob-style file searches. The main entry point of the module is the function it exports, which has the following signature:

```
const EventEmitter = glob(pattern, [options], callback)
```

The function takes a `pattern` as the first argument, a set of `options`, and a `callback` that is invoked with the list of all the files matching the provided pattern. At the same time, the function returns an `EventEmitter`, which provides a more fine-grained report about the state of the search process. For example, it is possible to be notified in real time when a match occurs by listening to the `match` event, to obtain the list of all the matched files with the `end` event, or to know whether the process was manually aborted by listening to the `abort` event. The following code shows what this looks like in practice:

```
import glob from 'glob'

glob('data/*.txt',
  (err, files) => {
    if (err) {
      return console.error(err)
    }
    console.log(`All files found: ${JSON.stringify(files)}`)
  })
.on('match', match => console.log(`Match found: ${match}`))
```

Combining an `EventEmitter` with traditional callbacks is an elegant way to offer two different approaches to the same API. One approach is usually meant to be simpler and more immediate to use, while the other is targeted at more advanced scenarios.



The `EventEmitter` can also be combined with other asynchronous mechanisms such as promises (which we will look at in *Chapter 5, Asynchronous Control Flow Patterns with Promises and Async/Await*). In this case, just return an object (or array) containing both the promise and the `EventEmitter`. This object 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. You 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 you 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, you 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.

4

Asynchronous Control Flow Patterns with Callbacks

Moving from a synchronous programming style to a platform such as Node.js, where **continuation-passing style (CPS)** and asynchronous APIs are the norm, can be frustrating. Asynchronous code can make it hard to predict the order in which statements are executed. Simple problems such as iterating over a set of files, executing tasks in sequence, or waiting for a set of operations to complete require the developer to take on new approaches and techniques just to avoid ending up writing inefficient and unreadable code. When using callbacks to deal with asynchronous control flow, the most common mistake is to fall into the trap of callback hell and see the code growing horizontally, rather than vertically, with a nesting that makes even simple routines hard to read and maintain.

In this chapter, you will see how it's actually possible to tame callbacks and write clean, manageable asynchronous code by using some discipline and with the aid of some patterns. Knowing how to properly deal with callbacks will pave the way for adopting modern approaches such as promises and `async/await`.

In short, in this chapter, you will learn about:

- The challenges of asynchronous programming.
- Avoiding callback hell and other callback best practices.
- Common asynchronous patterns such as sequential execution, sequential iteration, parallel execution, and limited parallel execution.

The difficulties of asynchronous programming

Losing control of asynchronous code in JavaScript is undoubtedly easy. Closures and in-place definitions of anonymous functions allow for a smooth programming experience that doesn't require the developer to jump to other points in the codebase. This is perfectly in line with the **KISS principle (Keep It Simple, Stupid)**; it's simple, it keeps the code flowing, and we get it working in less time. Unfortunately, sacrificing qualities such as modularity, reusability, and maintainability will, sooner or later, lead to the uncontrolled proliferation of callback nesting, functions growing in size, and poor code organization. Most of the time, creating callbacks as in-place functions is not strictly required, so it's more a matter of discipline than a problem related to asynchronous programming. Recognizing that our code is becoming unwieldy or, even better, knowing in advance that it might become unwieldy and then acting accordingly with the most adequate solution, is what differentiates a novice from an expert.

Creating a simple web spider

To explain this problem, 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 a file. In the code presented in this chapter, we are going to use a couple of npm dependencies:

- `superagent`: A library to streamline HTTP calls ([nodejshq.link/superagent](https://nodejshq.github.io/superagent/))
- `mkdirp`: A small utility to create directories recursively ([nodejshq.link/mkdirp](https://nodejshq.github.io/mkdirp/))

Also, 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 [nodejshq.link/repo](https://nodejshq.github.io/).

The core functionality of our application is contained inside a module named `spider.js`. Let's see how it looks. To start with, let's load all the dependencies that we are going to use:

```
import fs from 'fs'
import path from 'path'
import superagent from 'superagent'
import mkdirp from 'mkdirp'
import { urlToFilename } from './utils.js'
```

Next, let's create a new function named `spider()`, which takes in the URL to download and a callback function that will be invoked when the download process completes:

```
export function spider (url, cb) {
  const filename = urlToFilename(url)
  fs.access(filename, err => {                                     // (1)
    if (err && err.code === 'ENOENT') {
      console.log(`Downloading ${url} into ${filename}`)
      superagent.get(url).end((err, res) => {                     // (2)
        if (err) {
          cb(err)
        } else {
          mkdirp(path.dirname(filename), err => {                 // (3)
            if (err) {
              cb(err)
            } else {
              fs.writeFile(filename, res.text, err => {           // (4)
                if (err) {
                  cb(err)
                } else {
                  cb(null, filename, true)
                }
              })
            }
          })
        }
      })
    }
  })
}
} else {
  cb(null, filename, false)
}
```

There is a lot going on here, so let's discuss in more detail what happens in every step:

1. The code checks whether the URL was already downloaded by verifying that the corresponding file was not already created. If `err` is defined and has type `ENOENT`, then the file does not exist and it's safe to create it:

```
fs.access(filename, err => ...
```

2. If the file is not found, the URL is downloaded using the following line of code:

```
superagent.get(url).end((err, res) => ...
```

3. Then, we make sure that the directory that will contain the file exists:

```
mkdirp(path.dirname(filename), err => ...
```

4. Finally, we write the body of the HTTP response to the filesystem:

```
fs.writeFile(filename, res.text, err => ...
```

To complete our web spider application, we just need to invoke the `spider()` function by providing a URL as an input (in our case, we read it from the command-line arguments). The `spider()` function is exported from the file we defined previously. Let's now create a new file called `spider-cli.js` that can be directly invoked from the command line:

```
import { spider } from './spider.js'

spider(process.argv[2], (err, filename, downloaded) => {
  if (err) {
    console.error(err)
  } else if (downloaded) {
    console.log(`Completed the download of "${filename}"`)
  } else {
    console.log(`"${filename}" was already downloaded`)
  }
})
```

Now, we are ready to try our web spider application, but first, make sure you have the `utils.js` module and the `package.json` file containing the full list of dependencies in your project directory. Then, install all the dependencies by running the following command:

```
npm install
```

Now, let's execute the `spider-cli.js` module to download the contents of a web page with a command like this:

```
node spider-cli.js http://www.example.com
```



Our 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 to be downloaded, as this is just a simple example to demonstrate how asynchronous programming works.

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

Looking at the `spider()` function we defined earlier, you will likely notice that even though the algorithm we implemented is really straightforward, the resulting code has several levels of indentation and is very hard to read. Implementing a similar function with a direct style blocking API would be straightforward, and most likely, the code would be much more readable. However, using asynchronous CPS is another story, and making bad use of in-place callback definitions can lead to incredibly bad code.

The situation where the abundance of closures and in-place callback definitions transforms the code into an unreadable and unmanageable blob is known as **callback hell**. It's one of the most widely recognized and severe anti-patterns in Node.js and JavaScript in general. The typical structure of code affected by this problem looks as follows:

```
asyncFoo(err => {  
  asyncBar(err => {  
    asyncFooBar(err => {  
      //...  
    })  
  })  
})
```

You can see how code written in this way assumes the shape of a pyramid due to deep nesting, and that's why it is also colloquially known as the **pyramid of doom**.

The most evident problem with code such as the preceding snippet is its poor readability. Due to the nesting being so deep, it's almost impossible to keep track of where a function ends and where another one begins.

Another issue is caused by the overlapping of the variable names used in each scope. Often, we have to use similar or even identical names to describe the content of a variable. The best example is the error argument received by each callback. Some people often try to use variations of the same name to differentiate the object in each scope, for example, `err`, `error`, `err1`, `err2`, and so on. Others prefer to just shadow the variable defined in the upper scope by always using the same name, for example, `err`. Both alternatives are far from perfect, and cause confusion and increase the probability of introducing defects.

Also, we have to keep in mind that closures come at a small price in terms of performance and memory consumption. In addition, they can create memory leaks that are not very easy to identify. In fact, we shouldn't forget that any context referenced by an active closure is retained from garbage collection.



For a great introduction to how closures work in V8, you can refer to the following blog post by Vyacheslav Egorov, a software engineer at Google working on V8, which you can read at nodejsdp.link/v8-closures.

If you look at our `spider()` function, you will notice that it clearly represents a callback hell situation and has all the problems just described. That's exactly what we are going to fix with the patterns and techniques that are covered in the following sections of this chapter.

Callback best practices and control flow patterns

Now that you have met your first example of callback hell, you know what you should definitely avoid; however, that's not the only concern when writing asynchronous code. In fact, there are several situations where controlling the flow of a set of asynchronous tasks requires the use of specific patterns and techniques, especially if we are only using plain JavaScript without the aid of any external library. For example, iterating over a collection by applying an asynchronous operation in sequence is not as easy as invoking `forEach()` over an array; it actually requires a technique similar to recursion.

In this section, you will learn not only about how to avoid callback hell, but also how to implement some of the most common control flow patterns, using only simple and plain JavaScript.

Callback discipline

When writing asynchronous code, the first rule to keep in mind is to not abuse in-place function definitions when defining callbacks. It can be tempting to do so, because it does not require any additional thinking for problems such as modularization and reusability; however, you have seen how this can have more disadvantages than advantages. Most of the time, fixing the callback hell problem does not require any libraries, fancy techniques, or changes of paradigm; you just need some common sense.

These are some basic principles that can help us keep the nesting level low and improve the organization of our code in general:

- Exit as soon as possible. Use `return`, `continue`, or `break`, depending on the context, to immediately exit the current statement instead of writing (and nesting) complete `if...else` statements. This will help to keep our code shallow.
- Create named functions for callbacks, keeping them out of closures and passing intermediate results as arguments. Naming our functions will also make them look better in stack traces.
- Modularize the code. Split the code into smaller, reusable functions whenever possible.

Now, let's put these principles into practice.

Applying the callback discipline

To demonstrate the power of the ideas mentioned in the previous section, let's apply them to fix the callback hell in our web spider application.

For the first step, we can refactor our error-checking pattern by removing the `else` statement. This is made possible by returning from the function immediately after we receive an error. So, instead of having code such as the following:

```
if (err) {  
  cb(err)  
} else {  
  // code to execute when there are no errors  
}
```


We can improve the organization of our code by writing the following instead:

```
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) {  
  callback(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 callback(...)
```

Otherwise, we'd have to write slightly more verbose code, such as the following:

```
callback(...)  
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 can be easily factored out into a separate function, as follows:

```
function saveFile (filename, contents, cb) {  
  mkdirp(path.dirname(filename), err => {  
    if (err) {
```

```

    return cb(err)
  }
  fs.writeFile(filename, contents, 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}`)
  superagent.get(url).end((err, res) => {
    if (err) {
      return cb(err)
    }
    saveFile(filename, res.text, err => {
      if (err) {
        return cb(err)
      }
      console.log(`Downloaded and saved: ${url}`)
      cb(null, res.text)
    })
  })
}

```

For the last step, we modify the `spider()` function, which, thanks to our changes, will now look like the following:

```

export function spider (url, cb) {
  const filename = urlToFilename(url)
  fs.access(filename, err => {
    if (!err || err.code !== 'ENOENT') { // (1)
      return cb(null, filename, false)
    }
    download(url, filename, err => {
      if (err) {
        return cb(err)
      }
      cb(null, filename, true)
    })
  })
}

```

The functionality and the interface of the `spider()` function remained exactly the same; what changed was the way the code was organized. One important detail to notice (1) is that we inverted the check for the file's existence so that we could apply the *early return principle* discussed previously.

By applying the early return principle and the other callback discipline principles, we were able to drastically reduce the nesting of our code and, at the same time, increase its reusability and testability. In fact, we could think about exporting both `saveFile()` and `download()` so that we could reuse them in other modules. This would also allow us to test their functionality as independent units.

The refactoring we carried out in this section clearly demonstrates that most of the time, all we need is some discipline to make sure we do not abuse closures and anonymous functions. It works brilliantly, requires minimal effort, and it doesn't require external libraries.

Now that you know how to write clean asynchronous code using callbacks, we are ready to explore some of the most common asynchronous patterns, such as sequential and parallel execution.

Sequential execution

In this section, we will look at asynchronous control flow patterns and start by analyzing the sequential execution flow.

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:

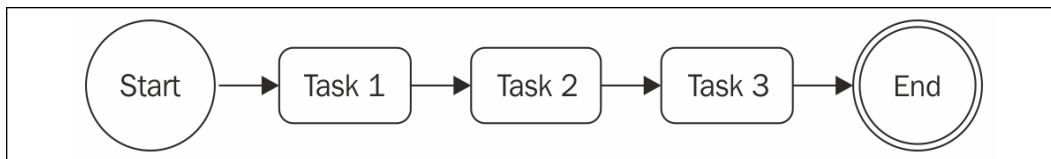


Figure 4.1: An example of sequential execution flow with three tasks

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, despite being trivial when implemented using a direct style blocking API, is usually the main cause of the callback hell problem when using asynchronous CPS.

Executing a known set of tasks in sequence

We already looked at a sequential execution flow while implementing the `spider()` function in the previous section. By applying some simple rules, we were able to organize a set of known tasks in a sequential execution flow. Taking that code as a guideline, we can now generalize the solution with the following pattern:

```
function task1 (cb) {
  asyncOperation(() => {
    task2(cb)
  })
}

function task2 (cb) {
  asyncOperation(() => {
    task3(cb)
  })
}

function task3 (cb) {
  asyncOperation(() => {
    cb() // finally executes the callback
  })
}

task1(() => {
  // executed when task1, task2 and task3 are completed
  console.log('tasks 1, 2 and 3 executed')
})
```

The preceding pattern shows how each task invokes the next upon completion of a generic asynchronous operation. The pattern puts the emphasis on the modularization of tasks, showing how closures are not always necessary to handle asynchronous code.

Sequential iteration

The pattern described in the previous section works perfectly if we know in advance what and how many tasks are to be executed. This allows us to hardcode the invocation of the next task in the sequence, but what happens if we want to execute an asynchronous operation for each item in a collection? In cases such as this, we can't hardcode the task sequence anymore; instead, we have to build it dynamically.

Web spider version 2

To show an example of sequential iteration, let's introduce a new feature to the web spider application. We now want to download all the links contained in a web page recursively. To do that, we are going to extract all the links from the page and then trigger our web spider on each recursively and in sequence.

The first step is modifying our `spider()` function so that it triggers a recursive download of all the links of a page by using a function named `spiderLinks()`, which we are going to create shortly.

Also, instead of checking whether the file already exists, we will try to read it and start spidering its links. This way, we will be able to resume interrupted downloads. As a final change, we need to make sure we propagate a new parameter, `nesting`, which will help us to limit the recursion depth. The code is as follows:

```
export function spider (url, nesting, cb) {
  const filename = urlToFilename(url)
  fs.readFile(filename, 'utf8', (err, fileContent) => {
    if (err) {
      if (err.code !== 'ENOENT') {
        return cb(err)
      }

      // The file doesn't exist, so let's download it
      return download(url, filename, (err, requestContent) => {
        if (err) {
          return cb(err)
        }

        spiderLinks(url, requestContent, nesting, cb)
      })
    }

    // The file already exists, let's process the links
    spiderLinks(url, fileContent, nesting, cb)
  })
}
```

```

    })
  }

```

In the next section, we will explore how the `spiderLinks()` function can be implemented.

Sequential crawling of links

Now, we can create the core of this new version of our web spider application, the `spiderLinks()` function, which downloads all the links of an HTML page using a sequential asynchronous iteration algorithm. Pay attention to the way we are going to define that in the following code block:

```

function spiderLinks (currentUrl, body, nesting, cb) {
  if (nesting === 0) {
    // Remember Zalgo from chapter 3?
    return process.nextTick(cb)
  }

  const links = getPageLinks(currentUrl, body) // (1)
  if (links.length === 0) {
    return process.nextTick(cb)
  }

  function iterate (index) { // (2)
    if (index === links.length) {
      return cb()
    }

    spider(links[index], nesting - 1, function (err) { // (3)
      if (err) {
        return cb(err)
      }
      iterate(index + 1)
    })
  }

  iterate(0) // (4)
}

```

The important steps to understand from this new function are as follows:

1. We obtain the list of all the links contained in the page using the `getPageLinks()` function. This function returns only the links pointing to an internal destination (the same hostname).
2. We iterate over the links using a local function called `iterate()`, which takes the index of the next link to analyze. In this function, the first thing we do is check whether the index is equal to the length of the `links` array, in which case we immediately invoke the `cb()` function, as it means we have processed all the items.
3. At this point, everything should be ready for processing the link. We invoke the `spider()` function by decreasing the nesting level and invoking the next step of the iteration when the operation completes.
4. As the last step in the `spiderLinks()` function, we bootstrap the iteration by invoking `iterate(0)`.

The algorithm that was just presented allows us to iterate over an array by executing an asynchronous operation in sequence, which in our case is the `spider()` function.

Finally, we can change our `spider-cli.js` a bit so that we can specify the nesting level as an additional command-line interface (CLI) argument:

```
import { spider } from './spider.js'

const url = process.argv[2]
const nesting = Number.parseInt(process.argv[3], 10) || 1

spider(url, nesting, err => {
  if (err) {
    console.error(err)
    process.exit(1)
  }

  console.log('Download complete')
})
```

We can now try this new version of the spider application and watch it download all the links of a web page recursively, one after the other. To interrupt the process, which can take a while if there are many links, remember that we can always use `Ctrl + C`. If we then decide to resume it, we can do so by launching the spider application and providing the same URL we used for the first run.



Now that our web spider application can potentially trigger the download of an entire website, please consider using it carefully. For example, do not set a high nesting level or leave the spider running for more than a few seconds. It is not polite to overload a server with thousands of requests. In some circumstances, this can also be considered illegal. 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 really 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

Execute a list of tasks in sequence by creating a function named `iterator`, which invokes the next available task in the collection and makes sure to invoke the next step of the iteration when the current task completes.

In the next section, we will explore the parallel execution pattern, which is more convenient when the order of the various tasks is not important.

Parallel execution

There are some situations where the order of execution of a set of asynchronous tasks is not important, and all we want is to be notified when all those running tasks are completed. Such situations are better handled using a parallel execution flow, as shown in *Figure 4.2*:

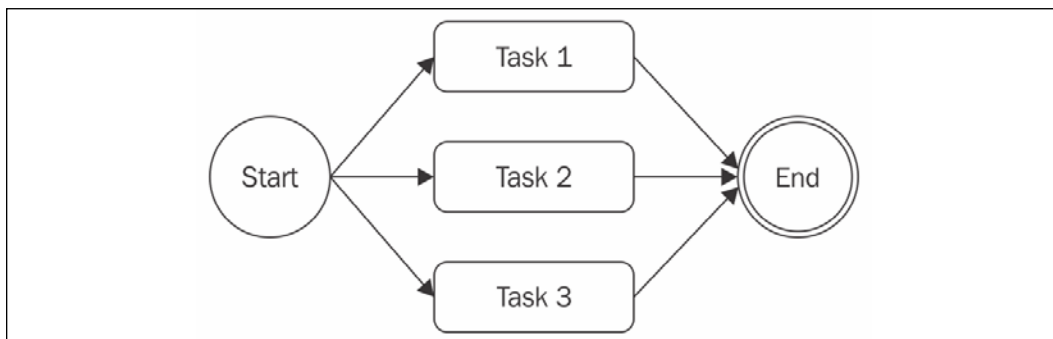


Figure 4.2: An example of parallel execution with three tasks

This may sound strange if you consider that Node.js is single-threaded, but if you remember what we discussed in *Chapter 1, The Node.js Platform*, you'll realize that even though we have just one thread, we can still achieve concurrency, thanks to the non-blocking nature of Node.js. In fact, the word *parallel* is used improperly in this case, as it does not mean that the tasks run simultaneously, but rather that their execution is carried out by an underlying, non-blocking API and interleaved by the event loop.

As you know, a task gives control back to the event loop when it requests a new asynchronous operation, allowing the event loop to execute another task. The proper word to use for this kind of flow is *concurrency*, but we will still use *parallel* for simplicity.

The following diagram shows how two asynchronous tasks can run in parallel in a Node.js program:

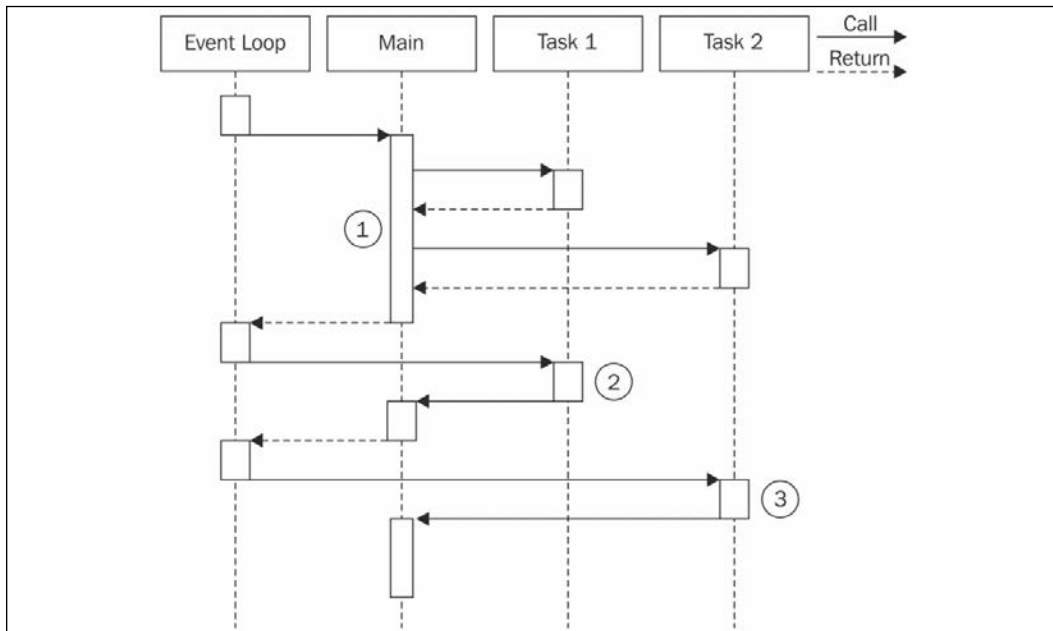


Figure 4.3: An example of how asynchronous tasks run in parallel

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 can only execute asynchronous operations in parallel, because their concurrency is handled internally by the non-blocking APIs. In Node.js, synchronous (blocking) operations can't run concurrently unless their execution is interleaved with an asynchronous operation, or interleaved with `setTimeout()` or `setImmediate()`. You will see this 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 parallel 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 in parallel.

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, nesting, cb) {
  if (nesting === 0) {
    return process.nextTick(cb)
  }

  const links = getPageLinks(currentUrl, body)
  if (links.length === 0) {
    return process.nextTick(cb)
  }

  let completed = 0
  let hasErrors = false

  function done (err) {
    if (err) {
```

```

        hasErrors = true
        return cb(err)
      }
      if (++completed === links.length && !hasErrors) {
        return cb()
      }
    }

    links.forEach(link => spider(link, nesting - 1, done))
  }
}

```

Let's discuss what we changed. 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:

```
links.forEach(link => spider(link, nesting - 1, done))
```

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 is invoked:

```

function done (err) {
  if (err) {
    hasErrors = true
    return cb(err)
  }
  if (++completed === links.length && !hasErrors) {
    return cb()
  }
}

```



The `hasErrors` variable is necessary because if one parallel task fails, we want to immediately call the callback with the given error. Also, we need to make sure that other parallel 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 in parallel, without waiting for the previous link to be processed.

The pattern

Finally, we can extract our nice little pattern for the parallel execution flow. Let's represent a generic version of the pattern with the following code:

```
const tasks = [ /* ... */ ]

let completed = 0
tasks.forEach(task => {
  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 Parallel Execution pattern

Run a set of asynchronous tasks in parallel by launching them all at once, and then wait for all of them to complete by counting the number of times their callbacks are invoked.

When we have multiple tasks running in parallel, 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 in parallel is, in fact, straightforward and cheap in terms of resources.

This is one of the most important strengths of Node.js, because it makes parallelization a common practice rather than a complex technique to only use when strictly necessary.

Another important characteristic of the concurrency model of Node.js is the way we deal with task synchronization and race conditions. In multithreaded programming, this is usually done using constructs such as locks, mutexes, semaphores, and monitors, and it can be one of the most complex aspects of parallelization, and has a considerable impact on performance. 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.

To see a concrete example, we will refer again to our web spider application, and in particular, the last version we created, which actually contains a race condition (can you spot it?). 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, nesting, cb) {
  const filename = urlToFilename(url)
  fs.readFile(filename, 'utf8', (err, fileContent) => {
    if (err) {
      if (err.code !== 'ENOENT') {
        return cb(err)
      }
      return download(url, filename, (err, requestContent) => {
        // ...
      })
    }
  })
}
```

The problem is that two spider tasks operating on the same URL might invoke `fs.readFile()` 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:

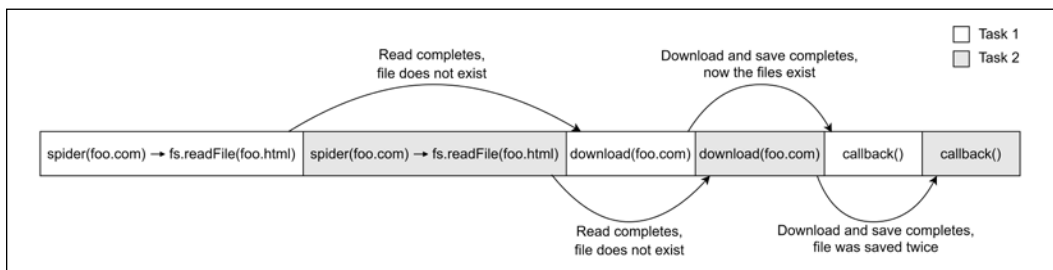


Figure 4.4: An example of a race condition in our `spider()` function

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)

  // ...
}
```

The fix does not require many comments. We simply exit the function immediately if the given `url` is already present in the `spidering` set; otherwise, we add the `url` to the set and continue with the download. In our case, we don't need to release the lock, as we are not interested in downloading a URL twice, even if the spider tasks are executed at two completely different points in time. 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 many problems, even if we are in a single-threaded environment. In some circumstances, they can lead to data corruption and are usually very hard to debug because of their ephemeral nature. So, it's always good practice to double-check for these types of situations when running tasks in parallel.

Also, running an arbitrary number of parallel tasks can be a dangerous practice. In the next section, you will discover why it can be a problem and how to keep the number of parallel tasks under control.

Limited parallel execution

Spawning parallel tasks without control can often lead to excessive load. Imagine having thousands of files to read, URLs to access, or database queries to run in parallel. A common problem in such situations is running out of resources. The most common example is when an application tries to open too many files at once, utilizing all the file descriptors available to the process.

A server that spawns unbounded parallel tasks to handle a user request could be exploited with a **denial-of-service (DoS)** attack. That is when a malicious actor can forge one or more requests to push the server to consume all the available resources and become unresponsive. Limiting the number of parallel tasks is, in general, a good practice that helps with building resilient applications.

Version 3 of our web spider does not limit the number of parallel tasks and therefore, it is susceptible to crashing in a number of cases. For instance, if we try to run it against a significantly big website, we might see it running for a few seconds and then failing with the error code `ECONNREFUSED`. When we are downloading too many pages concurrently from a web server, the server might decide to start rejecting new connections from the same IP. In this case, our spider would just crash and we would be forced to relaunch the process if we wanted to continue crawling the website. We could just handle `ECONNREFUSED` to stop the process from crashing, but we would still be risking allocating too many parallel tasks and might run into other issues.

In this section, you will see how we can make our spider more resilient by keeping the concurrency limited.

The following diagram shows a situation where we have five tasks that run in parallel with a concurrency limit of two:

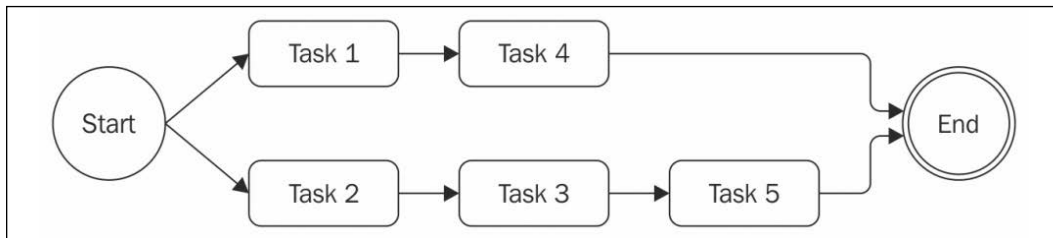


Figure 4.5: An example of how concurrency can be limited to a maximum of two parallel tasks

From *Figure 4.5*, it should be clear how our algorithm works:

1. Initially, we spawn as many tasks as we can without exceeding the concurrency limit.
2. Then, every time a task is completed, we spawn one or more tasks until we reach the limit again.

In the next section, we will explore a possible implementation of the limited parallel execution pattern.

Limiting concurrency

We will now look at a pattern that will execute a set of given tasks in parallel with limited concurrency:

```
const tasks = [
  // ...
]

const concurrency = 2
let running = 0
let completed = 0
let index = 0

function next () {                                     // (1)
  while (running < concurrency && index < tasks.length) {
    const task = tasks[index++]
    task(() => {                                       // (2)
      if (++completed === tasks.length) {
        return finish()
      }
      running--
      next()
    })
    running++
  }
}
next()

function finish() {
  // all tasks finished
}
```

This algorithm can be considered a mixture of sequential execution and parallel execution. In fact, you might notice similarities with both patterns:

1. We have an iterator function, which we call `next()`, and then an inner loop that spawns as many tasks as possible in parallel while staying within the concurrency limit.
2. The next important part is the callback we pass to each task, which checks whether we completed all the tasks in the list. If there are still tasks to run, it invokes `next()` to spawn another set of tasks.

Pretty simple, isn't it?

Globally limiting concurrency

Our web spider application is perfect for applying what we just learned about limiting the concurrency of a set of tasks. In fact, to avoid the situation in which we have thousands of links being crawled at the same time, we can enforce a limit on the concurrency of this process by adding some predictability regarding the number of concurrent downloads.

We could apply this implementation of the limited concurrency pattern to our `spiderLinks()` function, but by doing that, we would only be limiting the concurrency of tasks spawned from the links found within a given page. If we chose, for example, a concurrency of two, we would have, at most, two links downloaded in parallel for each page. However, as we can download multiple links at once, each page would then spawn another two downloads, causing the grand total of download operations to grow exponentially anyway.

In general, this implementation of the limited concurrency pattern works very well when we have a predetermined set of tasks to execute, or when the set of tasks grows linearly over time. When, instead, a task can spawn two or more tasks directly, as happens with our web spider, this implementation is not suitable for limiting the global concurrency.

Queues to the rescue

What we really want, then, is to limit the global number of download operations we can have running in parallel. We could slightly modify the pattern shown in the previous section, but this is left as an exercise for you. Instead, let's discuss another mechanism that makes use of **queues** to limit the concurrency of multiple tasks. Let's see how this works.

We are now going to implement a simple class named `TaskQueue`, which will combine a queue with the algorithm that was presented while discussing limited concurrency. Let's create a new module named `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) {
      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()`. Note that we have to use `bind` because otherwise, the `next` function will lose its context when invoked by `process.nextTick`.

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 '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:

- 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.
- 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*.

Web spider version 4

Now that we have our generic queue to execute tasks in a limited parallel flow, let's use it straightaway 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, nesting, queue, cb) { // (1)
  const filename = urlToFilename(url)
  fs.readFile(filename, 'utf8', (err, fileContent) => {
    if (err) {
      if (err.code !== 'ENOENT') {
        return cb(err)
      }

      return download(url, filename, (err, requestContent) => {
        if (err) {
          return cb(err)
        }
      })
    }
  })
}
```

```

    }

    spiderLinks(url, requestContent, nesting, queue)    // (2)
    return cb()
  })
}

spiderLinks(url, fileContent, nesting, queue)          // (3)
return cb()
})
}

```

Other than renaming the function, you might have noticed that we applied some other small changes:

- 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.
- 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.
- We also need to pass the `queue` instance to `spiderLinks` when we are invoking that from an already downloaded file.

Let's revisit the `spiderLinks()` function. This function can now be greatly simplified as it doesn't have to keep track of task completion anymore, as this work has been delegated to the `queue`. Its job becomes effectively synchronous now; it just needs to invoke the new `spider()` function (which we will define shortly) to push a new task to the `queue`, one for each discovered link:

```

function spiderLinks (currentUrl, body, nesting, queue) {
  if (nesting === 0) {
    return
  }

  const links = getPageLinks(currentUrl, body)
  if (links.length === 0) {
    return
  }

  links.forEach(link => spider(link, nesting - 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, nesting, queue) {
  if (spidering.has(url)) {
    return
  }

  spidering.add(url)
  queue.pushTask((done) => { // (2)
    spiderTask(url, nesting, 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 nesting = 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, nesting, 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. In the next section, we will close this chapter by looking at a famous library that provides a production-ready implementation of these patterns and many other asynchronous utilities.

The `async` library

If you take a look for a moment at every control flow pattern we have analyzed so far, you will see that they can be used as a base to build reusable and more generic solutions. For example, we could wrap the unlimited parallel execution algorithm into a function that accepts a list of tasks, runs them in parallel, and invokes the given callback when all of them are complete. This way of wrapping control flow algorithms into reusable functions can lead to a more declarative and expressive way of defining asynchronous control flows, and that's exactly what `async` (`nodejsdp.link/async`) does.

The `async` library (not to be confused with the `async/await` keywords, which we will discuss later in this book) is a very popular solution, in Node.js and JavaScript in general, for dealing with asynchronous code. It offers a set of functions that greatly simplify the execution of tasks in different configurations, and it also provides useful helpers for dealing with collections asynchronously. Even though there are several other libraries with a similar goal, `async` is the de facto standard in Node.js due to its historic popularity, especially when using callbacks to define asynchronous tasks.

Just to give you an idea of some of the most important capabilities of the `async` module, here is a sample of the functionalities it exposes:

- Execute asynchronous functions over a collection of elements (in series or in parallel with limited concurrency).
- Execute a chain of asynchronous functions (waterfall) where the output of every function becomes the input of the next one.

- Offers a queue abstraction functionally equivalent to the one we implemented with our `TaskQueue` utility.
- Provides other interesting asynchronous patterns such as **race** (executes multiple asynchronous functions in parallel and stops when the first one completes).

Check out the `async` documentation (nodejsdp.link/async) to find out more about the module and to see some examples.

Once you've understood the fundamentals of the asynchronous patterns described in this chapter, you shouldn't rely on the simplified implementations presented here for your everyday control flow needs. Instead, it's better to adopt a broadly used and battle-tested library like `async` for your production applications, unless your use case is so advanced that you require a custom algorithm.

Summary

At the beginning of this chapter, it was stated that Node.js programming can be tough because of its asynchronous nature, especially for people used to developing on other platforms. However, throughout this chapter, you saw how asynchronous APIs can be bent to your will. You discovered that the tools at your disposal are indeed versatile and provide good solutions to most of your problems, in addition to offering a programming style for every taste.

In this chapter, we also kept refactoring and improving our web crawler example. When dealing with asynchronous code, it can sometimes be challenging to figure out the right ergonomics that can keep your code simple and effective, so allow yourself some time to digest the concepts explored in this chapter and to experiment with them.

Our journey with asynchronous Node.js programming has just started. In the next few chapters, you will be introduced to other broadly adopted techniques that leverage promises, and `async/await`. After you've learned all these techniques, you will be able to choose the best solution for your needs or use many of them together in the same project.

Exercises

- **4.1 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).

- **4.2 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.

- **4.3 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 in parallel, but be careful to keep the number of parallel tasks under control!

5

Asynchronous Control Flow Patterns with Promises and Async/Await

Callbacks are the low-level building blocks of asynchronous programming in Node.js, but they are far from being developer-friendly. In fact, in the last chapter, we learned techniques to implement different control flow constructs using callbacks, and we can say that they are quite complex and verbose compared to the (low) level of complexity of the tasks they try to accomplish. In particular, serial execution flow, which is the predominant control flow structure in most of the code we write, 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 many 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 like 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 tackle our asynchronous programming problems with the right approach.

In this chapter, you will learn the following:

- 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.

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 represent a big step ahead 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 all the major 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 embodies 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 **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 a better structured and more elegant code, such as the following:

```
asyncOperationPromise(arg)  
  .then(result => {  
    // do stuff with result  
  }, err => {  
    // handle the error  
  })
```

In the code above, `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 there is nothing major going on, 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:

- Fulfill with x if x is a value
- Fulfill with the fulfillment value of x if x is a Promise
- Reject with the eventual rejection reason of x if x is a Promise

This behavior 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 trivial 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:

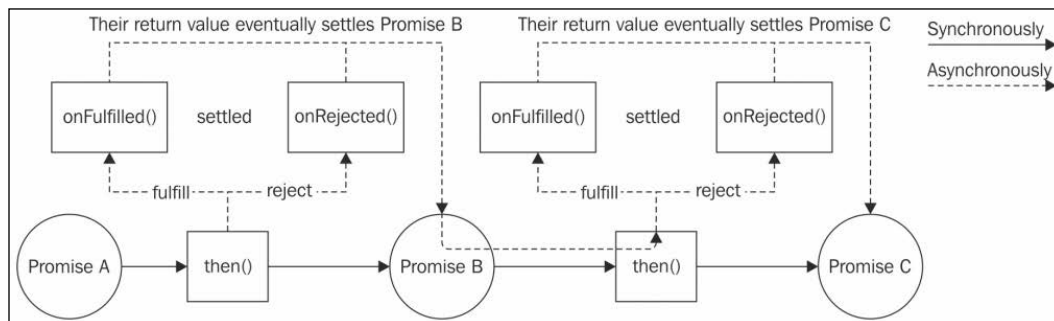


Figure 5.1: Promise chain execution flow

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.

An important property of promises is that the `onFulfilled()` and `onRejected()` callbacks are *guaranteed* to be invoked asynchronously and at most once, even if we resolve the `Promise` synchronously with a value. Not only that, the `onFulfilled()` and `onRejected()` callbacks will be invoked asynchronously even if the `Promise` object is already settled at the moment in which `then()` is called. This behavior shields our code against all those situations where we could unintentionally release Zalgo (see *Chapter 3, Callbacks and Events*), making our asynchronous code more consistent and robust without any extra effort.

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 provided as the rejection reason. 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.

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** 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 an overview to give you an idea of what we can do with promises, so 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.

The Promise constructor (`new Promise((resolve, reject) => {})`) creates a new Promise instance that fulfills 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 fulfill 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 take a 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 fulfills with an array of fulfillment values when every item in the input `iterable` (such as an Array) object fulfills. If any Promise in the `iterable` object rejects, then the Promise returned by `Promise.all()` will reject with the first rejection reason. Each item in the `iterable` object can be a Promise, a generic thenable, or a value.
- `Promise.allSettled(iterable)`: This method waits for all the input promises to fulfill 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 fulfill 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.

Finally, the following are the main methods available on a Promise instance:

- `promise.then(onFulfilled, onRejected)`: This is the essential method of a Promise. 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 once 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

Let's now see how we can create a Promise using its constructor. 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 we—as developers—are consumers of 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 date after a specified number of milliseconds. Let's take a look at it:

```
function delay (milliseconds) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve(new Date())  
    }, milliseconds)  
  })  
}
```

As you probably already guessed, we used `setTimeout()` to invoke the `resolve()` function of the Promise constructor. 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.

The `delay()` function we just created can then be used with some code like the following:

```
console.log(`Delaying...${new Date().getSeconds()}s`)  
delay(1000)  
  .then(newDate => {  
    console.log(`Done ${newDate.getSeconds()}s`)  
  })
```

The `console.log()` within the `then()` handler will be executed approximately after 1 second from the invocation of `delay()`.



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 promises 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 (callbackBasedApi) {
  return function promisified (...args) {
    return new Promise((resolve, reject) => {           // (1)
      const newArgs = [
        ...args,
        function (err, result) {                       // (2)
          if (err) {
            return reject(err)
          }

          resolve(result)
        }
      ]
      callbackBasedApi(...newArgs)                     // (3)
    })
  }
}
```

The preceding function returns another function called `promisified()`, which represents the promisified version of the `callbackBasedApi` given as the input. This is how it works:

1. The `promisified()` 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 make sure to pass to `callbackBasedApi` a special callback. Since we know that the callback always comes last, we simply append it to the arguments list (`args`) provided to the `promisified()` function. In the special callback, if we receive an error, we immediately reject the `Promise`; otherwise, we resolve it with the given result.
3. Finally, we simply invoke `callbackBasedApi` with the list of arguments we have built.

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 conventions we already know very well. Let's see what this looks like:

```
import { randomBytes } from 'crypto'

const randomBytesP = promisify(randomBytes)
randomBytesP(32)
  .then(buffer => {
    console.log(`Random bytes: ${buffer.toString()}`)
  })
```

The previous code should print some gibberish to the console; that's because not all generated bytes have a corresponding 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 of 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

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 webpage 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 'fs'.`

In the `spider.js` module, the very first step required is to import our dependencies and promisify any callback-based function that we are going to use:

```
import { promises as fsPromises } from 'fs'           // (1)
import { dirname } from 'path'
import superagent from 'superagent'
import mkdirp from 'mkdirp'
import { urlToFilename, getPageLinks } from './utils.js'
import { promisify } from 'util'

const mkdirpPromises = promisify(mkdirp)             // (2)
```

There are two main differences here compared to the `spider.js` module of the previous chapter:

1. We import the `promises` object of the `fs` module to get access to all the `fs` functions already promisified.
2. We manually promisify the `mkdirp()` function.

Now, we can start converting the `download()` function:

```
function download (url, filename) {
  console.log(`Downloading ${url}`)
  let content
  return superagent.get(url)                               // (1)
    .then((res) => {
      content = res.text                                    // (2)
      return mkdirpPromises(dirname(filename))
    })
    .then(() => fsPromises.writeFile(filename, content))
    .then(() => {
      console.log(`Downloaded and saved: ${url}`)
    })
}
```

```
        return content // (3)
    })
}
```

We can straightaway appreciate the elegance of implementing sequential asynchronous operations with promises. We simply have a clean and very intuitive chain of `then()` invocations.

Compared to the previous version of the function, this time we are leveraging the out-of-the-box support for promises of the `superagent` package. Instead of invoking `end()` on the request object returned by `superagent.get()`, we simply invoke `then()` to send the request (1) and receive a `Promise` that fulfills/rejects with the result of the request.

The final return value of the `download()` function is the `Promise` returned by the last `then()` call in the chain, which fulfills with the content of the webpage (3), which we initialized in the `onFulfilled` handler of the first `then()` call (2). This makes sure that the caller receives a `Promise` that fulfills with content only after all operations (`get`, `mkdirp`, and `writeFile`) have completed.

In the `download()` function that we've just seen, we have executed a known set of asynchronous operations in sequence. 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, content, nesting) {
  let promise = Promise.resolve() // (1)
  if (nesting === 0) {
    return promise
  }
  const links = getPageLinks(currentUrl, content)
  for (const link of links) {
    promise = promise.then(() => spider(link, nesting - 1)) // (2)
  }

  return promise
}
```

To iterate over all the links of a webpage asynchronously, we had to dynamically build a chain of promises 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 actually 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.

Now, we can finally convert the `spider()` function:

```
export function spider (url, nesting) {
  const filename = urlToFilename(url)
  return fsPromises.readFile(filename, 'utf8')
    .catch((err) => {
      if (err.code !== 'ENOENT') {
        throw err
      }

      // The file doesn't exist, so let's download it
      return download(url, filename)
    })
    .then(content => spiderLinks(url, content, nesting))
}
```

In this new `spider()` function, we are using `catch()` to handle any error produced by `readFile()`. In particular, if the error has code `'ENOENT'`, it means that the file doesn't exist yet and therefore we need to download the corresponding URL. The Promise returned from `download()`, if fulfilled, will return the content at the URL. On the other hand, if the Promise produced by `readFile()` fulfills, it will *skip* the `catch()` handler and go straight to the next `then()`. In both cases, the `onFulfilled` handler of the last `then()` call will always receive the content of the webpage, either coming from the local file or from a fresh download.

Now that we have converted our `spider()` function as well, we can finally modify the `spider-cli.js` module:

```
spider(url, nesting)
  .then(() => console.log('Download complete'))
  .catch(err => console.error(err))
```


The `catch()` handler here will intercept any error originating from the entire `spider()` process.

If we look again at all the code we have written so far, we will be pleasantly surprised by the fact that we haven't included any error propagation logic (as we would be forced to do when using callbacks). 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.



An alternative of 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())
```

Parallel execution

Another execution flow that becomes trivial with promises is the parallel 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 fulfills 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 in parallel.

To demonstrate this, let's consider version 3 of our web spider application, which downloads all the links of a page in parallel. Let's just update the `spiderLinks()` function again to implement a parallel execution flow using promises:

```
function spiderLinks (currentUrl, content, nesting) {  
  if (nesting === 0) {  
    return Promise.resolve()  
  }  
}
```

```
const links = getPageLinks(currentUrl, content)
const promises = links.map(link => spider(link, nesting - 1))

return Promise.all(promises)
}
```

The pattern here consists in starting the `spider()` 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 parallel execution

So far, promises have not disappointed our expectations. We were able to greatly improve our code for both serial and parallel execution. Now, with limited parallel execution, things should not be that different, considering that this flow is just a combination of serial and parallel 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. In other words, we are going to implement our solution in a class that we can use to instantiate objects that we can pass around to different functions of the same application. If you are just interested in a simple solution to locally limit the parallel 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.



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 are going to reuse the TaskQueue class we implemented in the previous chapter. Let's start with the next() method, where we trigger the execution of a set of tasks until we reach the concurrency limit:

```
next () {
  while (this.running < this.concurrency && this.queue.length) {
    const task = this.queue.shift()
    task().finally(() => {
      this.running--
      this.next()
    })
    this.running++
  }
}
```

The core change in the next() method is where we invoke task(). In fact, now we expect that task() returns a Promise, so all we have to do is invoke finally() on that Promise so we can reset the count of running tasks if it either fulfills or rejects.

Now, we implement a new method called runTask(). This method is responsible for queueing a special wrapper function and also for returning a newly built Promise. Such a Promise will essentially forward the result (fulfillment or rejection) of the Promise eventually returned by task(). Let's see what this method looks like:

```
runTask (task) {
  return new Promise((resolve, reject) => {           // (1)
    this.queue.push(() => {                           // (2)
      return task().then(resolve, reject)             // (4)
    })
    process.nextTick(this.next.bind(this))           // (3)
  })
}
```

In the method we have just seen:

1. We create a new Promise using its constructor.
2. We add a special wrapper function to the tasks queue. This function is going to be executed at a later next() run, when there are enough concurrency slots left.

3. We invoke `next()` to trigger a new set of tasks to be run. We defer this to a subsequent run of the event loop to guarantee that task is always invoked asynchronously with respect to when `runTask()` is invoked. This prevents the problems we described in *Chapter 3, Callbacks and Events* (for example, Zalgo). In fact, we can notice that in the `next()` method there is another invocation of `next()` itself, in the `finally()` handler, that is always asynchronous.
4. When the wrapper function we queued is finally run, we execute the task we have received as the input, and we forward its results—fulfilment value or rejection reason—to the outer Promise, the one we return from the `runTask()` method.

With this, we have completed the implementation of our new `TaskQueue` class using promises. 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 parallel execution flow using the `TaskQueue` class we have just created.

First, we need to split the `spider()` function into two functions, one simply initializing a new `TaskQueue` object and another actually executing the spidering task, which we will call `spiderTask()`. Then, we need to update the `spiderLinks()` function to invoke the newly created `spiderTask()` function and forward the task queue instance received as an input. Let's see what all this looks like:

```
function spiderLinks (currentUrl, content, nesting, queue) {
  if (nesting === 0) {
    return Promise.resolve()
  }

  const links = getPageLinks(currentUrl, content)
  const promises = links
    .map(link => spiderTask(link, nesting - 1, queue))

  return Promise.all(promises) // (2)
}

const spidering = new Set()
function spiderTask (url, nesting, queue) {
  if (spidering.has(url)) {
    return Promise.resolve()
  }
}
```

```
spidering.add(url)

const filename = urlToFilename(url)

return queue
  .runTask(() => {                                     // (1)
    return fsPromises.readFile(filename, 'utf8')
    .catch((err) => {
      if (err.code !== 'ENOENT') {
        throw err
      }

      // The file doesn't exists, so Let's download it
      return download(url, filename)
    })
  })
  .then(content => spiderLinks(url, content, nesting, queue))
}

export function spider (url, nesting, concurrency) {
  const queue = new TaskQueue(concurrency)
  return spiderTask(url, nesting, queue)
}
```

The crucial instruction in the code we have just seen is where we invoke `queue.runTask()` (1). Here, the task that we are queuing (and therefore limiting) comprises just the retrieval of the contents of the URL from either the local filesystem or the remote URL location. Only after this task has been run by the queue can we continue to spider the links of the webpage. Note that we are intentionally keeping `spiderLinks()` outside of the task that we want to limit. This is because `spiderLinks()` can trigger more `spiderTasks()` and that would create a deadlock if the depth of the spidering process is higher than the concurrency limit of the queue.

We can also notice how in `spiderLinks()` we simply continue to use `Promise.all()` (2) to download all the links of a webpage in parallel. This is because it's the responsibility of our queue to limit the concurrency of the tasks.



In production code, you can use the package `p-limit` (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.

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` dichotomy 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 a 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...', new Date())  
  
  const dateAfterOneSecond = await delay(1000)  
  console.log(dateAfterOneSecond)
```

```
const dateAfterThreeSeconds = await delay(3000)
console.log(dateAfterThreeSeconds)

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 saved, and the control returned to the event loop. Once the `Promise` that has been *awaited* resolves, the control is given back to the `async` function, returning the fulfilment value of the `Promise`.



The `await` expression works with any value, not just promises. If a value other than a `Promise` is provided, then its behavior is similar to awaiting a value that it first passed to `Promise.resolve()`.

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.



Invoking an `async` function is instantaneous, like any other asynchronous operation. In other words, `async` functions return a `Promise` synchronously. That `Promise` will then eventually settle based on the result or error produced by the function.

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 a 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.

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 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, invoking the function like this:

```
playingWithErrors(true)
```


Will print to the console the following:

```
We have an error: This is a synchronous error
Done
```

While invoking the function with `false` as the input, like this:

```
playingWithErrors(false)
```

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, supporting 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 function that is returning the Promise.

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. As a consequence, the local catch block will never be invoked. In fact, the previous code will output:

```
Error caught by the caller: Error after 1000ms
```

If our intention is catching locally any error generated by the asynchronous operation that produces the value that we want to return to the caller, 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
```

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 the `download()` function of our web spider; this is how it looks with `async/await`:

```
async function download (url, filename) {  
  console.log(`Downloading ${url}`)  
  const { text: content } = await superagent.get(url)  
  await mkdirpPromises(dirname(filename))  
  await fsPromises.writeFile(filename, content)  
  console.log(`Downloaded and saved: ${url}`)  
  return content  
}
```

Let's appreciate for a moment how simple and compact the `download()` function has become. Let's just consider that the same functionality was implemented with callbacks in two different functions using a total of 19 lines of code. Now we just have seven. 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, content, nesting) {  
  if (nesting === 0) {  
    return  
  }  
  const links = getPageLinks(currentUrl, content)  
  for (const link of links) {  
    await spider(link, nesting - 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` on the Promise returned by `spider()`.

The next code fragment shows the `spider()` function implemented using `async/await`. The aspect to notice here is how errors are easily dealt with using just a `try...catch` statement, making everything easier to read:

```
export async function spider (url, nesting) {  
  const filename = urlToFilename(url)  
  let content  
  try {  
    content = await fsPromises.readFile(filename, 'utf8')  
  } catch (err) {
```

```
    if (err.code !== 'ENOENT') {  
      throw err  
    }  
  
    content = await download(url, filename)  
  }  
  
  return spiderLinks(url, content, nesting)  
}
```

And 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, of course, 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 in parallel, and the execution continues immediately after invoking `forEach()`, without waiting for all the `spider()` operations to complete.

Parallel execution

There are mainly two ways to run a set of tasks in parallel 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. If we wanted to purely use the `await` expression to implement an unlimited parallel asynchronous execution flow, we would do it with some code like the following:

```
async function spiderLinks (currentUrl, content, nesting) {
  if (nesting === 0) {
    return
  }
  const links = getPageLinks(currentUrl, content)
  const promises = links.map(link => spider(link, nesting - 1))
  for (const promise of promises) {
    await promise
  }
}
```

That's it—very simple. In the previous code, we first start all the `spider()` tasks in parallel, 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 reject. 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 parallel 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)
}

```

What we just learned about parallel 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 parallel execution

To implement a limited parallel execution pattern with `async/await`, we can simply reuse the `TaskQueue` class that we created in the *Limited parallel execution* subsection within the *Promises* section. We can either use it as it is or convert its internals to `async/await`. Converting the `TaskQueue` class to `async/await` is a trivial operation and we'll leave this to you as an exercise. Either way, the `TaskQueue` external interface shouldn't change; both implementations will have a `runTask()` method that returns a `Promise` that settles when the task has been run by the queue.

Starting from this assumption, converting the web spider v4 from promises to `async/await` is also a trivial task and we won't show all the steps here as we wouldn't be learning anything new. Instead, what we'll do in this section is examine a third variation of the `TaskQueue` class that uses `async/await` and a **producer-consumer** approach.

The general idea to apply this approach to our problem goes as follows:

- On one side, we have an unknown set of *producers* adding tasks into a queue.
- On the other side, we have a predefined set of *consumers*, responsible for extracting and executing the tasks from the queue, one at a time.

The following diagram should help us understand the setup:

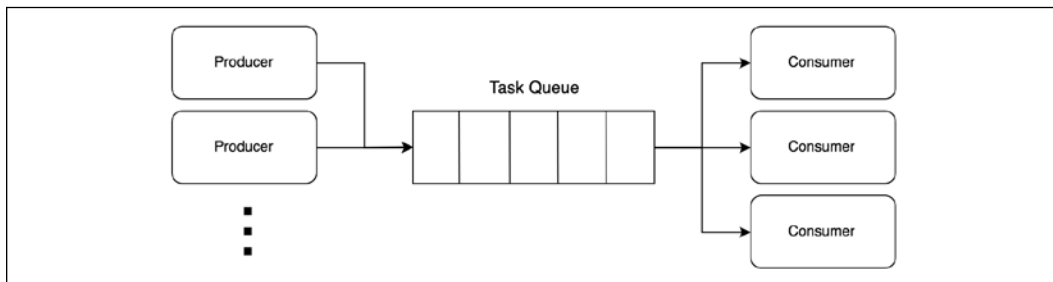


Figure 5.2: Using the Producer-Consumer pattern to implement limited parallel execution

The number of consumers will determine the concurrency with which the tasks will be executed. The challenge here is to put the consumers to "sleep" when the queue is empty and "wake them up" again when there are new tasks to run. But we are lucky, since Node.js is single-threaded, so putting a task to "sleep" just means giving back control to the event loop, while "resuming" a task is equivalent to invoking a callback.

With this in mind, let's then take a look at some code. We will create a new class called `TaskQueuePC` with a public interface similar to one of the `TaskQueue` classes we implemented previously in this chapter. Taking a top-down approach, let's see how we can implement the constructor:

```
export class TaskQueuePC {
  constructor (concurrency) {
    this.taskQueue = []
    this.consumerQueue = []

    // spawn consumers
    for (let i = 0; i < concurrency; i++) {
      this.consumer()
    }
  }

  // ...
}
```

First of all, we can notice that we now have two queues, one to hold our tasks (`taskQueue`) and the other to store our sleeping consumers (`consumerQueue`). It will be clearer in a while how these queues will be used. In the second part of our constructor, we spawn as many consumers as the concurrency we want to attain. Let's see what a consumer looks like:

```
async consumer () {
  while (true) { // (1)
    try {
      const task = await this.getNextTask() // (2)
      await task() // (3)
    } catch (err) {
      console.error(err) // (4)
    }
  }
}
```

Our consumer is an infinite `while` loop (1). At each iteration, we try to retrieve a new task from the queue using `getNextTask()` (2). As we will see, this will cause the current consumer to sleep if the queue is empty. When a new task is eventually available, we just execute it (3). Any error thrown from the above operation should not cause the consumer to stop, so we simply log it (4) and continue with the next iteration.



By the look of it, it may seem that each consumer in `TaskQueuePC` is an actual thread. In fact, our `consumer()` function has an infinite loop and it can "pause" until awakened by some other "thread." In reality, we should not forget that each consumer is an `async` function, which is nothing more than a nice syntax built around promises and callbacks. The `while` loop may seem to be spinning continuously consuming CPU cycles, but under the hood, the loop is more similar to an asynchronous recursion than a traditional `while` loop.

With the next code fragment, we should start to get an idea of what's going on. Let's take a look at the implementation of `getNextTask()`:

```
async getNextTask () {
  return new Promise((resolve) => {
    if (this.taskQueue.length !== 0) {
      return resolve(this.taskQueue.shift()) // (1)
    }

    this.consumerQueue.push(resolve) // (2)
  })
}
```

The `getNextTask()` method returns a new `Promise` that resolves with the first task in the queue if the queue is not empty. The first task is removed from `taskQueue` and used as an argument to invoke `resolve` (1). If the queue is instead empty, we postpone the resolution of the `Promise` by queuing the `resolve` callback into the `consumerQueue`. This will effectively put the `Promise`—and the consumer that is awaiting the `Promise`—to sleep.

Now comes the "gluing" part of the whole `TaskQueuePC` class, which corresponds to the *producer* side of the algorithm. That's implemented in the `runTask()` method:

```
runTask (task) {
  return new Promise((resolve, reject) => {
    const taskWrapper = () => { // (1)
      const taskPromise = task()
```



```
        taskPromise.then(resolve, reject)
        return taskPromise
    }

    if (this.consumerQueue.length !== 0) { // (2)
        const consumer = this.consumerQueue.shift()
        consumer(taskWrapper)
    } else { // (3)
        this.taskQueue.push(taskWrapper)
    }
})
}
```

First, we create a `taskWrapper` function (1) that, when executed, has the responsibility for running the input task and forwarding the status of the Promise returned by `task()` to the outer Promise returned by `runTask()`. Next, if the `consumerQueue` is not empty (2), it means that there is at least one consumer that is asleep, waiting for new tasks to run. We then extract the first consumer from the queue (remember, that's essentially the `resolve` callback of the Promise returned by `getNextTask()`) and we invoke it immediately by passing our `taskWrapper`. If, instead, all the consumers are already busy (3), we push `taskWrapper` into the `taskQueue`.

This concludes the implementation of our `TaskQueuePC` class. The public interface of the `TaskQueuePC` class is identical to that of the `TaskQueue` class that we implemented in the *Promises* section, so migrating the code of our web spider to the new algorithm will be a trivial task.

This also concludes our exploration of the `async/await` construct. But, before we wrap up the chapter, we'll dive into a subtle problem affecting promises.

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 therefore 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 seems to affect 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 like the encoding/decoding of live audio/video streams, the processing of live cryptocurrency market data, and the monitoring of IoT sensors. But we can have much more trivial situations than those, for example, when making heavy use of functional programming.

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 `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 code above 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 on the Promises/A+ GitHub repository at nodejsdp.link/promisesaplus-memleak.

So, the next time you are building an infinite promise chain, remember to double-check if 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've 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 in parallel is also very easy thanks to the `Promise.all()` utility.

But the advantages of using promises and `async/await` don't stop here. We've 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. 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 the 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 parallel 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 in parallel at each given time.

6

Coding with Streams

Streams are 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, a top contributor to the Node.js community, defines streams as "Node's best and most misunderstood idea." There are different reasons that make Node.js streams so attractive; again, 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.

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:

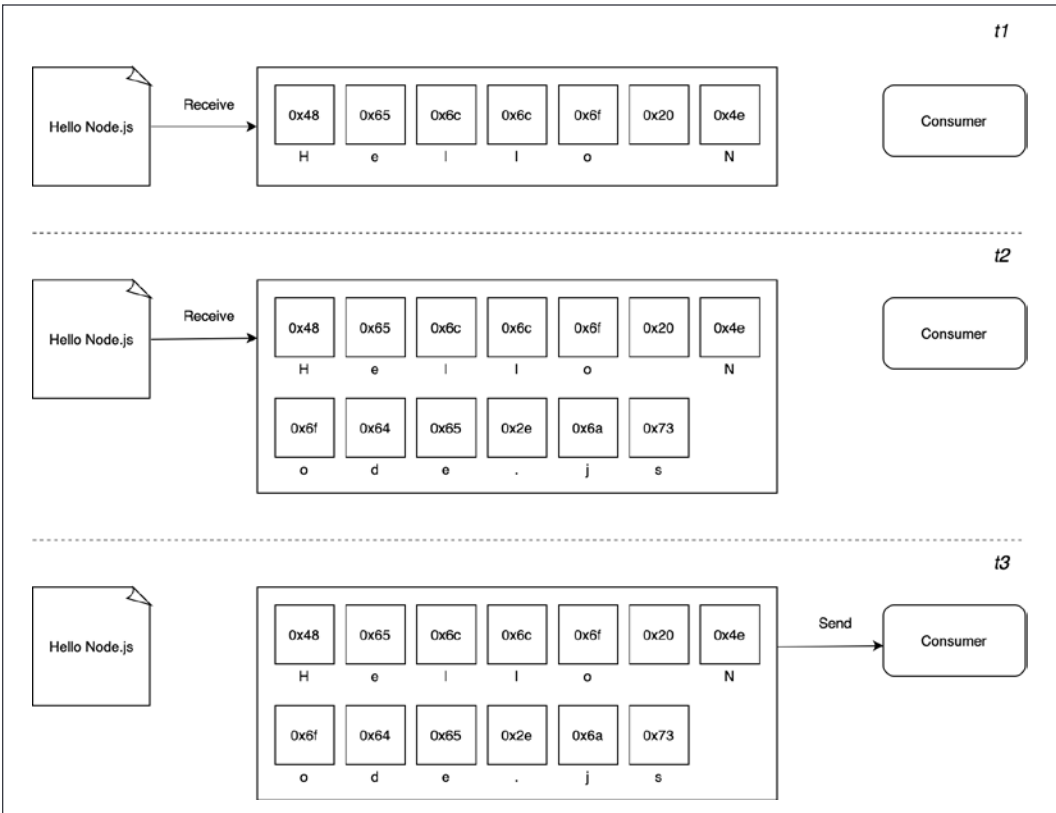


Figure 6.1: Buffering

In *Figure 6.1*, we can see that, at time $t1$, some data is received from the resource and saved into the buffer. At time $t2$, another data chunk is received – the final one – which completes the read operation, so that, at $t3$, the entire buffer is sent to the consumer.

On the other side, streams allow us to process the data as soon as it arrives from the resource. This is shown in the following diagram:

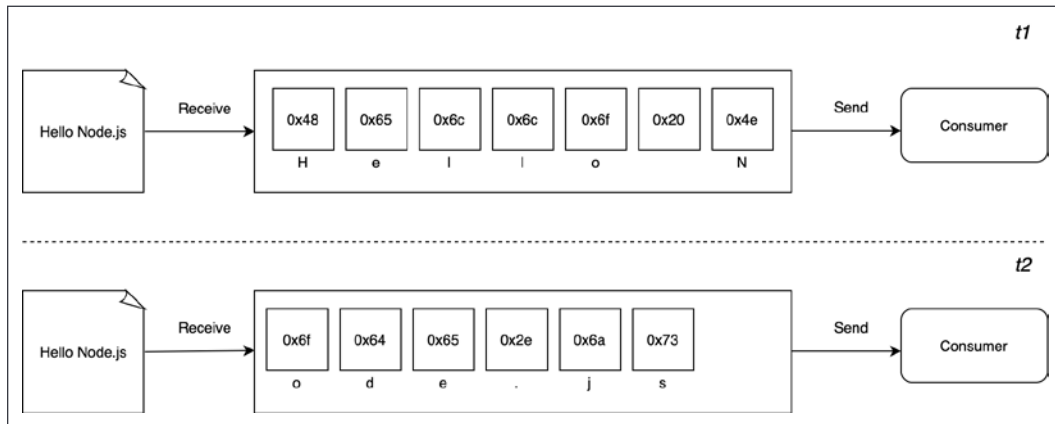


Figure 6.2: Streaming

This time, *Figure 6.2* shows you 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 can be more efficient in terms of both space (memory usage) and time (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's the limit in bytes in a given platform, you can run this code:

```
import buffer from '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):

```
import { promises as fs } from 'fs'
import { gzip } from 'zlib'
import { promisify } from 'util'
const gzipPromise = promisify(gzip)

const filename = process.argv[2]

async function main () {
  const data = await fs.readFile(filename)
  const gzippedData = await gzipPromise(data)
  await fs.writeFile(`${filename}.gz`, gzippedData)
  console.log('File successfully compressed')
}

main()
```

Now, we can try to put the preceding code in a file named `gzip-buffer.js` and then run it with the following command:

```
node gzip-buffer.js <path to file>
```

If we choose a file that is big enough (for instance, about 8 GB), we will most likely receive an error message saying that the file that we are trying to read is bigger than the maximum allowed buffer size:

```
RangeError [ERR_FS_FILE_TOO_LARGE]: File size (8130792448) is greater
than possible Buffer: 2147483647 bytes
```

That's exactly what we expected, and it's a symptom of the fact that we are using the wrong approach.

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 'fs'
import { createGzip } from '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

Let's now consider the case of an application that compresses a file and uploads it to a remote HTTP server, which, in turn, decompresses it and saves it on the filesystem. If the client component of our application was implemented using a buffered API, the upload would start only when the entire file had been read and compressed. On the other hand, the decompression would start on the server only when all the data had been received. A better solution to achieve the same result involves the use of streams. On the client machine, streams allow us to compress and send the data chunks as soon as they are read from the filesystem, whereas on the server, they allow us to decompress every chunk as soon as it is received from the remote peer. Let's demonstrate this by building the application that we mentioned earlier, starting from the server side.

Let's create a module named `gzip-receive.js` containing the following code:

```
import { createServer } from 'http'
import { createWriteStream } from 'fs'
import { createGunzip } from 'zlib'
import { basename, join } from 'path'

const server = createServer((req, res) => {
  const filename = basename(req.headers['x-filename'])
  const destFilename = join('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, `req` is a stream object that is used by the server to receive the request data in chunks from the network. Thanks to Node.js streams, every chunk of data is decompressed and saved to disk as soon as it is received.



You might have noticed that, in our server application, we are using `basename()` to remove any possible path from the name of the received file. This is a security best practice as we want to make sure that the received file is saved exactly within our `received_files` folder. Without `basename()`, a malicious user could craft a request that could effectively override system files and inject malicious code into the server machine. Imagine, for instance, what happens if `filename` is set to `/usr/bin/node`? In such a case, the attacker could effectively replace our Node.js interpreter with any arbitrary file.

The client side of our application will go into a module named `gzip-send.js`, and it looks as follows:

```
import { request } from 'http'
import { createGzip } from 'zlib'
import { createReadStream } from 'fs'
import { basename } from 'path'

const filename = process.argv[2]
const serverHost = process.argv[3]

const httpRequestOptions = {
  hostname: serverHost,
  port: 3000,
  path: '/',
  method: 'PUT',
  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 are again using streams to read the data from the file, and then compressing and sending each chunk as soon as it is read from the filesystem.

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 file big enough, we can appreciate how the data flows from the client to the server. But why exactly is this paradigm – where we have flowing data – more efficient compared to using a buffered API? *Figure 6.3* should make this concept easier to grasp:

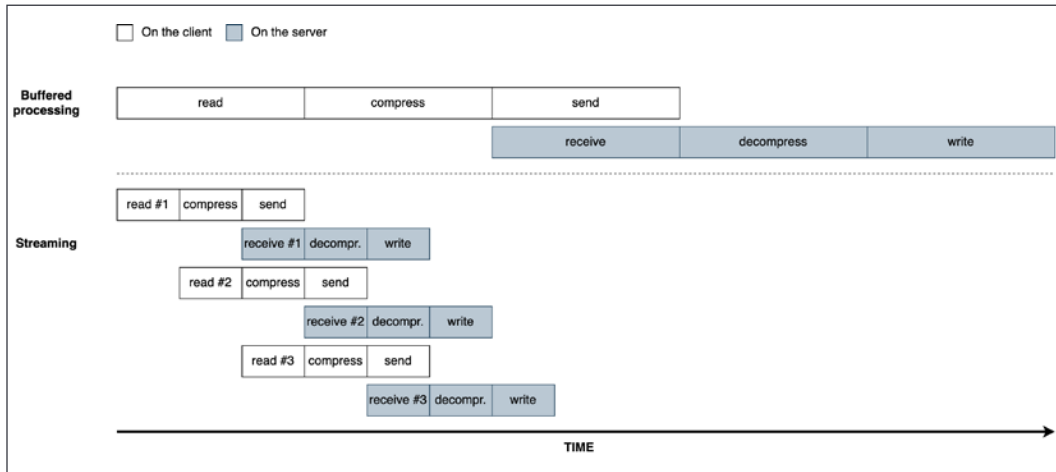


Figure 6.3: Buffering and streaming compared

When a file is processed, it goes through a number of sequential stages:

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 have to 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 in parallel. This works perfectly because each task that we execute is asynchronous, so it can be parallelized by Node.js. The only constraint is that the order in which the chunks arrive in 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.

Composability

The code we have seen so far has already given us an overview of how streams can be composed thanks to the `pipe()` method, which allows us to connect the different processing units, each being responsible for one single functionality, in perfect Node.js style. This is possible because streams have a uniform interface, and they can understand each other in terms of API. The only prerequisite is that the next stream in the pipeline has to support the data type produced by the previous stream, which can be either binary, text, or even objects, as we will see later in this chapter.

To take a look at another demonstration of the power of this property, we can try to add an encryption layer to the `gzip-send/gzip-receive` application that we built previously.

In order to do this, we will need to apply some small changes to both our client and server.

Adding client-side encryption

Let's start with the client:

```
// ...
import { createCipheriv, randomBytes } from 'crypto' // (1)
const filename = process.argv[2]
const serverHost = process.argv[3]
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 `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 for the file encryption.

Now, we can update the piece of code responsible for creating the HTTP request:

```
const httpRequestOptions = {
  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 `crypto` module, which we can use to generate a random encryption key (the secret):

```
// ...
import { createDecipheriv, randomBytes } from '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** ([nodejsdp.link/iv](https://nodejs.org/en/docs/faq/faq/iv/)) 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 as a means to simplify and modularize the code.

Now that we have introduced streams, we are ready to explore, in a more structured way, the different types of streams available in Node.js.

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 event, such as `end` when a `Readable` stream has finished reading, `finish` when a `Writable` stream has completed writing, 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 of 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 imperatively 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:

```
async function main () {
  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')
}

main()
```

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 if you need to write a function that consumes an entire Readable stream and returns a Promise, this syntax could come in very handy.

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 usually indicates that the method is not 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 called `random-stream.js` that contains the code of our random string generator:

```
import { Readable } from 'stream'
import Chance from 'chance'

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)
    }
  }
}
```

At the top of the file, we load our dependencies. There is nothing special there, except that we are loading an npm module called `chance` (`nodejsdp.link/chance`), which is a library for generating all sorts of random values, from numbers to strings to entire sentences.

The next step is to create a new class called `RandomStream`, which specifies `Readable` as its parent. In the preceding code, invoking `super(options)` in the `RandomStream` constructor will call the constructor of the parent class, allowing us to initialize the stream's internal state.



If you have a constructor that only invokes `super(options)`, you can remove it as you will inherit the parent constructor. 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)

Okay, now let's explain 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.

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.

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:

```
// 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 flowing 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 our `RandomStream` using the simplified constructor approach:

```
import { Readable } from 'stream'
import Chance from 'chance'

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 use 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.

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 with this helper, let's look at a simple example where we convert data from an array into a Readable stream:

```
import { Readable } from '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



Try not to instantiate 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 error, or the standard output interface. In Node.js, it's 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 'http'
import Chance from 'chance'

const chance = new Chance()
const server = createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' }) // (1)
  while (chance.bool({ likelihood: 95 })) { // (2)
    res.write(`${chance.string()}\n`) // (3)
  }
  res.end('\n\n') // (4)
  res.on('finish', () => console.log('All data sent')) // (5)
})
server.listen(8080, () => {
  console.log('listening on http://localhost:8080')
})
```

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.
2. We start a loop that terminates with a likelihood of 5% (we instruct `chance.bool()` to return `true` 95% of the time).
3. Inside the loop, we write a random string into the stream.
4. 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.
5. 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:8080` or use `curl` from the terminal as follows:

```
curl localhost:8080
```

At this point, the server should start sending random strings to the HTTP client that you chose (please bear in mind that some browsers might buffer the data, and the streaming behavior might not be apparent).

Backpressure

Similar to a liquid flowing in a real piping system, Node.js streams can also suffer from bottlenecks, where data is written faster than the stream can consume it. The mechanism to cope with this problem involves buffering the incoming data; however, if the stream doesn't give any feedback to the writer, we may incur a situation where more and more data is accumulated in the internal buffer, leading to undesired levels of memory usage.

To prevent this from happening, `writable.write()` will return `false` when the internal buffer exceeds the `highWaterMark` limit. In `Writable` streams, the `highWaterMark` property is the limit of the internal buffer size, beyond which the `write()` method starts returning `false`, indicating that the application should now stop writing. When the buffer is emptied, the `drain` event is emitted, communicating that it's safe to start writing again. This mechanism is called **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 server = createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  function generateMore () {                                     // (1)
    while (chance.bool({ likelihood: 95 })) {
      const randomChunk = chance.string({                       // (2)
        length: (16 * 1024) - 1
      })
    }
  }
})
```

```
    const shouldContinue = res.write(`${randomChunk}\n`) // (3)
    if (!shouldContinue) {
      console.log('back-pressure')
      return res.once('drain', generateMore)
    }
  }
  res.end('\n\n')
}
generateMore()
res.on('finish', () => console.log('All data sent'))
})
// ...
```

The most important steps of the previous code can be summarized as follows:

1. We wrapped the main 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. 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`, 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.

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 has to 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 has to work in object mode.

Let's call the module `to-file-stream.js`:

```
import { Writable } from 'stream'
import { promises as fs } from 'fs'
import { dirname } from 'path'
import mkdirp from 'mkdirp-promise'

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:

```
import { join } from 'path'
import { ToFileStream } from './to-file-stream.js'
const tfs = new ToFileStream()

tfs.write({
  path: join('files', 'file1.txt'), content: 'Hello' })
tfs.write({
  path: join('files', 'file2.txt'), content: 'Node.js' })
tfs.write({
  path: join('files', '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 our 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:

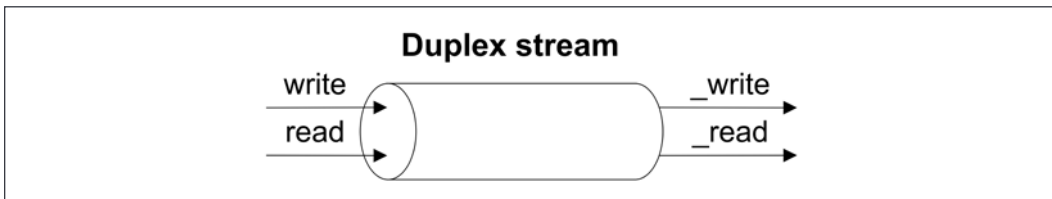


Figure 6.4: Duplex stream schematic representation

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:

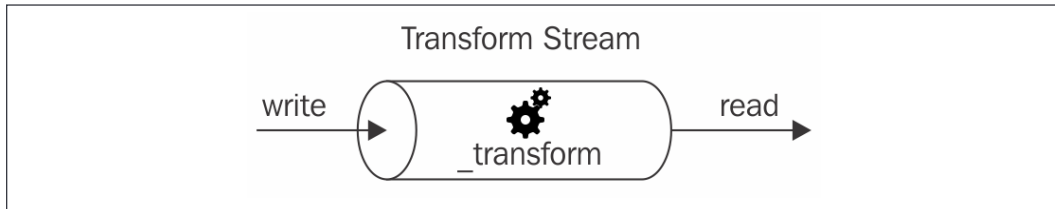


Figure 6.5: Transform stream schematic representation

From the outside, the interface of a Transform stream is exactly like that of a Duplex stream. However, when we want to build 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. To do this, we have to create a new module named `replaceStream.js`. Let's jump directly to the implementation:

```
import { Transform } from 'stream'

export class ReplaceStream extends Transform {
  constructor (searchStr, replaceStr, options) {
    super({ ...options })
    this.searchStr = searchStr
    this.replaceStr = replaceStr
    this.tail = ''
  }

  _transform (chunk, encoding, callback) {
    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)
        callback()
    }

    _flush (callback) {
        this.push(this.tail)
        callback()
    }
}

```

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.

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:

```
import { ReplaceStream } from './replace-stream.js'

const replaceStream = new ReplaceStream('World', 'Node.js')
replaceStream.on('data', chunk => console.log(chunk.toString()))

replaceStream.write('Hello W')
replaceStream.write('orld!')
replaceStream.end()
```

To make life a little bit harder for our stream, we spread the search term (which is `World`) across two different chunks, then, using the flowing mode, we read from the same stream, logging each transformed chunk. Running the preceding program should produce the following output:

```
Hel
lo Node.js
!
```



Please note that the preceding output is broken into multiple lines because we are using `console.log()` to print it out. This allows us to demonstrate that our implementation is able to replace string matches correctly, even when the matching text spans multiple chunks of data.

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:

```
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 mentioned in the previous section, `Transform` streams are the perfect building blocks for implementing data transformation pipelines. In the previous section, we illustrated an example of a `Transform` stream that can replace words in a stream of text. But `Transform` streams can be used to implement other types of data transformation as well. For instance, it's quite common to use `Transform` streams to implement data filtering and data aggregation.

Just to make a practical example, let's imagine we are asked by a Fortune 500 company to analyze a big file containing all the sales for the year 2020. The company wants us to use `data.csv`, a sales report in CSV format, to calculate the total profit for the sales made in Italy.

For simplicity, 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 have to find all the records that have "Italy" as country 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 `csv-parse` module (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:

```
import { createReadStream } from 'fs'
import parse from 'csv-parse'
import { FilterByCountry } from './filter-by-country.js'
import { SumProfit } from './sum-profit.js'

const csvParser = parse({ columns: true })

createReadStream('data.csv')           // (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`.

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 one 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:

```
import { Transform } from '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 filter on. In the constructor, we also set the stream to run in `objectMode` because we know it will be used to process objects (records 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:

```
import { Transform } from '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!

PassThrough streams

There is a fifth type of stream that is worth mentioning: `PassThrough`. This type of stream is a special type of `Transform` that outputs every data chunk without applying any transformation.

`PassThrough` is possibly the most underrated type of stream, but there are actually 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 in a given point of a stream pipeline. Let's see a simplified example to be able to appreciate this concept:

```
import { PassThrough } from '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 in a given point of 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.



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 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 to the file stream before the upload. For instance, let's say we want to compress or encrypt the data? Also, what if we have to do 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 also compresses it using the **Brotli** compression. We are going to assume that the third-party `upload()` function is provided in a file called `upload.js`:

```
import { createReadStream } from 'fs'
import { createBrotliCompress } from 'zlib'
import { PassThrough } from 'stream'
import { basename } from '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 the 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 actually 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 actually 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 effectively create proxies for actual stream instances, where the proxied instance is not created until some piece of code is actually starting 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 lazy `Writable` streams 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, which has the following interface:

```
readable.pipe(writable, [options])
```

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 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 worse, in the event of an error, the failing stream is only uniped 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—definitely something we should avoid if we can!

Luckily, the core `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)
```

This helper pipes every stream passed in the arguments list to the next one. For each stream, it 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. 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.

In order to build up some practice with this helper, 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

Let's call this module `uppercasify-gzipped.js`:

```
import { createGzip, createGunzip } from 'zlib'           // (1)
import { Transform, pipeline } from 'stream'

const uppercasify = new Transform({                      // (2)
  transform (chunk, enc, cb) {
    this.push(chunk.toString().toUpperCase())
  },
  cb()
})
```

```
    }  
  })  
  
  pipeline(                                     // (3)  
    process.stdin,  
    createGunzip(),  
    uppercasify,  
    createGzip(),  
    process.stdout,  
    (err) => {                                   // (4)  
      if (err) {  
        console.error(err)  
        process.exit(1)  
      }  
    }  
  )  
}
```

In this example:

1. We are importing the necessary dependencies from `zlib` and the `stream` 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.
4. We add a callback to monitor the completion of the stream. In the event of an error, we print the error in the standard error interface, and we exit with error code 1.

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 error similar to the following:

```
Error: unexpected end of file
  at Zlib.zlibOnError [as onerror] (zlib.js:180:17) {
  errno: -5,
  code: 'Z_BUF_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.



The `pipeline()` function can be easily *promisified* by using the `promisify()` helper from the `core util` module.

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.

Some code is always better than too much explanation, so let's work 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 { createWriteStream, createReadStream } from 'fs'
import { Readable, Transform } from '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 })
          src.on('error', done)
          src.on('end', done)                          // (3)
        }
      }))
      .on('error', reject)
      .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. Let's do that in a new file, called `concat.js`:

```
import { concatFiles } from './concat-files.js'

async function main () {
  try {
    await concatFiles(process.argv[2], process.argv.slice(3))
  } catch (err) {
    console.error(err)
    process.exit(1)
  }

  console.log('All files concatenated successfully')
}
main()
```

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 parallel task execution.

Unordered parallel execution

We just saw that streams process each data chunk 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 parallelize the execution 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 parallel streams cannot be used when the order in which the data is processed is important.

To parallelize the execution of a Transform stream, we can apply the same patterns that we learned 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 parallel stream

Let's immediately demonstrate how to implement an unordered parallel stream with an example. Let's create a module called `parallel-stream.js` and define a generic Transform stream that executes a given transform function in parallel:

```
import { Transform } from 'stream'

export class ParallelStream 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 && 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. We invoke the parent constructor and for convenience, we enable the object mode by default.

2. Next, it is the `_transform()` method's turn. 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 in parallel 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.

The `ParallelStream` class we just built allows us to easily create a Transform stream that executes its tasks in parallel, but there is a caveat: it does not preserve the order of the items as they are received. In fact, asynchronous operations can complete and push data at any time, regardless of when they are started. We immediately understand that 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 `ParallelStream` 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 `ParallelStream` class to parallelize the checking of the URLs.

Let's build this simple application immediately in a new module called `check-urls.js`:

```
import { pipeline } from 'stream'
import { createReadStream, createWriteStream } from 'fs'
import split from 'split'
```

```

import superagent from 'superagent'
import { ParallelStream } from './parallel-stream.js'

pipeline(
  createReadStream(process.argv[2]),           // (1)
  split(),                                     // (2)
  new ParallelStream(                           // (3)
    async (url, enc, push, done) => {
      if (!url) {
        return done()
      }
      try {
        await superagent.head(url, { timeout: 5 * 1000 })
        push(`${url} is up\n`)
      } catch (err) {
        push(`${url} is down\n`)
      }
      done()
    }
  ),
  createWriteStream('results.txt'),             // (4)
  (err) => {
    if (err) {
      console.error(err)
      process.exit(1)
    }
    console.log('All urls have been checked')
  }
)

```

As we can see, with streams, our code looks very elegant and straightforward: everything is contained in a single streaming pipeline. Let's see how it works:

1. First, we create a `Readable` stream from the file given as input.
2. We pipe the contents of the input file through `split` (nodejsdp.link/split), a `Transform` stream that ensures each line is emitted in a different chunk.
3. Then, it's time to use our `ParallelStream` to check the URL. We do this by sending a `head` request and waiting for a response. When the operation completes, we push the result down the stream.
4. Finally, all the results are piped into a file, `results.txt`.

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 in parallel, 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 `ParallelStream` 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 parallel tasks running at a given time.

Unordered limited parallel 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 in parallel, 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 concurrency of the parallel tasks.

Let's see how this works with streams by creating a `limited-parallel-stream.js` module, which is an adaptation of `parallel-stream.js` 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 LimitedParallelStream 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 have to 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 can simply 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 `ParallelStream` 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 && 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 `LimitedParallelStream` class. We can now use it in the `check-urls.js` module in place of `ParallelStream` and have the concurrency of our tasks limited to the value that we set.

Ordered parallel execution

The parallel 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 in parallel; all we have to do is to sort the data emitted by each task so that it follows the same order in which the data was received.

This technique 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 parallel 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 in parallel. We can do this using `parallel-transform`:

```
//...
import parallelTransform from 'parallel-transform'

pipeline(
  createReadStream(process.argv[2]),
  split(),
  parallelTransform(4, async function (url, done) {
    if (!url) {
      return done()
    }
    console.log(url)
    try {
      await request.head(url, { timeout: 5 * 1000 })
      this.push(`${url} is up\n`)
    } catch (err) {
      this.push(`${url} is down\n`)
    }
    done()
  }),
  createWriteStream('results.txt'),
  (err) => {
    if (err) {
      console.error(err)
      process.exit(1)
    }
    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 4. 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 in parallel and can possibly complete in any order.



When using the ordered parallel 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 in 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:

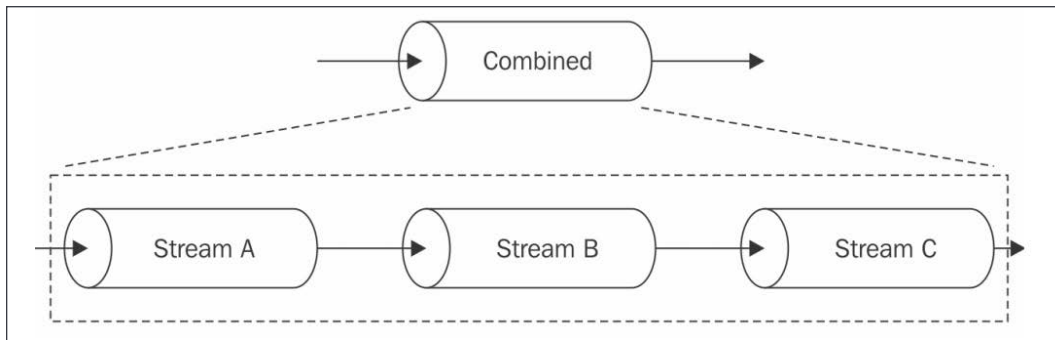


Figure 6.6: Combining streams

From *Figure 6.6*, we should already get a hint of how this works:

- When we write into the combined stream, we are actually writing into the first stream of the pipeline.
- When we read from the combined stream, we are actually 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 `duplexer2` (nodejsdp.link/duplexer2) or `duplexify` (nodejsdp.link/duplexify).

But that's not enough. In fact, another important characteristic of a combined stream is that it has to 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 'fs'
import { Transform, pipeline } from 'stream'
import { strict as assert } from 'assert'

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.strictEqual(streamC, pipelineReturn) // valid  
  
const pipeReturn = streamA.pipe(streamB).pipe(streamC)  
assert.strictEqual(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 a pretty common practice, so if we don't have any particular need, we might just want to reuse an existing library such as `pumpify` (nodejsdp.link/pumpify).

This library offers a very simple interface. In fact, all you have to do to obtain a combined stream is to call `pumpify()`, passing all the streams you want in your pipeline. This is very similar to the signature of `pipeline()`, except that there's no callback:

```
const combinedStream = pumpify(streamA, streamB, streamC)
```

When we do something like this, `pumpify` 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.



If you are curious to see what it takes to build a library like `pumpify`, you should check its source code on GitHub (nodejsdp.link/pumpify-gh). One interesting fact is that, internally, `pumpify` uses `pump` (nodejsdp.link/pump), a module that was born before the Node.js `pipeline()` helper. `pump` is effectively the module that inspired the development of `pipeline()`. If you compare their source code, you will find out that, unsurprisingly, the two modules have a lot in common.

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 a library such as `pumpify`, 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 'zlib'
import {
  createCipheriv,
  createDecipheriv,
  scryptSync
} from 'crypto'
import pumpify from 'pumpify'
function createKey (password) {
  return scryptSync(password, 'salt', 24)
}

export function createCompressAndEncrypt (password, iv) {
  const key = createKey(password)
  const combinedStream = pumpify(
    createGzip(),
    createCipheriv('aes192', key, iv)
  )
  combinedStream.iv = iv

  return combinedStream
}

export function createDecryptAndDecompress (password, iv) {
  const key = createKey(password)
  return pumpify(
    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 'fs'
import { pipeline } from 'stream'
import { randomBytes } from '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 `archiveFile`. 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.



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 to be representing 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. From one side, it allows us to create reusable compositions of streams, and from 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:

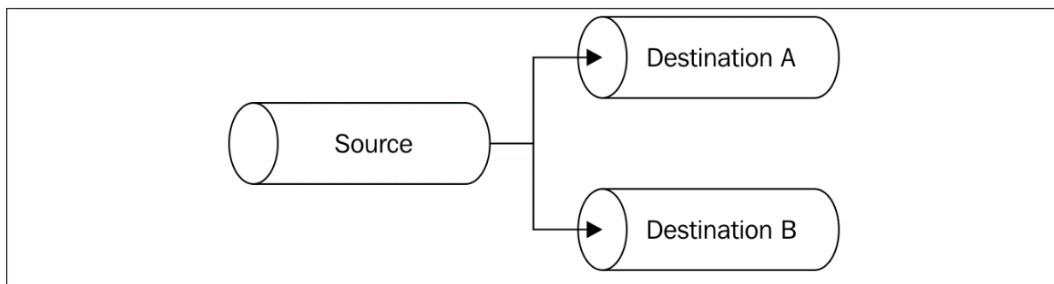


Figure 6.7: Forking a stream

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 'fs'
import { createHash } from '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 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*:

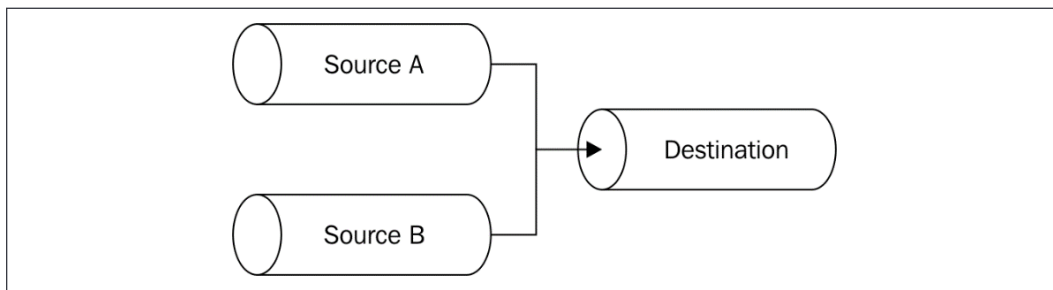


Figure 6.8: Merging streams

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 'fs'
import split from 'split'
```

```
const dest = process.argv[2]
const sources = process.argv.slice(3)
```

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 })
  sourceStream.on('end', () => {
    if (++endCount === sources.length) {
      destStream.end()
      console.log(`${dest} created`)
    }
  })
  sourceStream
    .pipe(split((line) => line + '\n'))
    .pipe(destStream, { end: false })
}
```

In the preceding code, we created a `Readable` stream for every source file. Then, for each source stream, we attached an `end` listener, which will terminate the destination stream only when all the files have been read completely. Finally, we piped every source stream to `split()`, a `Transform` stream that makes sure that we produce a chunk for every line of text, and finally, we piped the results to our destination stream. This is when the real merge happens. We are piping multiple source streams into one single destination.

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:

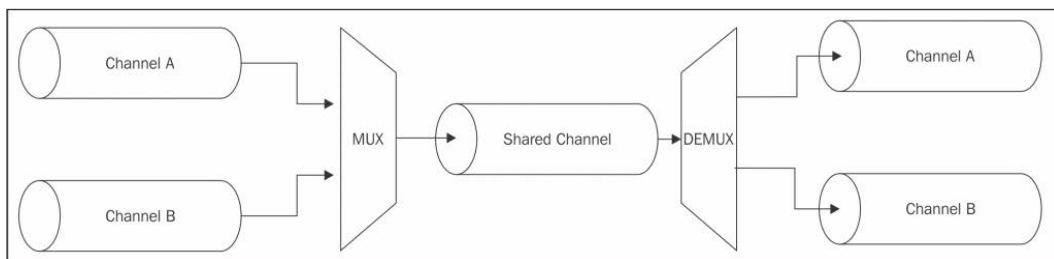


Figure 6.9: Multiplexing and demultiplexing streams

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:

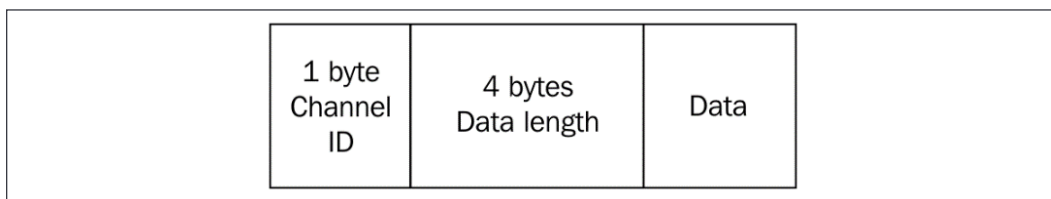


Figure 6.10: Bytes structure of the data packet for our remote logger

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 'child_process'
import { connect } from '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 () {
        let chunk
        // (1)
```

```

    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 takes in, as input, the source streams to be multiplexed and the destination channel, 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.
2. When a chunk is read, we wrap it into a packet that contains, in order, 1 byte (`UInt8`) for the channel ID, 4 bytes (`UInt32BE`) for the packet size, and then the actual data.
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 to be able to multiplex up to 256 different source streams because we only have 1 byte 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 `multiplexChannels()` 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 'fs'  
import { createServer } from 'net'  
  
function demultiplexChannel (source, destinations) {  
  let currentChannel = null  
  let currentLength = null  
  
  source  
    .on('readable', () => {  
      let chunk  
      if (currentChannel === null) {  
        chunk = source.read(1)  
        currentChannel = chunk && chunk.readUInt8(0)  
      }  
  
      if (currentLength === null) {  
        chunk = source.read(4)  
        currentLength = chunk && 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)
    destinations.forEach(destination => 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.
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.
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; let's call it `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 then, on the server, be demultiplexed into two separate files.



Please make a note that, as we are using `child_process.fork()` ([nodejsdp.link/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*:

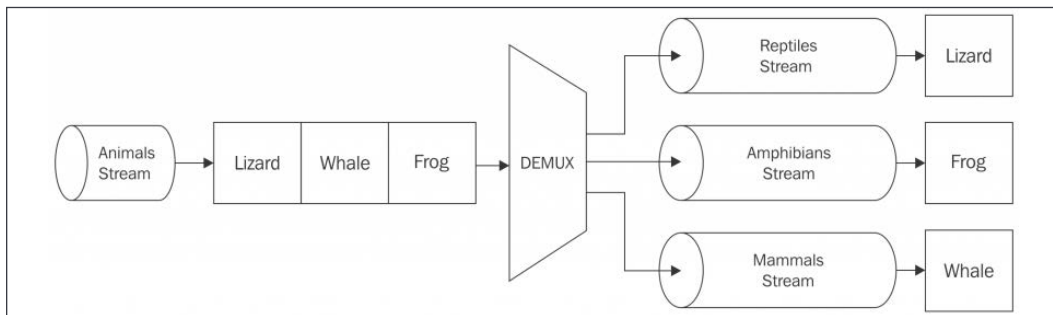


Figure 6.11: Demultiplexing an object stream

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.

Summary

In this chapter, we have 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 data sets, such as the 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

A design pattern is a reusable solution to a recurring problem. The term is really broad in its definition and 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 almost 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 now that 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* classes or abstract 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.

In this chapter and in 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**. 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 the modules within our applications.



This chapter, as well as the following two, 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 has more than just one purpose. Its main advantage is its ability to decouple the creation of an object from one particular implementation. This allows us, for example, to create an object whose class is determined at runtime. Factory also allows us to expose "a surface area" that is much smaller than that of a class; a class can be extended or manipulated, while a factory, being just a function, offers fewer options to the user, making it more robust and easier to understand. Finally, a factory can also be used to enforce encapsulation by leveraging closures.

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 to 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:

```
function createImage (name) {
  if (name.match(/\.(jpe?g$/)) {
    return new ImageJpeg(name)
  } else if (name.match(/\.(gif$/)) {
    return new ImageGif(name)
  } else if (name.match(/\.(png$/)) {
    return new ImagePng(name)
  } else {
    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 not be possible to enforce if `name` was 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 hashbang `#` prefix syntax), introduced in Node.js 12. More on this at nodejsdp.link/tc39-private-fields. This is the most modern approach, but at the time of writing, the feature is still experimental and has yet to be included in the official ECMAScript specification.
- 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 the 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 with the following properties:

- 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 alternatively, 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 abstracting 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.

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 (intNumber) {
  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 = 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
```

While if we run the module in development mode, we will see the profiling information printed to the console:

```
node index.js 2201307499
```

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 having 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 (which argument is what?). 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, 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 what is the parameter that receives each value. 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.

The Builder pattern fixes even these last few flaws and provides a fluent interface that is simple to read, self-documenting, and that provides guidance toward the creation of a coherent object. Let's take a look at the `BoatBuilder` class, which implements the Builder pattern for the `Boat` class:

```
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:

```
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 not an exact rule to 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, 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.



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.

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`).

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 have to keep track of the argument position to know what component of the URL we are passing. Take a 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. 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 pretty 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 what 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).



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, 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. This can be mitigated by adding extra validations, but at the cost of adding more complexity.

In the wild

The Builder pattern is a quite common pattern 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](https://nodejs.org/docs/latest/api/http.html)), we can immediately see it accepts a large amount of options, which is the usual sign that the Builder pattern can potentially provide a better interface. In fact, one of the most popular HTTP(S) request wrappers, `superagent` ([nodejsdp.link/superagent](https://nodejs.org/docs/latest/api/http.html)), 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:

```
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), and have not used the `new` operator. 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.



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).

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 "Gang of Four" book, since it originated directly from the 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
- 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 (**revealed members** (3)).

For the pattern to work, the revealed functionality must otherwise be not accessible by 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 and 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 if 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 (mainly 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*.

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 criterions for justifying the Singleton pattern. Therefore, what we usually want is to configure and instantiate one 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:

```
// file '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 instance of the dbInstance module. 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 an identical 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 incompatible 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 in an attempt to fix the version incompatibility.

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 break 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.

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 a 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:

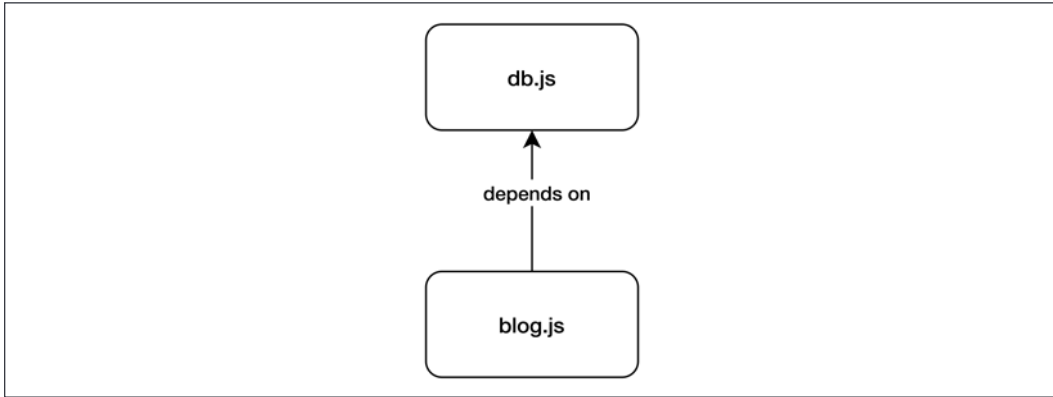


Figure 7.1: Dependency graph between the blog module and the database module

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 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 (the file `db.js`):

```

import { dirname, join } from 'path'
import { fileURLToPath } from 'url'
import sqlite3 from 'sqlite3'
const __dirname = dirname(fileURLToPath(import.meta.url))
export const db = new sqlite3.Database(
  join(__dirname, 'data.sqlite'))
  
```

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 module `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 { promisify } from 'util'
import { db } from './db.js'

const dbRun = promisify(db.run.bind(db))
const dbAll = promisify(db.all.bind(db))

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 dbRun(initQuery)
  }

  createPost (id, title, content, createdAt) {
    return dbRun('INSERT INTO posts VALUES (?, ?, ?, ?)',
      id, title, content, createdAt)
  }

  getAllPosts () {
    return dbAll('SELECT * FROM posts ORDER BY created_at DESC')
  }
}
```

The `blog.js` module exports a class called `Blog` containing three methods:

- `initialize()`: Creates the `posts` table if it doesn't exist. The table will be used to store the blog post's data.
- `createPost()`: 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()`: Retrieves all the posts available in the database and returns them as an array.

Now, let's create a module to try out the functionality of the blog module we just created (the file `index.js`):

```
import { Blog } from './blog.js'

async function main () {
  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)
  }
}

main().catch(console.error)
```

This 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 for 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. But, as you can image, 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 pattern**.

Dependency Injection (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, each dependency, instead of being hardcoded into the module, 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:

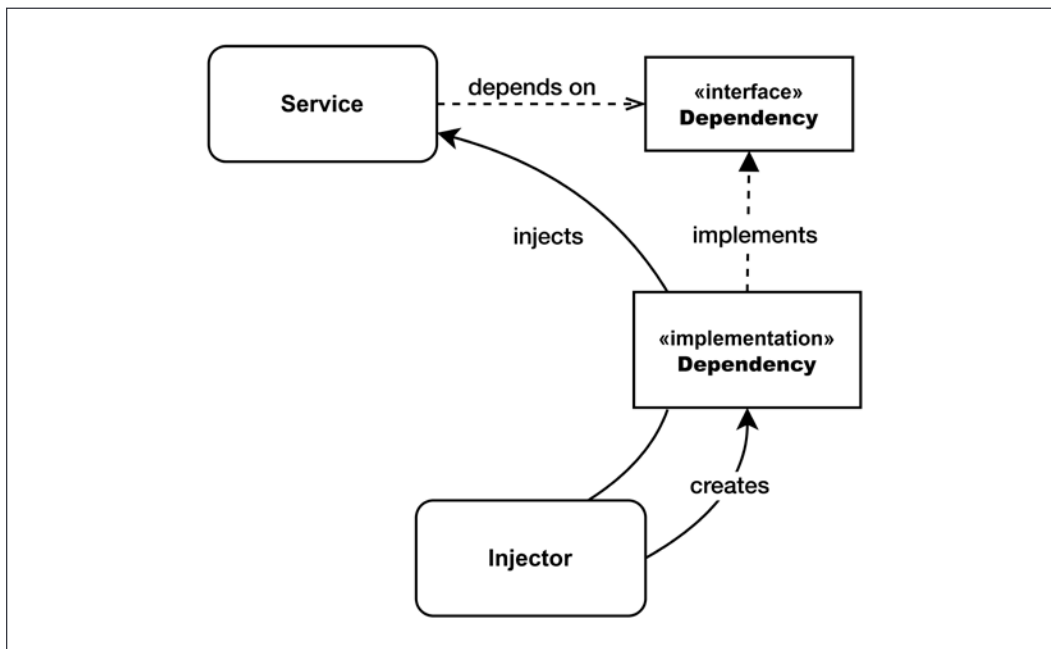


Figure 7.2: Dependency injection schematic

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:

```

import { promisify } from 'util'

export class Blog {
  constructor (db) {
    this.db = db
    this.dbRun = promisify(db.run.bind(db))
    this.dbAll = promisify(db.all.bind(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.dbRun(initQuery)
  }

  createPost (id, title, content, createdAt) {
    return this.dbRun('INSERT INTO posts VALUES (?, ?, ?, ?)',
      id, title, content, createdAt)
  }

  getAllPosts () {
    return this.dbAll(
      '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 sqlite3 from 'sqlite3'

export function createDb (dbFile) {
  return new sqlite3.Database(dbFile)
}

```


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 files if we have 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 { dirname, join } from 'path'
import { fileURLToPath } from 'url'
import { Blog } from './blog.js'
import { createDb } from './db.js'

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

async function main () {
  const db = createDb(join(__dirname, 'data.sqlite'))
  const blog = new Blog(db)
  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)
  }
}

main().catch(console.error)
```

This code is also 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 Dependency Injection 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 amount 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, Dependency Injection 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 **dependency injection 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 on 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 gently 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 thought up by the *Gang of Four* 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 take into account.

Among the patterns you've learned in this chapter, the Builder pattern may seem 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 Dependency Injection. 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 just 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 piece of information that we can exploit 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](#) 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 will explore some of the most popular structural design patterns and discover how they apply to Node.js. Structural design patterns are focused on providing ways to realize relationships between entities.

In particular, in this chapter, we will examine the following patterns:

- **Proxy:** A pattern that allows us to control access to another object
- **Decorator:** A common pattern to augment the behavior of an existing object dynamically
- **Adapter:** A pattern that allows us to access the functionality of an object using a different interface

Throughout the chapter, we will also explore some interesting concepts such as **reactive programming (RP)**, and we will also spend some time playing with LevelDB, a database technology that is commonly adopted in the Node.js ecosystem.

By the end of this chapter, you will be familiar with several scenarios in which structural design patterns can be useful and you will be able to implement them effectively in your 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:

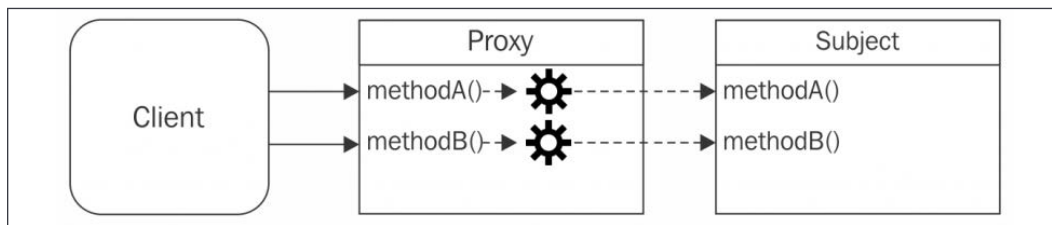


Figure 8.1: Proxy pattern schematic

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
- **Logging:** The proxy intercepts the method invocations and the relative parameters, recoding 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 actually 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 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 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.

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 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](https://nodejsdp.link/delegates) (nodejsdp.link/delegates). 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 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 a **target** and a **handler** as arguments:

```
const proxy = new Proxy(target, handler)
```

Here, **target** represents the object on which the proxy is applied (the **subject** for 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 function () {
        // additional validation logic
        const divisor = target.peekValue()
        if (divisor === 0) {
          throw 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 up 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 is not implementing 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 old browsers or old versions of Node.js that don't support the Proxy object directly.



Transpilation: Short for *transcompilation*. It indicates 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 have to 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 properties can be delegated using `Object.defineProperty()`. Find out more at nodejsdp.link/define-prop.

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 situation where composition is almost necessary; this is when we want to control the initialization of the subject, for example, to create it only when needed (*lazy initialization*).

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 function (...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 '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, which 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.
- Version 3 of 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 in 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 described in *Figure 8.2*:

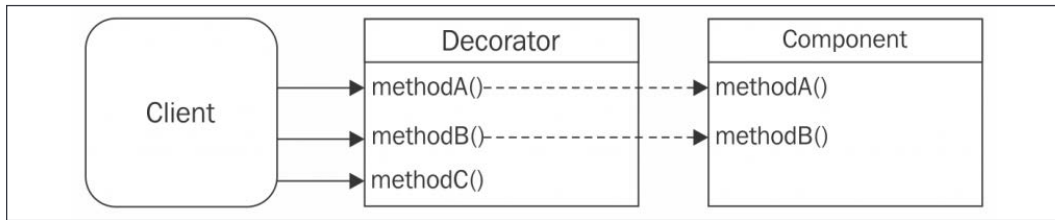


Figure 8.2: Decorator pattern schematic

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 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 augmentation

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 = function () {  
    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 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
      }
    } else if (property === 'divide') {
      // modified method
      return function () {
        // additional validation logic
        const divisor = target.peekValue()
        if (divisor === 0) {
          throw 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!

Decorating a LevelUP database

Before we start coding the next example, let's say a few words about **LevelUP**, the module that we are now going to work with.

Introducing LevelUP and LevelDB

LevelUP (nodejsdp.link/levelup) is a Node.js wrapper around Google's **LevelDB**, a key-value store originally built to implement IndexedDB in the Chrome browser, but it's much more than that. LevelDB has been defined as the "Node.js of databases" because of its minimalism and extensibility. Like Node.js, LevelDB provides blazingly fast performance and only the most basic set of features, allowing developers to build any kind of database on top of it.

The Node.js community, and in this case Rod Vagg, did not miss the chance to bring the power of this database into the Node.js world by creating LevelUP. Born as a wrapper for LevelDB, it then evolved to support several kinds of backends, from in-memory stores, to other NoSQL databases such as Riak and Redis, to web storage engines such as IndexedDB and localStorage, allowing us to use the same API on both the server and the client, opening up some really interesting scenarios.

Today, there is a vast ecosystem around LevelUP 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 LevelUP, 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 LevelUP ecosystem at nodejsdp.link/awesome-level.

Implementing a LevelUP plugin

In the next example, we are going to show you how we can create a simple plugin for LevelUP 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.



For convenience, we are going to use the `level` package (nodejsdp.link/level), which bundles both `levelup` and the default adapter called `leveldown`, which uses LevelDB as the backend.

What we want to build is a plugin for LevelUP 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 `level1-subscribe.js`. We will then insert the following code:

```
export function levelSubscribe (db) {
  db.subscribe = (pattern, listener) => {           // (1)
    db.on('put', (key, val) => {                     // (2)
      const match = Object.keys(pattern).every(
        k => (pattern[k] === val[k])                 // (3)
      )
      if (match) {
        listener(key, val)                           // (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 `put` 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 { dirname, join } from 'path'
import { fileURLToPath } from 'url'
import level from 'level'
import { levelSubscribe } from './level-subscribe.js'

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

const dbPath = join(__dirname, 'db')
const db = level(dbPath, { valueEncoding: 'json' }) // (1)
levelSubscribe(db) // (2)

db.subscribe( // (3)
  { doctype: 'tweet', language: 'en' },
  (k, val) => console.log(val)
)
db.put('1', { // (4)
  doctype: 'tweet',
  text: 'Hi',
  language: 'en'
})
db.put('2', {
  doctype: 'company',
  name: 'ACME Co.'
})
```

This is how the preceding code works:

1. First, we initialize our LevelUP 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: 'tweet' and language: 'en'.

4. Finally, we save some values in the database using `put`. The first call triggers the callback associated with our 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.

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 undoubted power if used appropriately.



For simplicity, our plugin works only in combination with `put` operations, but it can be easily expanded to work even with `batch` operations (nodejsdp.link/levelup-batch).

In the wild

For more examples of how decorators are used in the real world, you can inspect the code of some more LevelUP plugins:

- `level-inverted-index` (nodejsdp.link/level-inverted-index): This is a plugin that adds inverted indexes to a LevelUP database, allowing us to perform simple text searches across the values stored in the database
- `levelplus` (nodejsdp.link/levelplus): This is a plugin that adds atomic updates to a LevelUP database

Aside from LevelUP 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 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:

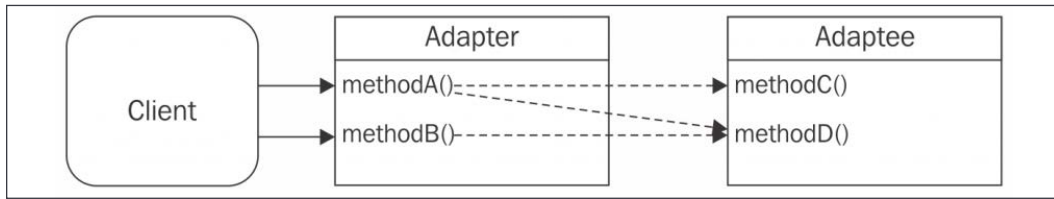


Figure 8.3: Adapter pattern schematic

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 pretty straightforward, so let's immediately work on an example.

Using LevelUP through the filesystem API

We are now going to build an adapter around the LevelUP API, transforming it into an interface that is compatible with the core `fs` 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 LevelUP database as a storage backend for simple filesystem operations.

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 'path'

export function createFsAdapter (db) {
  return ({
    readFile (filename, options, callback) {
      // ...
    },
    writeFile (filename, contents, options, callback) {
      // ...
    }
  })
}
```


Next, we will implement the `readFile()` function inside the factory and ensure that its interface is compatible with the one of the original function from the `fs` module:

```
readFile (filename, options, callback) {
  if (typeof options === 'function') {
    callback = options
    options = {}
  } else if (typeof options === 'string') {
    options = { encoding: options }
  }

  db.get(resolve(filename), {                                     // (1)
    valueEncoding: options.encoding
  },
  (err, value) => {
    if (err) {
      if (err.type === 'NotFoundError') {                       // (2)
        err = new Error(`ENOENT, open "${filename}"`)
        err.code = 'ENOENT'
        err.errno = 34
        err.path = filename
      }
      return callback && callback(err)
    }
    callback && callback(null, value)                             // (3)
  })
}
```

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 `fs.readFile()` function. The steps performed by the function are described as follows:

1. 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 an input.
2. If the key is not found in the database, we create an error with `ENOENT` as the error code, which is the code used by the original `fs` module to indicate a missing file. Any other type of error is forwarded to `callback` (for the scope of this example, we are adapting only the most common error condition).
3. If the key-value pair is retrieved successfully from the database, we will return the value to the caller using the `callback`.

The function that we created does not want to be a perfect replacement for the `fs.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:

```
writeFile (filename, contents, options, callback) {
  if (typeof options === 'function') {
    callback = options
    options = {}
  } else if (typeof options === 'string') {
    options = { encoding: options }
  }

  db.put(resolve(filename), contents, {
    valueEncoding: options.encoding
  }, callback)
}
```

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. If we now write a small test module, we can try to use it:

```
import fs from 'fs'

fs.writeFile('file.txt', 'Hello!', () => {
  fs.readFile('file.txt', { encoding: 'utf8' }, (err, res) => {
    if (err) {
      return console.error(err)
    }
    console.log(res)
  })
})

// try to read a missing file
fs.readFile('missing.txt', { encoding: 'utf8' }, (err, res) => {
  console.error(err)
})
```

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:

```
Error: ENOENT, open "missing.txt"
Hello!
```

Now, we can try to replace the `fs` module with our adapter, as follows:

```
import { dirname, join } from 'path'
import { fileURLToPath } from 'url'
import level from 'level'
import { createFSAdapter } from './fs-adapter.js'

const __dirname = dirname(fileURLToPath(import.meta.url))
const db = level(join(__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 LevelUP 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 LevelUP itself has adapters that enable the database to also run in the browser. One of these adapters is `level-js` (nodejsdp.link/level-js). 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. We will soon realize that Adapter is an extremely important pattern when it comes to sharing code with the browser, as we will see in more detail in *Chapter 10, Universal JavaScript for Web Applications*.

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 LevelUP is able to 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) LevelUP API. Take a look at some of them to see how they are implemented at nodejsdp.link/level-stores.
- JugglingDB is a multi-database ORM and of course, multiple adapters are used to make it compatible with different databases. Take a look at some of them at nodejsdp.link/jugglingdb-adapters.
- nanoSQL (nodejsdp.link/nanosql) is a modern multi-model database abstraction library that makes heavy usage of the Adapter pattern to support a significant variety of databases.
- The perfect complement to the example that we created is `level-fs` (nodejsdp.link/level-fs), which is the proper implementation of the `fs` API on top of LevelUP.

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).
- **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 have learned patterns that aid us in the creation of objects and with building complex object structures. Now it's time to move onto another aspect of software design, which concerns the behavior of components. In this chapter, we will learn how to combine objects and how to define the way they communicate so that the behavior of the resulting structure becomes extensible, modular, reusable, and adaptable. Problems such as "How do I change parts of an algorithm at runtime?", "How can I change the behavior of an object based on its state?", and "How can I iterate over a collection without knowing its implementation?" are the typical kinds of problems solved by the patterns presented in this chapter.

You've already met 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 witness once again how the implementation of some of those patterns can be radically different in JavaScript compared to a purer object-oriented approach. A great example of this thesis can be found in the Iterator pattern, which you will meet later in the chapter. To implement the Iterator pattern, in fact, we won't need to extend any class or build any complex hierarchy. Instead, we will just need to add a special method to a class. Moreover, one particular pattern in this chapter, Middleware, tightly resembles another popular GoF pattern, which is the Chain of Responsibility pattern, but its implementation in Node.js has become such a standard that it can be considered a pattern of its own.

Now, it's time to roll up your sleeves and get your 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 enables an object, called the **context**, to support variations in its logic by extracting the *variable* parts into separate, interchangeable objects called **strategies**. The context implements the common logic of a family of algorithms, while a strategy implements the mutable parts, allowing the context to adapt its behavior depending on different factors, such as an input value, a system configuration, or user preferences.

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:

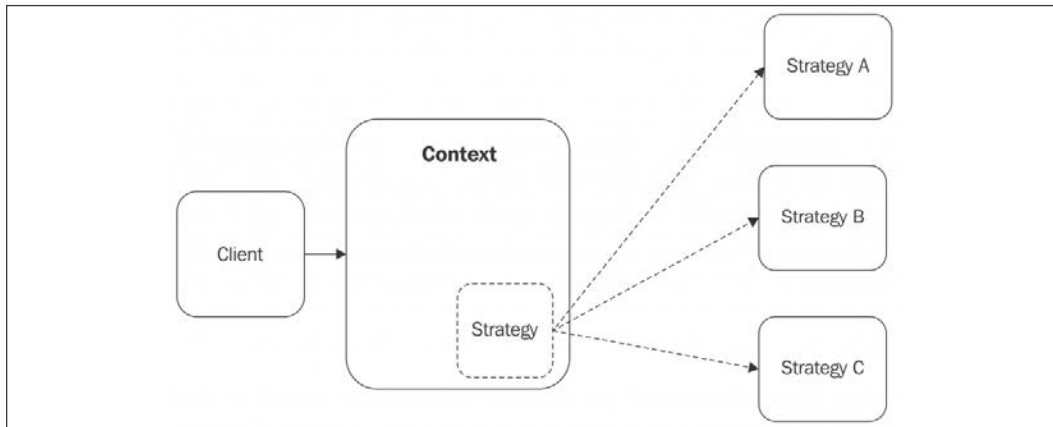


Figure 9.1: General structure of the Strategy pattern

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. 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, such as the database URL, the listening port of the server, and so on. 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, INI, or YAML.

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.js`, and let's define the *generic* part of our configuration manager:

```
import { promises as fs } from 'fs'
import objectPath from 'object-path'

export class Config {
  constructor (formatStrategy) { // (1)
    this.data = {}
    this.formatStrategy = formatStrategy
  }

  get (configPath) { // (2)
    return objectPath.get(this.data, configPath)
  }

  set (configPath, value) { // (2)
```

```

    return objectPath.set(this.data, configPath, value)
  }
  async load (filePath) {                                     // (3)
    console.log(`Deserializing from ${filePath}`)
    this.data = this.formatStrategy.deserialize(
      await fs.readFile(filePath, 'utf-8')
    )
  }

  async save (filePath) {                                     // (3)
    console.log(`Serializing to ${filePath}`)
    await fs.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. Then we also store `formatStrategy`, which represents the component that we will use to parse and serialize the data.
2. We provide two methods, `set()` and `get()`, to access the configuration properties using a dotted path notation (for example, `property.subProperty`) by leveraging a library called `object-path` (nodejsdp.link/object-path).
3. The `load()` and `save()` methods are where we delegate, respectively, the deserialization and serialization of the data to our strategy. This is where the logic of the `Config` class is altered based on the `formatStrategy` passed as an input in the constructor.

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.js`. Let's start with a strategy for parsing and serializing data using the INI file format, which is a widely used configuration format (more info about it here: nodejsdp.link/ini-format).

For the task, we will use an npm package called `ini` (nodejsdp.link/ini):

```
import ini from 'ini'

export const iniStrategy = {
  deserialize: data => ini.parse(data),
  serialize: data => ini.stringify(data)
}
```

Nothing really complicated! Our strategy simply implements the agreed interface, so that it can be used by the `Config` object.

Similarly, the next strategy that we are going to create allows us to support the JSON file format, widely used in JavaScript and in the web development ecosystem in general:

```
export const jsonStrategy = {
  deserialize: data => JSON.parse(data),
  serialize: data => JSON.stringify(data, null, '  ')
}
```

Now, to show you how everything comes together, let's create a file named `index.js`, and let's try to load and save a sample configuration using different formats:

```
import { Config } from './config.js'
import { jsonStrategy, iniStrategy } from './strategies.js'

async function main () {
  const iniConfig = new Config(iniStrategy)
  await iniConfig.load('samples/conf.ini')
  iniConfig.set('book.nodejs', 'design patterns')
  await iniConfig.save('samples/conf_mod.ini')

  const jsonConfig = new Config(jsonStrategy)
  await jsonConfig.load('samples/conf.json')
  jsonConfig.set('book.nodejs', 'design patterns')
  await jsonConfig.save('samples/conf_mod.json')
}

main()
```

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 a 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 tradeoff 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 (nodejsdp.link/passportjs) is an authentication framework for Node.js, which allows a web server to support different authentication schemes. With Passport, we can provide a *login with Facebook* or *login with Twitter* functionality to our web application with minimal effort. Passport uses the Strategy pattern to separate the common logic used during an authentication process from the parts that can change, namely the actual authentication step. For example, we might want to use OAuth in order to obtain an access token to access a Facebook or Twitter profile, or simply use a local database to verify a username/password pair. For Passport, these are all different strategies for completing the authentication process and, as we can imagine, this allows the library to support a virtually unlimited number of authentication services. Take a look at the number of different authentication providers supported at nodejsdp.link/passport-strategies to get an idea of what the Strategy pattern can do.

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:

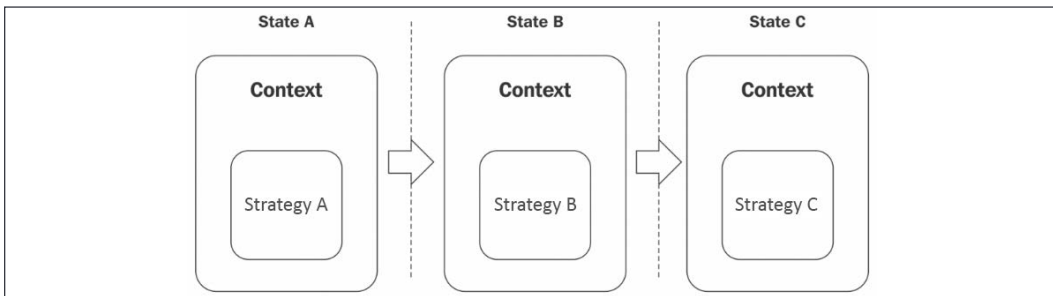


Figure 9.2: The State pattern

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 have to 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.

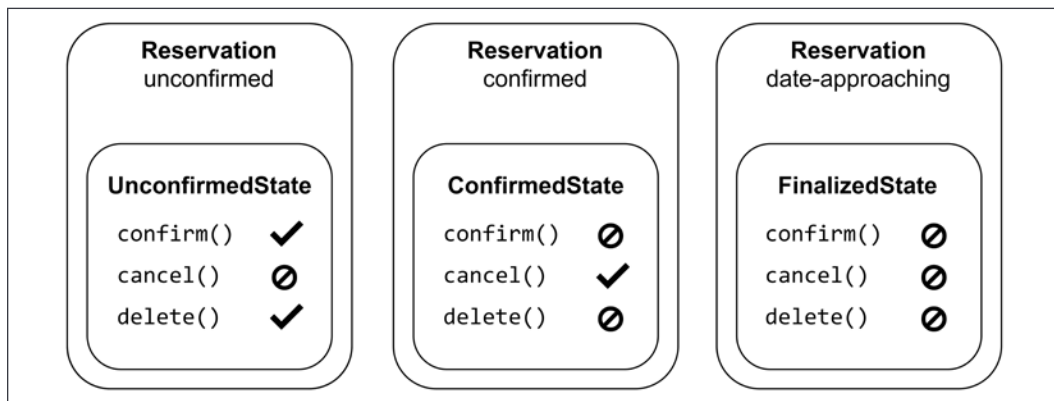


Figure 9.3: An example application of the State pattern

As illustrated in *Figure 9.3*, the State pattern is, instead, 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 jsonOverTcp from 'json-over-tcp-2'           // (1)

export class OfflineState {
  constructor (failsafeSocket) {
    this.failsafeSocket = failsafeSocket
  }

  send (data) {                                     // (2)
    this.failsafeSocket.queue.push(data)
  }

  activate () {                                     // (3)
    const retry = () => {
      setTimeout(() => this.activate(), 1000)
    }

    console.log('Trying to connect...')
    this.failsafeSocket.socket = jsonOverTcp.connect(
      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. Instead of using a raw TCP socket, we will use a little library called `json-over-tcp-2` (nodejsdp.link/json-over-tcp-2). This will greatly simplify our work since the library will take care of all the parsing and formatting of the data going through the socket into JSON objects.
2. 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.
3. The `activate()` method tries to establish a connection with the server using the `json-over-tcp-2` socket. 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
    this.hasDisconnected = false
  }
  send (data) { // (1)
    this.failsafeSocket.queue.push(data)
    this._safeWrite(data)
  }
  _safeWrite (data) { // (2)
    this.failsafeSocket.socket.write(data, (err) => {
      if (!this.hasDisconnected && !err) {
        this.failsafeSocket.queue.shift()
      }
    })
  }
}
```

```

activate () {                                     // (3)
  this.hasDisconnected = false
  for (const data of this.failSafeSocket.queue) {
    this._safeWrite(data)
  }

  this.failSafeSocket.socket.once('error', () => {
    this.hasDisconnected = true
    this.failSafeSocket.changeState('offline')
  })
}
}

```

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 write it directly into the socket, as we assume that we are online. It'll use the internal `_safeWrite()` method to do that.
2. The `_safeWrite()` method tries to write the data into the socket writable stream (see the official docs at [nodejsdp.link/writable-write](https://nodejs.org/docs/latest/api/writable)) and it waits for the data to be written into the underlying resource. If no errors are returned and if the socket didn't disconnect in the meantime, it means that the data was sent successfully and therefore we remove it from the queue.
3. The `activate()` method flushes any data that was queued while the socket was offline and it also starts listening for any error event; we will take this as a symptom that the socket went offline (for simplicity). When this happens, we transition to the offline state.

That's it for our `FailsafeSocket`. Now we are ready to build a sample client and a server to try it out. Let's put the server code in a module named `server.js`:

```

import jsonOverTcp from 'json-over-tcp-2'

const server = jsonOverTcp.createServer({ port: 5000 })
server.on('connection', socket => {
  socket.on('data', data => {
    console.log('Client data', data)
  })
})

```

```
server.listen(5000, () => console.log('Server started'))
```

Then, the client-side code, which is what we are really interested in, goes into `client.js`:

```
import { FailsafeSocket } from './failsafeSocket.js'

const failsafeSocket = new FailsafeSocket({ port: 5000 })

setInterval(() => {
  // send current memory usage
  failsafeSocket.send(process.memoryUsage())
}, 1000)
```

Our server simply prints to the console any JSON message it receives, while our clients are sending a measurement of their memory utilization every second, leveraging a `FailsafeSocket` object.

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 doesn't want 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*.

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 intent 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:

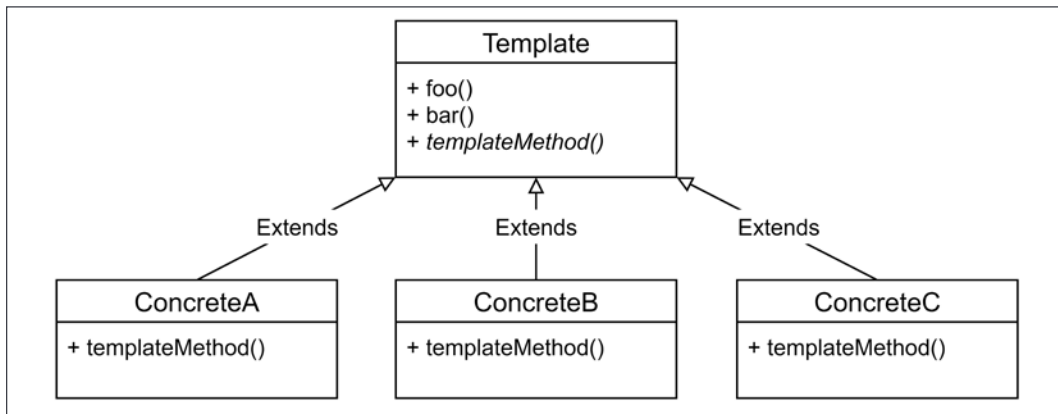


Figure 9.4: UML diagram of the Template pattern

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 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 an exception, indicating that the method has to be implemented. 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 Template and Strategy 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 has to 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 { promises as fsPromises } from 'fs'
import objectPath from 'object-path'

export class ConfigTemplate {
  async load (file) {
    console.log(`Deserializing from ${file}`)
    this.data = this._deserialize(
      await fsPromises.readFile(file, 'utf-8'))
  }

  async save (file) {
    console.log(`Serializing to ${file}`)
    await fsPromises.writeFile(file, this._serialize(this.data))
  }

  get (path) {
    return objectPath.get(this.data, path)
  }

  set (path, value) {
    return objectPath.set(this.data, path, value)
  }

  _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, for example, one that allows us to load and save the configuration using the JSON format:

```
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 an `IniConfig` class supporting the `.ini` file format using the same template class:

```
import { ConfigTemplate } from './configTemplate.js'
import ini from 'ini'

export class IniConfig extends ConfigTemplate {
  _deserialize (data) {
    return ini.parse(data)
  }

  _serialize (data) {
    return ini.stringify(data)
  }
}
```

Now we can use our concrete configuration manager classes to load and save some configuration data:

```
import { JsonConfig } from './jsonConfig.js'
import { IniConfig } from './iniConfig.js'

async function main () {
  const jsonConfig = new JsonConfig()
  await jsonConfig.load('samples/conf.json')
  jsonConfig.set('nodejs', 'design patterns')
  await jsonConfig.save('samples/conf_mod.json')

  const iniConfig = new IniConfig()
  await iniConfig.load('samples/conf.ini')
  iniConfig.set('nodejs', 'design patterns')
  await iniConfig.save('samples/conf_mod.ini')
}

main()
```

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.

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 ECMAScript2015 specification).

The Iterator pattern defines a common interface or protocol for iterating the elements of a container, such as an array 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 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 in advance.

The starting point for implementing the Iterator pattern in JavaScript is the **iterator protocol**, which defines an interface for producing a sequence of values. So, we'll call **iterator** an object implementing a `next()` method having the following behavior: each time the method is called, the function returns the next element in the iteration through 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 meet those in a moment).

Let's use 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 = 65
const Z_CHAR_CODE = 90

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 have to 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 side, 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. However, as we will see later in this section, it's not the only way we have to consume an iterator. JavaScript has, in fact, 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. Such objects are called **iterables**. Usually, an iterable is a container of elements, such as a data structure, but 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 relative 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  
  }  
}
```

To show how this works in practice, let's build a class to manage information organized in a bidimensional matrix structure. We want this class to be implementing 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', '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) as well as syntax constructs accepting iterables as an 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.



Note that all the APIs and syntax constructs we've just seen accept as input an iterable and not an iterator. But, what can we do if we have a function returning an iterator, such as in our `createAlphabetIterator()` example? How can we leverage all the built-in APIs and syntax constructs? A possible solution is implementing the `@@iterator` method in the iterator object itself, which will simply return the iterator object itself. This way we'll be able to write something such as the following:

```
for (const letter of createAlphabetIterator()) {  
  //...  
}
```

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` and `Set`, and even `String` all implement the `@@iterator` method. In Node.js land, `Buffer` is probably the most notable iterable.



A trick to make sure that an array doesn't contain duplicate elements is the following: `const uniqArray = Array.from(new Set(arrayWithDuplicates))`. This also shows us how iterables offer a way for different components to talk to each other using a shared interface.

Generators

The ES2015 specification introduced a syntax construct that is closely related to iterators. We are talking about **generators**, also known as **semicoroutines**. They are a generalization of standard functions, in which there can be different entry points. In a standard function, we can have only one entry point, which corresponds to the invocation of the function itself, but a generator can be suspended (using the `yield` statement) and then resumed at a later time. Among other things, generators are very well suited to implement iterators, in fact, as we will see in a bit, the generator object returned by a generator function is indeed 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 will not execute its body immediately. Rather, it will return a **generator object**, which, as we already mentioned, is both an iterator and an iterable. But it doesn't end here; invoking `next()` on the generator object will start or resume the execution of the generator until the `yield` instruction is invoked or the generator returns (either implicitly or explicitly with a `return` instruction). Within the generator, using the keyword `yield` followed by a value `x` is equivalent to returning `{done: false, value: x}` from the iterator, while returning a value `x` is 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. For 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

Generator objects are more than standard iterators, in fact, their `next()` method optionally accepts an argument (whereas, as specified by the iterator protocol, it does not need to accept one). Such an argument is passed as the return value of the `yield` instruction. To show this, let's create a new simple generator:

```
function * twoWayGenerator () {  
  const what = yield null  
  yield 'Hello ' + what  
}
```

```
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 is invoked, the generator reaches the first `yield` statement and is then put on pause.
2. When `next('world')` is invoked, the generator resumes from the point where it was put on pause, which is on the `yield` instruction, but this time we have a value that is passed back to the generator. This value will then be set to the `what` variable. The generator then appends the `what` variable to the string `'Hello '` and yields 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 what = yield null
    yield 'Hello ' + what
  } 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 was 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 (stay unchanged)

  * [Symbol.iterator] () { // (1)
    let nextRow = 0 // (2)
    let nextCol = 0

    while (nextRow !== this.data.length) { // (3)
      yield this.data[nextRow][nextCol]

      if (nextCol === this.data[nextRow].length - 1) {
        nextRow++
        nextCol = 0
      } else {
        nextCol++
      }
    }
  }
}
```

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 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 a standard loop to iterate over the elements of the matrix. This is certainly more intuitive than trying to imagine a loop that invokes the `next()` method of an iterator.

As we can see, generators are an excellent alternative to writing iterators from scratch. They will improve the readability of our iteration routine and will offer the same level of functionality (or even better).



The **generator delegation** instruction, `yield * iterable`, 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.

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 items that require an asynchronous operation to be produced.

Imagine, for example, to iterate 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, and since that's the only extra requirement, it means that we can also use an `async` function to define the `next()` method of the iterator. 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.

Async iterables can be looped over using the `for await...of` syntax, which can only be used inside an async function. With the `for await...of` syntax, we are essentially implementing a sequential asynchronous execution flow on top of the Iterator pattern. Essentially, it's 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) as, for example, over an array of promises. It will work even if not all the elements (or none) of the iterator are promises.

To quickly demonstrate this, 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`:

```
import superagent from 'superagent'

export class CheckUrls {
  constructor (urls) { // (1)
    this.urls = urls
  }

  [Symbol.asyncIterator] () {
    const urlsIterator = this.urls[Symbol.iterator]() // (2)

    return {
      async next () { // (3)
        const iteratorResult = urlsIterator.next() // (4)
        if (iteratorResult.done) {
          return { done: true }
        }

        const url = iteratorResult.value
        try {
          const checkResult = await superagent // (5)
            .head(url)
            .redirects(2)
          return {
```

```

        done: false,
        value: `${url} is up, status: ${checkResult.status}`
      }
    } catch (err) {
      return {
        done: false,
        value: `${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. Since we now know how to use iterators and iterables, we can say that this list of URLs can be just any iterable.
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 invoking its `@@iterable` method.
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. Note how we can now use the `await` instruction to asynchronously get the result of the HEAD request sent to the current URL.

Now, let's use the `for await...of` syntax we mentioned earlier to iterate over a `CheckUrls` object:

```

import { CheckUrls } from './checkUrls.js'

async function main () {
  const checkUrls = new CheckUrls([
    'https://nodejsdesignpatterns.com',
    'https://example.com',
    'https://mustbedownforsurehopefully.com'
  ])

  for await (const status of checkUrls) {

```

```
        console.log(status)
    }
}

main()
```

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 well 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 superagent
        .head(url)
        .redirects(2)
      yield `${url} is up, status: ${checkResult.status}`
    } catch (err) {
      yield `${url} is down, error: ${err.message}`
    }
  }
}
}

```

Interestingly, using an async generator in place of a bare async iterator allowed us to save 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 we 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'

async function main () {
  const stream = process.stdin.pipe(split())
  for await (const line of stream) {
    console.log(`You wrote: ${line}`)
  }
}

main()

```


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? This, as always, depends on the use case and many other factors; however, to help you with the decision, this is a list of aspects that set the two constructs apart:

- Streams are *push*, meaning that data is pushed into the internal buffers by the stream and then consumed from the buffers. Async iterators are *pull* by default (unless another logic is explicitly implemented by the iterator), meaning that data is only retrieved/produced on demand by the consumer.
- Streams are better suited to process binary data since they natively provide internal buffering and backpressure.
- Streams can be composed using a well-known and streamlined API, `pipe()`, while async iterators do not offer any standardized composition method.



We can iterate 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`.

In the wild

Iterators and, 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 packages `@databases/pg`, `@databases/mysql` and `@databases/sqlite` 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, which can be used to easily iterate over the results of a query. For 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 definitely **Middleware**. Unfortunately, it's also one of the most confusing for the inexperienced, 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 OS 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 relative 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*. In fact, 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:

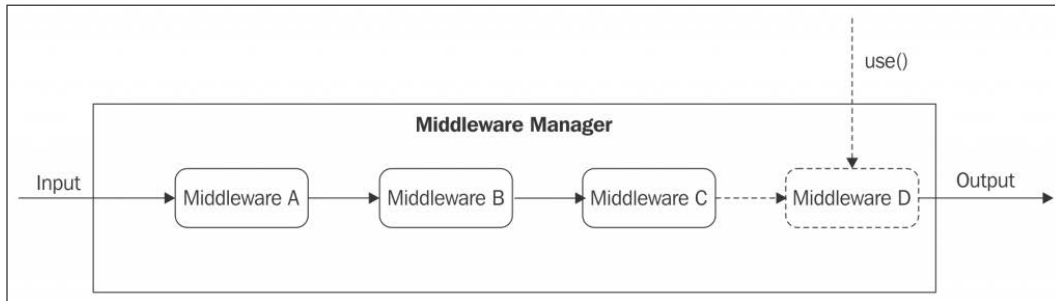


Figure 9.5: The structure of the Middleware pattern

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 return fresh copies as the 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 are 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 {  
  constructor (socket) {                                     // (1)  
    this.socket = socket  
    this.inboundMiddleware = []  
    this.outboundMiddleware = []  
  
    this.handleIncomingMessages()  
      .catch(err => console.error(err))  
  }  
  
  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 our `ZmqMiddlewareManager`:

1. In the first part of the class, we define the constructor that accepts a ZeroMQ socket as an argument. In the constructor, we create two empty lists that will contain our middleware functions, one for inbound messages and another one for outbound messages. Next, we immediately start processing the messages coming from the socket. We do that in the `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 of middlewares.

3. Similarly to `handleIncomingMessages()`, the `send()` method will pass the message received as an argument down the `outboundMiddleware` pipeline. 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 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 back 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 const jsonMiddleware = function () {
  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):

```
import { inflateRaw, deflateRaw } from 'zlib'
import { promisify } from 'util'

const inflateRawAsync = promisify(inflateRaw)
const deflateRawAsync = promisify(deflateRaw)

export const zlibMiddleware = function () {
  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 zeromq from 'zeromq' // (1)
import { ZmqMiddlewareManager } from './zmqMiddlewareManager.js'
import { jsonMiddleware } from './jsonMiddleware.js'
import { zlibMiddleware } from './zlibMiddleware.js'

async function main () {
  const socket = new zeromq.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')
  }

  main()

```

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, in the `main()` function, 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'
import { ZmqMiddlewareManager } from './zmqMiddlewareManager.js'
import { jsonMiddleware } from './jsonMiddleware.js'
import { zlibMiddleware } from './zlibMiddleware.js'

async function main () {
  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')
}

main()
```

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 on a local port. The rest of the middleware setup is exactly 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.

Two other interesting examples are:

- Koa (nodejsdp.link/koa), which is known as the successor of Express. It was created by the same team behind Express and it shares with it its philosophy and main design principles. Koa's middleware is slightly different than that of Express since it uses modern programming techniques such as `async/await` instead of callbacks.
- Middy (nodejsdp.link/middy) is a classic example of the Middleware pattern applied to something different than a web framework. Middy is, in fact, a middleware engine for AWS Lambda functions.

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 at a later time. 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*:

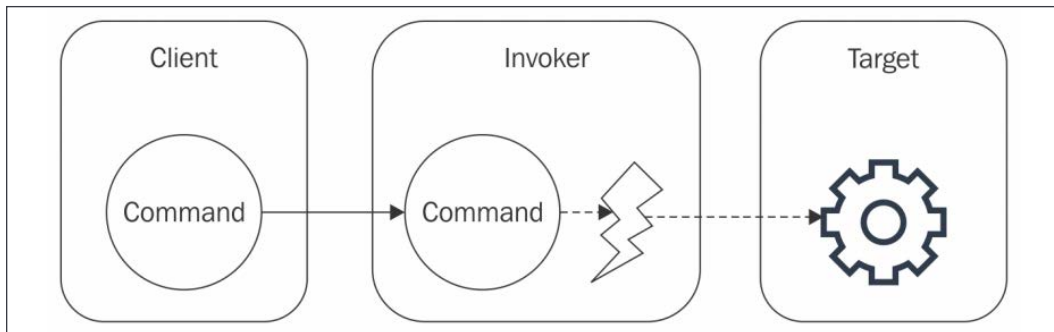


Figure 9.6: The components of the Command pattern

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.

Using the Command pattern instead of directly executing an operation has several applications:

- A command can be scheduled for execution at a later time.
- 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's not yet 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)**, which is the base for most of today's real-time collaborative software, such as collaborative text editing.



A great explanation of how OT works can be found at nodejsdp.link/operational-transformation.

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 off 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, of course, 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 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 Twitter-like service. We will use a mockup of such a service for simplicity (the `statusUpdateService.js` file):

```
const statusUpdates = new Map()  
  
// The Target  
export const statusUpdateService = {  
  postUpdate (status) {
```

```
const id = Math.floor(Math.random() * 1000000)
statusUpdates.set(id, status)
console.log(`Status posted: ${status}`)
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

  // The Command
  return {
    run () {
      postId = service.postUpdate(status)
    },
    undo () {
      if (postId) {
        service.destroyUpdate(postId)
        postId = null
      }
    },
    serialize () {
      return { type: 'status', action: 'post', status: status }
    }
  }
}
```

The preceding function is a factory 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 constructor and its `run()` method (the `invoker.js` file):

```
import superagent from 'superagent'

// The Invoker
export class Invoker {
  constructor () {
    this.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()
  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 superagent  
    .post('http://localhost:3000/cmd')  
    .send({ json: 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 in 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 little example that we have just implemented shows how wrapping an operation in a command can open a world of possibilities, and that's just the tip of the iceberg.

As the last remarks, it is worth noting that a fully-fledged Command pattern should be used only when strictly necessary. We saw, in fact, 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.

Summary

We opened this chapter with three closely related patterns, which are 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, instead, 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 Iterator. 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.

In the next chapter, we'll shift our attention to a topic that goes beyond the boundaries of server-side development. Thanks to Node.js, in fact, we can create "Universal" JavaScript applications, or in other words, applications that can run as seamlessly on the server as they run on the browser. Stay tuned, then, to learn about the most useful Universal JavaScript patterns.

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 a `ConsoleStrategy` to send the messages to the console, or a `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 `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

Universal JavaScript for Web Applications

JavaScript was born with the goal of giving web developers the power to execute code directly on the browser and build dynamic and interactive websites.

Since its inception, JavaScript has grown up a lot. If, at the very beginning, JavaScript was a very simple and limited language, today, it can be considered a complete general-purpose language that can be used even outside the browser to build almost any kind of application. In fact, JavaScript now powers frontend applications, web servers, and mobile applications, as well as embedded devices such as wearable devices, thermostats, and flying drones.

The language's availability across platforms and devices is fostering a new trend among JavaScript developers: being able to simplify code reuse across different environments in the same project. With Node.js, developers have the opportunity to build web applications where it is easy to share code between the server (backend) and the browser (frontend). This quest for code reuse was originally identified with the term **Isomorphic JavaScript**, but today, it's mostly recognized as **Universal JavaScript**.

In this chapter, we are going to explore the wonders of Universal JavaScript, specifically in the field of web development, and discover many tools and techniques we can use to share code between the server and the browser.

We will explore what a module bundler is and why we need one. We will then learn how module bundlers work and we will practice with one of the most popular, webpack. Then, we will discuss some generic patterns that can help us with code reuse across platforms.

Finally, we will learn the basic functionalities of React and we will use it to build a complete Universal JavaScript application that features universal rendering, universal routing, and universal data loading.

To summarize, here's a list of topics we will be covering in this chapter:

- How to share code between the browser and Node.js
- Fundamentals of cross-platform development (code branching, module swapping, and other useful patterns)
- A brief introduction to React
- How to build a complete Universal JavaScript application using React and Node.js

Sit tight, this is going to be an exciting chapter!

Sharing code with the browser

One of the main selling points of Node.js is the fact that it's based on JavaScript and runs on V8, a JavaScript engine that actually powers some of the most popular browsers: Google Chrome and Microsoft Edge. We might think that sharing the same JavaScript engine is enough to make sharing code between Node.js and the browser an easy task; however, as we will see in this chapter, this is not always true, unless we want to share only simple, self-contained, and generic fragments of code.

Developing code for both the client and the server requires a non-negligible level of effort in making sure that the same code can run properly in two environments that are intrinsically different. For example, in Node.js, we don't have the DOM or long-living views, while on the browser, we surely don't have the filesystem and many other interfaces to interact with the underlying operating system.

Another contention point is the level of support for modern JavaScript features. When we target Node.js, we can safely adopt modern language features because we know which Node.js version runs on our servers. For instance, for our server code, we can safely decide to adopt `async/await` if we know it will run on Node.js version 8 (or on a more recent version). Unfortunately, we can't have the same confidence when writing JavaScript code for the browser.

This is because different users will have different browsers with different levels of compatibility with the latest language features. Some users might be using a modern browser with full support for `async/await`, while other users might still be using an old device with an old browser that does not support `async/await`.

So, most of the effort required when developing for both platforms is to make sure to reduce those differences to a minimum. This can be done with the help of abstractions, patterns, and tools that enable the application to switch, dynamically or at build time, between browser-compatible code and Node.js code.

Luckily, with the rising interest in this new mind-blowing possibility, many libraries and frameworks in the ecosystem have started to support both environments. This evolution is also backed by a growing number of tools supporting this new kind of workflow, which, over the years, have been refined and perfected. This means that if we are using an npm package on Node.js, there is a good probability that it will work seamlessly on the browser as well. However, this is often not enough to guarantee that our application can run without problems on both the browser and Node.js. As we will see, a careful design is always needed when developing cross-platform code.

In this section, we are going to explore the fundamental problems we might encounter when writing code for both Node.js and the browser, and we are going to propose some tools and patterns that can help us with tackling this new and exciting challenge.

JavaScript modules in a cross-platform context

The first wall we hit when we want to share some code between the browser and the server is the mismatch between the module system used by Node.js and the heterogeneous landscape of the module systems used on the browser. Another problem is that on the browser, we don't have a `require()` function or the filesystem from which we can resolve modules. Most modern browsers support `import` and ES modules, but again, some of the users visiting our website might not have already adopted one of those modern browsers.

In addition to these problems, we have to take into account the differences in distributing code for the server and the browser. On the server, modules are loaded directly from the filesystem. This is generally a performant operation and therefore developers are encouraged to split their code into small modules to keep the different logic units small and organized.

On the browser, the script loading model is totally different. The process generally starts with the browser downloading an HTML page from a remote endpoint. The HTML code is parsed by the browser, which might find references to script files that need to be downloaded and executed. If we are dealing with a large application, there might be many scripts to download, so the browser will have to issue a significant number of HTTP requests and download and parse multiple script files before the application can be fully initialized. The higher the number of script files, the larger the performance penalty that we will have to pay to run an application on the browser, especially on slow networks. Even though some of this performance penalty can be mitigated with the adoption of **HTTP/2 Server Push** (nodejsdp.link/http2-server-push), client-side caching, preloading, or similar techniques, the underlying problem still stands: having to receive and parse a large number of files is generally worse than having to deal with a few optimized files.

A common practice to address this problem is to "build" packages (or **bundles**) for the browser. A typical build process will collate all the source files into a very small number of bundles (for instance, one JavaScript file per page) so that the browser won't have to download a huge number of scripts for each page visit. A build process is not limited to just reducing the number of files, in fact, it can perform other interesting optimizations. Another common optimization is *code minification*, which allows us to reduce the number of characters to a minimum without altering the functionality. This is generally done by removing comments, removing unused code, and renaming function and variable names.

Module bundlers

If we want to write large portions of code that can work as seamlessly as possible both on the server and on the browser, we need a tool to help us with "bundling" all the dependencies together at build time. These tools are generally called **module bundlers**. Let's visualize this with an example of how shared code can be loaded on to the server and the client using a module bundler:

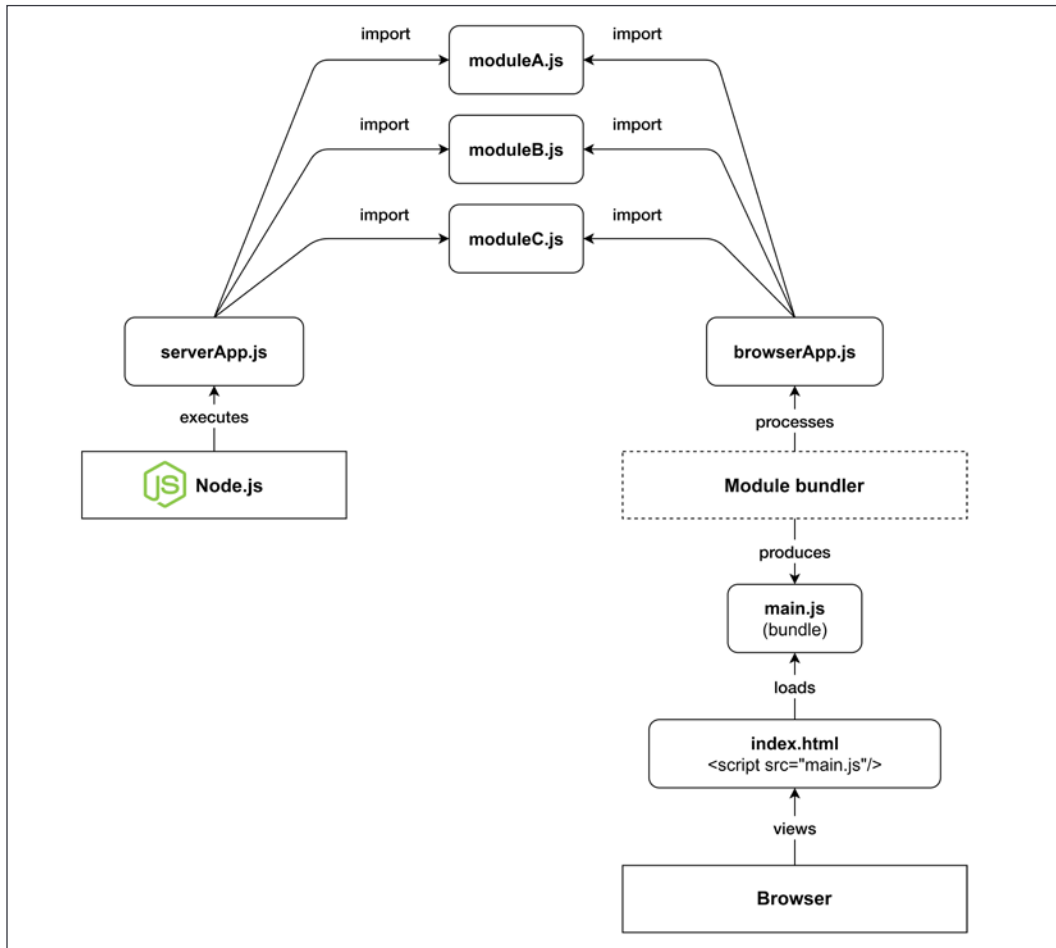


Figure 10.1: Loading shared modules on the server and on the browser (using a module bundler)

By looking at *Figure 10.1*, we can see that the code is processed and loaded differently on the server side and on the browser:

- **On the server side:** Node.js can directly execute our `serverApp.js`, which, in turn, will import the modules `moduleA.js`, `moduleB.js`, and `moduleC.js`.
- **On the browser:** We have `browserApp.js`, which also imports `moduleA.js`, `moduleB.js`, and `moduleC.js`. If our index file were to include `browserApp.js` directly, we would have to download a total of five files (`index.html`, `browserApp.js`, and the three dependency modules) before the app would be fully initialized. The module bundler allows us to reduce the total number of files to only two by preprocessing `browserApp.js` and all its dependencies and producing a single equivalent bundle called `main.js`, which is then referenced by `index.html` and therefore loaded by the browser.

To summarize, on the browser, we generally have to deal with two logical phases, build and runtime, while on the server, we generally don't need a build phase and we can execute our source code directly.

When it comes to picking a module bundler, the most popular option is probably **webpack** (nodejsdp.link/webpack). Webpack is one of the most complete and mature module bundlers currently available and it is the one we are going to use in this chapter. It's worth mentioning, though, that there is a quite prosperous ecosystem full of alternatives, each one with its own strengths. If you are curious, here are some of the most well-known alternatives to webpack:

- **Parcel** (nodejsdp.link/parcel): Aims to be fast and to work "auto-magically" without any configuration.
- **Rollup** (nodejsdp.link/rollup): One of the first module bundlers to fully support ESM and to offer a number of optimizations like tree shaking and dead code elimination.
- **Browserify** (nodejsdp.link/browserify): The first module bundler with support for CommonJS and is still widely adopted.

Other trending module bundlers are **FuseBox** (nodejsdp.link/fusebox), **Brunch** (nodejsdp.link/brunch), and **Microbundle** (nodejsdp.link/microbundle).

In the next section, we will discuss in greater detail how a module bundler works.

How a module bundler works

We can define a module bundler as a tool that takes the source code of an application (in the form of an entry module and its dependencies) and produces one or more bundle files. The bundling process doesn't change the business logic of the app; it just creates files that are optimized to run on the browser. In a way, we can think of a bundler as a compiler for the browser.

In the previous section, we saw how a bundler can help to reduce the total number of files that the browser will need to load, but in reality, a bundler can do so much more than that. For instance, it can use a **transpiler** like **Babel** (nodejsdp.link/babel).

A transpiler is a tool that processes the source code and makes sure that modern JavaScript syntax is converted into equivalent ECMAScript 5 syntax so that a large variety of browsers (including older ones) can run the application correctly. Some module bundlers allow us to preprocess and optimize not just JavaScript code but also other assets such as images and stylesheets.

In this section, we will provide a simplified view of how a module bundler works and how it navigates the code of a given application to produce an equivalent bundle optimized for the browser. The work of a module bundler can be divided into two steps that we will call **dependency resolution** and **packing**.

Dependency resolution

The dependency resolution step has the goal of traversing the codebase, starting from the main module (also called the **entry point**), and discovering all the dependencies. The way a bundler can do this is by representing dependencies as an acyclic direct graph, known as a **dependency graph**.

Let's explore this concept with an example: a fictional calculator application. The implementation is intentionally incomplete as we only want to focus on the module structure, how the different modules depend on each other, and how the module bundler can build the dependency graph of this application:

```
// app.js (1)
import { calculator } from './calculator.js'
import { display } from './display.js'
display(calculator('2 + 2 / 4'))
```

```
// display.js (5)
export function display () {
  // ...
}
```

```
// calculator.js (2)
import { parser } from './parser.js'
import { resolver } from './resolver.js'
export function calculator (expr) {
  return resolver(parser(expr))
}
```

```
// parser.js (3)
export function parser (expr) {
  // ...
}
```

```
// resolver.js (4)
export function resolver (tokens) {
  // ...
}
```

Let's see how the module bundler will walk through this code to figure out the dependency graph:

1. The module bundler starts its analysis from the entry point of the application, the module `app.js`. In this phase, the module bundler will discover dependencies by looking at `import` statements. The bundler starts to scan the code of the entry point and the first `import` it finds references the `calculator.js` module. Now, the bundler suspends the analysis of `app.js` and jumps immediately into `calculator.js`. The bundler will keep tabs on the open files: it will remember that the first line of `app.js` was already scanned so that when it eventually restarts processing this file, it will continue from the second line.
2. In `calculator.js`, the bundler immediately finds a new import for `parser.js` so that the processing of `calculator.js` is interrupted to move into `parser.js`.
3. In `parser.js`, there's no `import` statement, so after the file has been scanned entirely, the bundler goes back into `calculator.js`, where the next `import` statement refers to `resolver.js`. Again, the analysis of `calculator.js` is suspended and the bundler jumps immediately into `resolver.js`.

4. The module `resolver.js` does not contain any imports, so the control goes back to `calculator.js`. The `calculator.js` module does not contain other imports, so the control goes back to `app.js`. In `app.js`, the next import is `display.js` and the bundler jumps straight into it.
5. `display.js` does not contain any imports. So, again the control goes back to `app.js`. There are no more imports in `app.js`, so the code has been fully explored, and the dependency graph has been fully constructed.

Every time the module bundler jumps from one file to another, it means we are discovering a new dependency and adding a new node to the dependency graph. A visual representation of the steps described in the preceding list can be found in *Figure 10.2*:

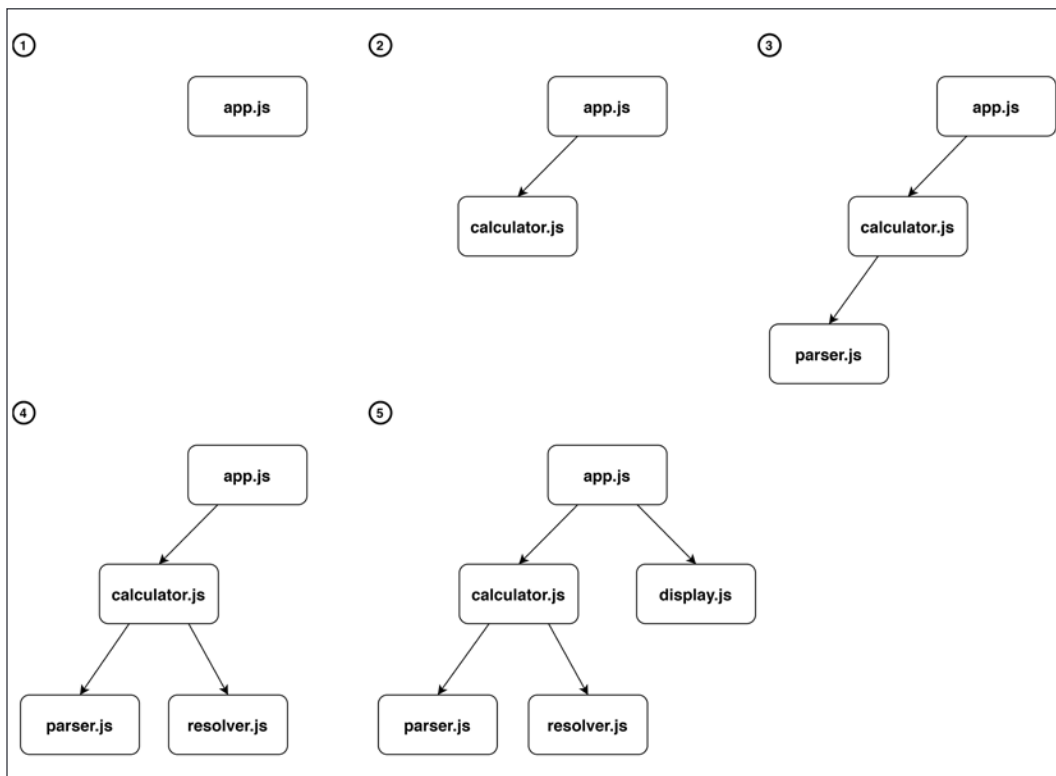


Figure 10.2: Dependency graph resolution

This way of resolving dependencies also works with cyclic dependencies. In fact, if the bundler encounters the same dependency for a second time, the dependency will be skipped because it's already present in the dependency graph.



Tree shaking

It's worth noting that if we have entities (for example, functions, classes, or variables) in our project modules that are never imported, then these won't appear in this dependency graph, so they won't be included in the final bundle.

A more advanced module bundler could also keep track of the entities imported from every module and the exported entities found in the dependency graph. This allows the bundle to figure out if there are exported functionalities that are never used in the application so that they can be pruned from the final bundle. This optimization technique is called **tree shaking** (nodejsdp.link/tree-shaking).

During the dependency resolution phase, the module bundler builds a data structure called **modules map**. This data structure is a hash map that has unique module identifiers (for example, file paths) as keys and a representation of the module source code as values. In our example, a simplified representation of the modules map might look like this:

```
{
  'app.js': (module, require) => { /* ... */ },
  'calculator.js': (module, require) => { /* ... */ },
  'display.js': (module, require) => { /* ... */ },
  'parser.js': (module, require) => { /* ... */ },
  'resolver.js': (module, require) => { /* ... */ }
}
```

Every module in the modules map is a *factory function* that accepts two arguments: `module` and `require`. We will see in more detail what those arguments are in the next section. What is important to understand now is that every module here is a complete representation of the code in the original source module. If we take, for example, the code for the `calculator.js` module, it might be represented as follows:

```
(module, require) => {
  const { parser } = require('parser.js')
  const { resolver } = require('resolver.js')
  module.exports.calculator = function (expr) {
    return resolver(parser(expr))
  }
}
```



Note how the ESM syntax has been converted into something that resembles the syntax of the CommonJS module system. Remember that the browser does not support CommonJS and that these variables are not global, so there is no risk of a naming collision here. In this simplified implementation, we decided to use exactly the same identifiers as in CommonJS (`module`, `require`, and `module.exports`) to make the similarity with CommonJS look more apparent. In reality, every module bundler will use its own unique identifiers. For instance, webpack uses identifiers such as `__webpack_require__` and `__webpack_exports__`.

Packing

The modules map is the final output of the dependency resolution phase. In the packing phase, the module bundler takes the modules map and converts it into an *executable bundle*: a single JavaScript file that contains all the business logic of the original application.

The idea is simple: we already have a representation of the original codebase of our application inside the modules map; we have to find a way to convert it into something that the browser can execute correctly and save it into the resulting bundle file.

Given the structure of our modules map, this can actually be done with just a few lines of code wrapping the modules map:

```
((modulesMap) => { // (1)
  const require = (name) => { // (2)
    const module = { exports: {} } // (3)
    modulesMap[name](module, require) // (4)
    return module.exports // (5)
  }
  require('app.js') // (6)
})(
{
  'app.js': (module, require) => { /* ... */},
  'calculator.js': (module, require) => { /* ... */},
  'display.js': (module, require) => { /* ... */},
  'parser.js': (module, require) => { /* ... */},
  'resolver.js': (module, require) => { /* ... */},
}
)
```


This is not a lot of code, but there's a lot happening here, so let's go through it together, step by step:

1. In this code snippet, we have an **Immediately Invoked Function Expression (IIFE)** that receives the entire modules map as an argument.
2. When the function is executed, it defines a *custom* require function. This function receives a module name as input and it will load and execute the corresponding module from `modulesMap`.
3. In the require function, a module object is initialized. This object has only one property called `exports`, which is an object with no attributes.
4. At this point, the factory function of the given module is invoked and we pass to it the module object we just created and a reference to the require function itself. Note that this is essentially an implementation of the Service Locator pattern (nodejsdp.link/service-locator-pattern). Here, the factory function, once executed, modifies the module object by attaching to it the functionality that the module exports. The factory function can also recursively require other modules by using the require function passed as an argument.
5. Finally, the require function returns the `module.exports` object, which was populated by the factory function that was invoked in the previous step.
6. The last step is to require the entry point of our dependency graph, which in our case is the module `app.js`. This last step is what actually bootstraps the entire application. In fact, by loading the entry point, it will, in turn, load and execute all its dependencies in the right order and then execute its own business logic.

With this process, we essentially created a self-sufficient module system that is capable of loading modules that have been properly organized within the same file. In other words, we managed to convert an app originally organized in multiple files into an equivalent app where all the code has been moved into a single file. This is the resulting bundle file.



Note that the preceding code has been intentionally simplified just to illustrate how module bundlers work. There are many edge cases that we did not take into account. For instance, what happens if we require a module that does not exist in the modules map?

Using webpack

Now that we know how a module bundler works, let's build a simple application that can work both on Node.js and on the browser. Throughout this exercise, we will learn how to write a simple library that can be used without changes from the browser app and the server app. We will be using webpack to build the browser bundle.

To keep things simple, our application will be nothing more than a simple "hello world" for now, but don't worry, we will be building a more realistic application in the *Creating a Universal JavaScript app* section, later in this chapter.

Let's start by installing the webpack CLI in our system with:

```
npm install --global webpack-cli
```

Let's now initialize a new project in a new folder with:

```
npm init
```

Once the guided project initialization is complete, since we want to use ESM in Node.js, we need to add the property "type": "module" to our package.json.

Now, we can run:

```
webpack-cli init
```

This guided procedure will install webpack in your project and it will help you to automatically generate a webpack configuration file. At the time of writing, using webpack 4, the guided procedure does not realize that we want to use ESM in Node.js, so we have to apply two small changes to the generated files:

- Rename webpack.config.js to webpack.config.cjs
- Change the following npm scripts in package.json:

```
"build": "webpack --config webpack.config.cjs"
"start": "webpack-dev-server --config webpack.config.cjs"
```

Now, we are ready to start writing our application.

Let's first write the module we want to share in `src/say-hello.js`:

```
import nunjucks from 'nunjucks'

const template = '<h1>Hello <i>{{ name }}</i></h1>'

export function sayHello (name) {
  return nunjucks.renderString(template, { name })
}
```

In this code, we are using the `nunjucks` template library (nodejsdp.link/nunjucks), which must be installed with `npm`. This module is exporting a simple `sayHello` function that accepts a `name` as the only argument and uses it to construct an HTML string.

Let's now write the browser application that will use this module (`src/index.js`):

```
import { sayHello } from './say-hello.js'

const body = document.getElementsByTagName('body')[0]
body.innerHTML = sayHello('Browser')
```

This code uses the `sayHello` function to build an HTML fragment saying *Hello Browser* and then inserts it into the `body` section of the current HTML page.

If you want to preview this application, you can run `npm start` in your terminal. This should open your default browser and you should see the application running.

If you want to generate a static version of the application, you can run:

```
npm run build
```

This will generate a folder called `dist` containing two files: an `index.html` and our bundle file (whose name will look like `main.12345678901234567890.js`).



The file name of the bundle is generated by using a hash of the file content. This way, every time our source code changes, we will obtain a new bundle with a different name. This is a useful optimization technique, called **cache busting**, that `webpack` adopts by default and it is particularly convenient when deploying our assets to a **content delivery network (CDN)**. With CDNs, it is generally quite expensive to override files that are geographically distributed across multiple servers and already cached in multiple layers, possibly including our users' browsers. By generating new files with every change, we avoid cache invalidation entirely.

You can open the `index.html` file with your browser to see a preview of your application.

If you are curious, you can have a look at the generated bundle file. You will notice that it is a bit more convoluted and verbose than the sample bundle we illustrated in the previous section. However, you should be able to recognize the structure and notice that the entire `nunjucks` library, as well as our `sayHello` module, have been embedded in the bundle code.

Now, what if we want to build an equivalent application that runs in Node.js? For instance, we could use the `sayHello` function and display the resulting code in the terminal:

```
// src/server.js
import { sayHello } from './say-hello.js'
console.log(sayHello('Node.js'))
```

That's it!

If we run this code with:

```
node src/server.js
```

We will see the following output:

```
<h1>Hello <i>Node.js</i></h1>
```

Yes, displaying HTML in the terminal is not particularly useful, but right now we achieved our goal of being able to use a library from both the browser and the server without any changes in the library codebase.

In the next sections, we will discuss some patterns that allow us to actually change the code where necessary if we want to provide more specialized behaviors on the browser or Node.js.

Fundamentals of cross-platform development

When developing for different platforms, the most common problem we face is how can we reuse as much code as possible and, at the same time, provide specialized implementations for details that are platform-specific. We will now explore some of the principles and the patterns to use when facing this challenge, such as code branching and module swapping.

Runtime code branching

The most simple and intuitive technique for providing different implementations based on the host platform is to dynamically branch our code. This requires that we have a mechanism to recognize the host platform at runtime and then dynamically switch the implementation with an `if...else` statement. Some generic approaches involve checking global variables that are available only on Node.js or only on the browser.

For example, we can check the existence of the `window` global variable. Let's modify our `say-hello.js` module to use this technique to provide a slightly different functionality depending on whether the module is running on the browser or on the server:

```
import nunjucks from 'nunjucks'

const template = '<h1>Hello <i>{{ name }}</i></h1>'

export function sayHello (name) {
  if (typeof window !== 'undefined' && window.document) {
    // client-side code
    return nunjucks.renderString(template, { name })
  }

  // Node.js code
  return `Hello \u001b[1m${name}\u001b[0m`
}
```



The escape sequence `\u001b[1m` is a special terminal formatting indicator that sets the text to bold. The sequence `\u001b[0m` instead resets the formatting to normal. If you are curious to find out more about escape sequences and their history, check out ANSI escape sequences: nodejsdp.link/ansi-escape-sequences.

Try again to run our application on Node.js and on the browser and see the differences! If you do that, you will not see HTML code on the terminal when running the Node.js application. Instead, you will see a string with proper terminal formatting. The frontend application on the browser remains unchanged.

Challenges of runtime code branching

Using a runtime branching approach for switching between Node.js and the browser is definitely the most intuitive and simple pattern we can use for this purpose; however, there are some inconveniences:

- The code for both platforms is included in the same module and therefore in the final bundle. This increases the bundle size, adding unreachable and unnecessary code. It is also possible that the unreachable code contains sensitive information like encryption keys or API keys that are not meant to be sent to a user's browser. In this case, this approach might also raise significant security concerns.
- If used too extensively, it can considerably reduce the readability of the code, as the business logic would be mixed with logic meant only to add cross-platform compatibility.
- Using dynamic branching to load a different module, depending on the platform, will result in all the modules being added to the final bundle, regardless of their target platform. For example, if we consider the following code fragment, both `clientModule` and `serverModule` will be included in a bundle generated with webpack, unless we explicitly exclude one of them from the build:

```
import { clientFunctionality } from 'clientModule'
import { serverFunctionality } from 'serverModule'
if (typeof window !== 'undefined' && window.document) {
  clientFunctionality()
} else {
  serverFunctionality()
}
```

This last inconvenience happens because of the following reasons:

- Bundlers have no sure way of knowing the value of a runtime variable at build time (unless the variable is a constant), so, in the preceding example, both branches of the `if...else` statement are always included in the final bundle, even though it is obvious that the browser will always execute only one of them.
- ES module imports are always defined declaratively at the top of the file and we don't have a way to filter the imports based on the current environment. The bundler will not try to understand whether you are conditionally using only a subset of the imported feature and it will include all the imported code anyway.

A consequence of this last property is that modules imported dynamically using variables are not included in the bundle. For example, from the following code, no module will be bundled:

```
moduleList.forEach(function(module) {  
  import(module)  
})
```

It's worth underlining that webpack overcomes some of these limitations and, under certain specific circumstances, it is able to guess all the possible values for a dynamic requirement. For instance, if you have a snippet of code like the following:

```
function getControllerModule (controllerName) {  
  return import(`./controller/${controllerName}`)  
}
```

Webpack will include all the modules available in the controller folder in the final bundle.

It's highly recommended to have a look at the official documentation to understand all the supported cases (nodejsdp.link/webpack-dynamic-imports).

Build-time code branching

In this section, we are going to see how to use webpack plugins to remove, at build time, all parts of the code that we want to run only on the server. This allows us to obtain lighter bundle files and to avoid accidentally exposing code containing sensible information (for instance, secrets, passwords, or API keys) that should only live on the server.

Webpack offers support for plugins, which allows us to extend webpack's capabilities and add new processing steps that can be used to produce the bundle file. To perform build-time code branching, we can leverage a built-in plugin called DefinePlugin and a third-party plugin called terser-webpack-plugin (nodejsdp.link/terser-webpack).

DefinePlugin can be used to replace specific code occurrences in our source files with custom code or variables. terser-webpack-plugin allows us to compress the resulting code and remove unreachable statements (**dead code elimination**).

Let's start by rewriting our say-hello.js module to explore these concepts:

```
import nunjucks from 'nunjucks'  
export function sayHello (name) {  
  if (typeof __BROWSER__ !== 'undefined') {
```

```

    // client-side code
    const template = '<h1>Hello <i>{{ name }}</i></h1>'
    return nunjucks.renderString(template, { name })
  }
  // Node.js code
  return `Hello \u001b[1m${name}\u001b[0m`
}

```

Note that we are checking for the existence of a generic variable called `__BROWSER__` to enable the browser code. This is the variable that we will replace at build time using `DefinePlugin`.

Now, let's install `terser-webpack-plugin` with:

```
npm install --save-dev terser-webpack-plugin
```

Finally, let's update our `webpack.config.cjs` file:

```

// ...
const TerserPlugin = require('terser-webpack-plugin')
module.exports = {
  mode: 'production',
  // ...
  plugins: [
    // ...
    new webpack.DefinePlugin({
      __BROWSER__: true
    })
  ],
  // ...
  optimization: {
    // ...
    minimize: true,
    minimizer: [new TerserPlugin()]
  }
}

```

The first change here is to set the option `mode` to `production`. This option will enable optimizations such as code **minification** (or minimization). Optimization options are defined in the dedicated optimization object. Here, we are enabling minification by setting `minimize` to `true` and we are providing a new instance of `terser-webpack-plugin` as the minimizer. Finally, we are also adding `webpack.DefinePlugin` and configuring it to replace the string `__BROWSER__` with the value `true`.

Every value in the configuration object of `DefinePlugin` represents a piece of code that will be evaluated by webpack at build time and then used to replace the currently matched snippet of code. This allows us to add external dynamic values containing, for instance, the content of an environment variable, the current timestamp, or the hash of the last git commit to the bundle.

With this configuration, when we build a new bundle, every occurrence of `__BROWSER__` is replaced with `true`. The first `if` statement will internally look like `if (true !== 'undefined')`, but webpack is smart enough to understand that this expression will always be evaluated as `true`, so it transforms the resulting code again into `if (true)`.

Once webpack has finished processing all the code, it will invoke `terser-webpack-plugin` to minimize the resulting code. `terser-webpack-plugin` is a wrapper around `Terser` (nodejsdp.link/terser), a modern JavaScript minifier. Terser is capable of removing dead code as part of its minimization algorithm, so given that, at this stage, our code will look like this:

```
if (true) {  
  const template = '<h1>Hello <i>{{ name }}</i></h1>'  
  return nunjucks.renderString(template, { name })  
}  
return `Hello \u001b[1m${name}\u001b[0m`
```

Terser will reduce it to:

```
const template = '<h1>Hello <i>{{ name }}</i></h1>'  
return nunjucks.renderString(template, { name })
```

This way, we got rid of all the server-side code in our browser bundle.

Even if build-time code branching is way better than runtime code branching because it produces much leaner bundle files, it can still make our source code cumbersome when abused. In fact, if you overuse this technique, you will end up with code that contains too many `if` statements, which will be hard to understand and debug.

When this happens, it is generally better to move all the platform-specific code into dedicated modules. We will discuss this alternative approach in the next section.

Module swapping

Most of the time, we already know at build time what code has to be included in the client bundle and what shouldn't. This means that we can take this decision upfront and instruct the bundler to replace the implementation of an entire module at build time. This often results in a leaner bundle, as we are excluding unnecessary modules, and more readable code because we don't have all the `if...else` statements required by runtime and build-time branching.

Let's find out how to adopt module swapping with webpack by updating our example.

The main idea is that we want to have two separate implementations of our `sayHello` functionality: one optimized for the server (`say-hello.js`) and one optimized for the browser (`say-hello-browser.js`). We will then tell webpack to replace any import of `say-hello.js` with `say-hello-browser.js`. Let's see what our new implementation looks like now:

```
// src/say-hello.js
import chalk from 'chalk'
export function sayHello (name) {
  return `Hello ${chalk.green(name)}`
}

// src/say-hello-browser.js
import nunjucks from 'nunjucks'
const template = '<h1>Hello <i>{{ name }}</i></h1>'
export function sayHello (name) {
  return nunjucks.renderString(template, { name })
}
```

Note that, on the server-side version, we introduced a new dependency, `chalk` (nodejsdp.link/chalk), a utility library that allows us to format text for the terminal. This is to demonstrate one of the main advantages of this approach. Now that we've separated our server-side code from the client-side code, we can introduce new functionalities and libraries without worrying about the impact that those might have on the frontend-only bundle. At this point, in order to tell webpack to swap the modules at build time, we have to replace `webpack.DefinePlugin` with a new plugin in our `webpack.config.cjs`, as follows:

```
plugins: [
  // ...
  new webpack.NormalModuleReplacementPlugin(
    /src\/say-hello\.js$/,
    path.resolve(__dirname, 'src', 'say-hello-browser.js')
  )
]
```

We are using `webpack.NormalModuleReplacementPlugin`, which accepts two arguments. The first argument is a regular expression and the second one is a string representing a path to a resource. At build time, if a module path matches the given regular expression, it is replaced with the one provided in the second argument.

Note that this technique is not limited to our internal modules, but it can also be used with external libraries in our `node_modules` folder.

Thanks to webpack and the module replacement plugin, we can easily deal with structural differences between platforms. We can focus on writing separate modules that are meant to provide platform-specific code and we can then swap Node.js-only modules with browser-specific ones in the final bundle.

Design patterns for cross-platform development

Let's now revise some of the design patterns we discussed in the previous chapters to see how we can leverage those for cross-platform development:

- **Strategy and template:** These two are probably the most useful patterns when sharing code with the browser. Their intent is, in fact, to define the common steps of an algorithm, allowing some of its parts to be replaced, which is exactly what we need! In cross-platform development, these patterns allow us to share the platform-agnostic part of our components, while allowing their platform-specific parts to be changed using a different strategy or template method (which can be changed using code branching (runtime or build-time) or module swapping).
- **Adapter:** This pattern is probably the most useful when we need to swap an entire component. We have already seen several examples in *Chapter 8, Structural Design Patterns*. If your server application is using a database like SQLite, you could use the Adapter pattern to provide an alternative data storage implementation that works in the browser. For instance you could use the `localStorage` API (nodejsdp.link/localstorage) or the `IndexedDB` API (nodejsdp.link/indexdb).
- **Proxy:** When code meant to run on the server runs on the browser, we often need functionality that is used on the server to be available on the browser as well. This is where the *remote* Proxy pattern is useful. Imagine if we wanted to access the filesystem of the server from the browser: we could think of creating an `fs` object on the client that proxies every call to the `fs` module living on the server, using Ajax or WebSockets as a way of exchanging commands and return values.

- **Dependency injection and service locator:** Both dependency injection and service locator can be useful to replace the implementation of a module at the moment of its injection. When we introduced the concept of modules maps, in the *Packing* section, we also saw how the Service Locator pattern was intrinsically used by module bundlers to collate all the code from different modules into one file.

As we can see, the arsenal of patterns at our disposal is quite powerful, but the most powerful weapon is still the ability of the developer to choose the best approach and adapt it to the specific problem at hand.

Now that we understand the fundamentals of module bundlers and we have learned a number of useful patterns to write cross-platform code, we are ready to move into the second part of this chapter, where we will learn about React and write our first universal JavaScript application.

A brief introduction to React

React is a popular JavaScript library created and maintained by Facebook. React is focused on providing a comprehensive set of functions and tools to build the view layer in web applications. React offers a view abstraction focused on the concept of a **component**. A component could be a button, a form input, a simple container such as an HTML div, or any other element in your user interface. The idea is that you should be able to construct the user interface of your application by just defining and composing highly reusable components with specific responsibilities.

What makes React different from other view libraries for the web is that it is not bound to the DOM by design. In fact, it provides a high-level abstraction called the **virtual DOM** (nodejsdp.link/virtual-dom) that fits very well with the web but that can also be used in other contexts, for example, for building mobile apps, modeling 3D environments, or even defining the interaction between hardware components. In simple terms, the virtual DOM can be seen as an efficient way to re-render data organized in a tree-like structure.

"Learn it once, use it everywhere."

— Facebook

This is the motto used by Facebook to introduce React. It intentionally mocks the famous Java motto *Write it once, run it everywhere* with the clear intention to underline a fundamental shift from the Java philosophy. The original design goal of Java was to allow developers to write applications once and run them on as many platforms as possible without changes. Conversely, the React philosophy acknowledges that every platform is inherently different and therefore encourages developers to write different applications that are optimized for the related target platform. React, as a library, shifts its focus on providing *convenient* design and architecture principles and tools that, once mastered, can be easily used to write platform-specific code.



If you are curious to learn about the applications of React in contexts not strictly related to the field of web development, you can have a look at the following projects: **React Native** for mobile apps (nodejsdp.link/react-native), **React PIXI** for 2D rendering with OpenGL (nodejsdp.link/react-pixi), **react-three-fiber** to create 3D scenes (nodejsdp.link/react-three-fiber), and **React Hardware** (nodejsdp.link/react-hardware).

The main reason why React is so interesting in the context of Universal JavaScript development is because it allows us to render React components both on the client and on the server using almost the same code. To put it another way, with React, we are able to render the HTML code that is required to display the page directly from Node.js. Then, when the page is loaded on the browser, React will perform a process called **hydration** (nodejsdp.link/hydration), which will add all the frontend-only side effects like click handlers, animations, additional asynchronous data fetching, dynamic routing, and so on. Hydration converts a static markup into a fully interactive experience.

This approach allows us to build **single-page applications (SPAs)**, where the first render happens mostly on the server, but then, once the page is loaded on the browser and the user starts to click around, only the parts of the page that need to be changed are dynamically refreshed, without requiring a full page reload.

This design offers two main advantages:

- **Better search engine optimization (SEO):** Since the page markup is pre-rendered by the server, various search engines can make sense of the content of the page by just looking at the HTML returned by the server. They won't need to simulate a browser environment and wait for the page to be fully loaded to see what a given page is about.

- **Better performance:** Since we are pre-rendering the markup, this will be already available and visible on the browser, even while the browser is still downloading, parsing, and executing the JavaScript code included with the page. This approach can lead to a better user experience as the content is perceived to load faster and there are less browser "flashes" during rendering.



It is worth mentioning that the React virtual DOM is capable of optimizing the way changes are rendered. This means that the DOM is not rendered in full after every change, but instead React uses a smart in-memory diffing algorithm that is able to pre-calculate the minimum number of changes to apply to the DOM in order to update the view. This results in a very efficient mechanism for fast browser rendering.

Now that we know what React is, in the next section, we will write our first React component!

Hello React

Without further ado, let's start to use React and jump to a concrete example. This will be a "Hello World" type of example but it will help us to illustrate the main ideas behind React, before we move onto more realistic examples.

Let's start by creating a new webpack project in a new folder with:

```
npm init -y
npm install --save-dev webpack webpack-cli
node_modules/.bin/webpack init
```

Then, follow the guided instructions. Now, let's install React:

```
npm install --save react react-dom
```

Now, let's create a file, `src/index.js`, with the following content:

```
import react from 'react'
import ReactDOM from 'react-dom'

const h = react.createElement // (1)

class Hello extends react.Component { // (2)
  render () { // (3)
```

```
    return h('h1', null, [                                // (4)
      'Hello ',
      this.props.name || 'World'                          // (5)
    ])
  }
}

ReactDOM.render(                                         // (6)
  h(Hello, { name: 'React' }),
  document.getElementsByTagName('body')[0]
)
```

Let's review what's happening with this code:

1. The first thing that we do is to create a handy alias for the `react.createElement` function. We will be using this function a couple of times in this example to create React elements. These could be plain DOM nodes (regular HTML tags) or instances of React components.
2. Now, we define our `Hello` component, which has to extend the `react.Component` class.
3. Every React component has to implement a `render()` method. This method defines how the component will be displayed on the screen when it is rendered on the DOM and it has to return a React element.
4. We are using the `react.createElement` function to create an `h1` DOM element. This method expects three or more arguments. The first argument is the tag name (as a string) or a React component class. The second argument is an object used to pass attributes (or **props**) to the component (or `null` if we don't need to specify any attribute). Finally, the third argument is an array (or you can pass multiple arguments as well) of children elements. Elements can also be text (text nodes), as in our current example.
5. Here, we are using `this.props` to access the attributes that are passed to this component at runtime. In this specific case, we are looking for the `name` attribute. If this is passed, we use it to construct a text node; otherwise, we default to the string `"World"`.
6. In this last block of code, we use `ReactDOM.render()` to initialize our application. This function is responsible for attaching a React application to the existing page. An application is nothing more than an instance of a React component. Here, we are instantiating our `Hello` component and passing the string `"React"` for the `name` attribute. Finally, as the last argument, we have to specify which DOM node in the page will be the parent element of our application. In this case, we are using the `body` element of the page, but you can target any existing DOM element in the page.

Now, you can see a preview of your application by running:

```
npm start
```

You should now see "Hello React" in your browser window. Congratulations, you have built your first React application!

Alternatives to `react.createElement`

Repeated usage of `react.createElement()` might compromise the readability of our React components. In fact, nesting many invocations of `react.createElement()`, even with our `h()` alias, will make it hard to understand the HTML structure we want our components to render.

For this reason, it is not very common to use `react.createElement()` directly. To address this problem, the React team offers and encourages an alternative syntax called **JSX** (nodejsdp.link/jsx).

JSX is a superset of JavaScript that allows you to embed HTML-like code into JavaScript code. JSX makes the creation of React elements similar to writing HTML code. With JSX, React components are generally more readable and easier to write. It is easier to see what we mean here by looking at a concrete example, so let's rewrite our "Hello React" application using JSX:

```
import react from 'react'
import ReactDOM from 'react-dom'

class Hello extends react.Component {
  render () {
    return <h1>Hello {this.props.name || 'World'}</h1>
  }
}

ReactDOM.render(
  <Hello name="React"/>,
  document.getElementsByTagName('body')[0]
)
```

Much more readable, isn't it?

Unfortunately, since JSX is not a standard JavaScript feature, adopting JSX would require us to "compile" JSX code into standard equivalent JavaScript code. In the context of Universal JavaScript applications, we would have to do this both on the client-side code and the server-side code, so, for the sake of simplicity, we are not going to use JSX throughout the rest of this chapter.

There are some relatively new JSX alternatives that rely on standard JavaScript tagged template literals (you can read more about JavaScript tagged template literals at nodejsdp.link/template-literals). Using template literals seems to be a good compromise between code that is still quite easy to read and write and not having to perform an intermediate compilation process. Two of the most promising libraries providing this functionality are `htm` (nodejsdp.link/htm) and `esx` (nodejsdp.link/esx).

In the rest of this chapter, we will be using `htm`, so let's rewrite once more our "Hello React" example, this time using `htm`:

```
import react from 'react'
import ReactDOM from 'react-dom'
import htm from 'htm'

const html = htm.bind(react.createElement) // (1)
class Hello extends react.Component {
  render () { // (2)
    return html`<h1>
      Hello ${this.props.name || 'World'}
    </h1>`
  }
}

ReactDOM.render(
  html`<${Hello} name="React"/>`, // (3)
  document.getElementsByTagName('body')[0]
)
```

This code looks quite readable, but let's quickly clarify how we are using `htm` here:

1. The first thing that we have to do is create the template tag function `html`. This function allows us to use template literals to generate React elements. At runtime, this template tag function will be calling `react.createElement()` for us when needed.

2. Here, we use a tagged template literal with the `html` tag function to create an `h1` tag. Note that, as this is a standard tagged template literal, we can use the regular placeholder syntax (`${expression}`) to insert dynamic expressions into the string. Remember that template literals and tagged template literals use backticks (```) instead of single quotes (`'`) to delimit the template string.
3. Similarly, we can use the placeholder syntax to create instances of React components (`<${ComponentClass}>`). Note that, if a component instance contains children elements, we can use the special `</>` tag to indicate the end of the component (for example, `<${Component}><child/></>`). Finally, we can pass props to the component as normal HTML attributes.

At this point, we should be able to understand the basic structure of a simple "Hello World" React component. In the next section, we will show you how to manage states in a React component, an important concept for most real-world applications.

Stateful components

In the previous example, we saw how to build a *stateless* React component. By *stateless*, we mean that the component only receives input from the outside (in our example, it was receiving a `name` property) and it doesn't need to calculate or manage any internal information to be able to render itself to the DOM.

While it's great to have stateless components, sometimes, you have to manage some kind of state. React allows us to do that, so let's learn how with an example.

Let's build a React application that displays a list of projects that have been recently updated on GitHub.

We can encapsulate all the logic for asynchronously fetching the data from GitHub and displaying it on a dedicated component: the `RecentGithubProjects` component. This component is configurable through the `query` prop, which allows us to filter the projects on GitHub. The `query` prop will receive a keyword such as "javascript" or "react", and this value will be used to construct the API call to GitHub.

Let's finally have a look at the code of the `RecentGithubProjects` component:

```
// src/RecentGithubProjects.js
import react from 'react'
import htm from 'htm'

const html = htm.bind(react.createElement)

function createRequestUri (query) {
  return `https://api.github.com/search/repositories?q=${query}`
}
```

```
        encodeURIComponent(query)
      ]&sort=updated`
    }

export class RecentGithubProjects extends react.Component {
  constructor (props) { // (1)
    super(props) // (2)
    this.state = { // (3)
      loading: true,
      projects: []
    }
  }

  async loadData () { // (4)
    this.setState({ loading: true, projects: [] })
    const response = await fetch(
      createRequestUri(this.props.query),
      { mode: 'cors' }
    )
    const responseBody = await response.json()
    this.setState({
      projects: responseBody.items,
      loading: false
    })
  }

  componentDidMount () { // (5)
    this.loadData()
  }

  componentDidUpdate (prevProps) { // (6)
    if (this.props.query !== prevProps.query) {
      this.loadData()
    }
  }

  render () { // (7)
    if (this.state.loading) {
      return 'Loading ...'
    }

    // (8)
    return html`<ul>
```

```

    ${this.state.projects.map(project => html`
      <li key=${project.id}>
        <a href=${project.html_url}>${project.full_name}</a>:
        ${' '}${project.description}
      </li>
    `)}
  </ul>`
}
}

```

There are some new React concepts in this component, so let's discuss the main details here:

1. In this new component, we are overriding the default constructor. A constructor accepts the props passed to the component as an argument.
2. The first thing we have to do is call the original constructor and propagate the props so that the component can be initialized correctly by React.
3. Now, we can define the initial component state. Our final state is going to be a list of GitHub projects, but those won't be available immediately as we will need to load them dynamically. Therefore, we define the initial state as a boolean flag, indicating that we are loading the data and the list of projects as an empty array.
4. The function `loadData()` is the function that is responsible for making the API request, fetching the necessary data, and updating the internal state using `this.setState()`. Note that `this.setState()` is called twice: before we issue the HTTP request (to activate the loading state) and when the request is completed (to unset the loading flag and populate the list of projects). React will re-render the component automatically when the state changes.
5. Here, we are introducing another new concept: the `componentDidMount lifecycle` function. This function is automatically invoked by React once the component has been successfully instantiated and attached (or *mounted*) to the DOM. This is the perfect place to load our data for the first time.
6. The function `componentDidUpdate` is another React lifecycle function and it is automatically invoked every time the component is updated (for instance, if new props have been passed to the component). Here, we check if the query prop has changed since the last update. If that's the case, then we need to reload the list of projects.

7. Finally, let's see what happens in our `render()` function. The main thing to note is that here we have to handle the two different states of the component: the loading state and the state where we have the list of projects available for display. Since React will invoke the `render()` function every time the state or the props change, just having an `if` statement here will be enough. This technique is often called **conditional rendering**.
8. In this final step, we are rendering a list of elements using `Array.map()` to create a list element for every project fetched using the GitHub API. Note that every list element receives a value for the `key` prop. The `key` prop is a special prop that is recommended whenever you are rendering an array of elements. Every element should provide a unique key. This prop helps the virtual DOM optimize every rendering pass (If you are curious to understand in detail what React does in this situation you can have a look at nodejsdp.link/react-reconciliation).



You might have noticed that we are not handling potential errors while fetching the data. There are several ways we can do this in React. The most elegant solution is probably implementing an `ErrorBoundary` component (nodejsdp.link/error-boundary), but we will leave that as an exercise for you.

Let's now write the main application component. Here, we want to display a navigation menu where the user can select different queries ("JavaScript", "Node.js", and "React") to filter for different types of GitHub projects:

```
// src/App.js
import react from 'react'
import htm from 'htm'
import { RecentGithubProjects } from './RecentGithubProjects.js'

const html = htm.bind(react.createElement)

export class App extends react.Component {
  constructor (props) {
    super(props)
    this.state = {
      query: 'javascript',
      label: 'JavaScript'
    }
    this.setQuery = this.setQuery.bind(this)
  }
}
```

```

setQuery (e) {
  e.preventDefault()
  const label = e.currentTarget.text
  this.setState({ label, query: label.toLowerCase() })
}

render () {
  return html`<div>
    <nav>
      <a href="#" onClick=${this.setQuery}>JavaScript</a>
      ${' '}
      <a href="#" onClick=${this.setQuery}>Node.js</a>
      ${' '}
      <a href="#" onClick=${this.setQuery}>React</a>
    </nav>
    <h1>Recently updated ${this.state.label} projects</h1>
    <${RecentGithubProjects} query=${this.state.query}/>
  </div>`
}
}

```

This component is using its internal state to track the currently selected query. Initially, the "javascript" query is set and passed down to the `RecentGithubProjects` component. Then, every time a keyword in the navigation menu is clicked, we update the state with the new selected keyword. When this happens, the `render()` method will be automatically invoked and it will pass the new value for the query prop to `RecentGithubProjects`. In turn, `RecentGithubProjects` will be marked as *updated*, and it will internally reload and eventually update the list of projects for the new query.

One interesting detail to underline is that, in the constructor, we are explicitly binding the `setQuery()` function to the current component instance. The reason why we do this is because this function is used directly as an event handler for the click event. In this case, the reference to `this` would be undefined without the bind and it would not be possible to call `this.setState()` from the handler.

At this point, we only need to attach the `App` component to the DOM to run our application. Let's do this:

```

// src/index.js
import react from 'react'
import ReactDOM from 'react-dom'
import htm from 'htm'

```

```
import { App } from './App.js'

const html = htm.bind(react.createElement)

ReactDOM.render(
  html`<${App}/>`,
  document.getElementsByTagName('body')[0]
)
```

Finally, let's just run the application with `npm start` and test it on the browser.



Note that since we used `async/await` in our application, the default configuration generated by webpack might not work straight away. If you have any issues, compare your configuration file with the one in the code examples provided with this book (nodejsdp.link/wpconf).

Try to refresh the page and click on the various keywords on the navigation menu. After a few seconds, you should see the list of projects being refreshed.

At this point, it should be quite clear to you how React works, how to compose components together, and how to take advantage of state and props. Hopefully, this simple exercise will also help you to find new, interesting, open source JavaScript projects that you might want to contribute to!



We've covered just enough ground for us to be able to build our first Universal React application. But if you want to be proficient with React, we recommend that you read the official React documentation (nodejsdp.link/react-docs) for a more exhaustive overview of the library.

We are finally ready to take what we learned about webpack and React to create a simple, yet complete, universal JavaScript application.

Creating a Universal JavaScript app

Now that we've covered the basics, let's start to build a more complete Universal JavaScript application. We are going to build a simple "book library" application where we can list different authors and see their biography and some of their masterpieces. Although this is going to be a very simple application, it will allow us to cover more advanced topics such as **universal routing**, **universal rendering**, and **universal data fetching**. The idea is that you can later use this application as a scaffold for a real project and build on top of it your next universal JavaScript application.

In this experiment, we are going to use the following technologies:

- **React** (nodejsdp.link/react), which we just introduced
- **React Router** (nodejsdp.link/react-router), a companion routing layer for React
- **Fastify** (nodejsdp.link/fastify), a fast and ergonomic framework to build web servers in Node.js
- **Webpack** as the module bundler

For practical reasons, we selected a very specific set of technologies for this exercise, but we will try to focus as much as possible on the design principles and patterns rather than the technologies themselves. As you learn these patterns, you should be able to use the acquired knowledge with any other combination of technologies and achieve similar results.



In order to keep things simple, we will be using webpack only to process the frontend code and we will leave the backend code unchanged, leveraging the native Node.js support for ESM.

At the time of writing, there are some subtle discrepancies between how webpack interprets the semantics of ESM imports as opposed to how Node.js does it, especially when importing modules written using the CommonJS syntax. For this reason, we recommend running the examples in the rest of this chapter using `esm` (nodejsdp.link/esm), a Node.js library that will preprocess ESM imports in a way that minimizes those differences. Once you have installed the `esm` module in your project, you can run a script with `esm` as follows:

```
node -r esm script.js
```


Frontend-only app

In this section, we are going to focus on building our app on the frontend only, using webpack as a development web server. In the next sections, we will expand and update this basic app to convert it to a full Universal JavaScript application.

This time, we will be using a custom webpack configuration, so let's start by creating a new folder and copying the `package.json` and `webpack.config.cjs` files from the code repository provided with this book (nodejsdp.link/frontend-only-app), then install all the necessary dependencies with:

```
npm install
```

The data we will be using is stored in a JavaScript file (as a simple substitute for a database), so make sure you copy the file `data/authors.js` into your project as well. This file contains some sample data in the following format:

```
export const authors = [
  {
    id: 'author\'s unique id',
    name: 'author\'s name',
    bio: 'author\'s biography',
    books: [ // author's books
      {
        id: 'book unique id',
        title: 'book title',
        year: 1914 // book publishing year
      },
      // ... more books
    ]
  },
  // ... more authors
]
```

Of course, feel free to change the data in this file if you want to add your favorite authors and books!

Now that we have all the configuration in place, let's quickly discuss what we want our application to look like.

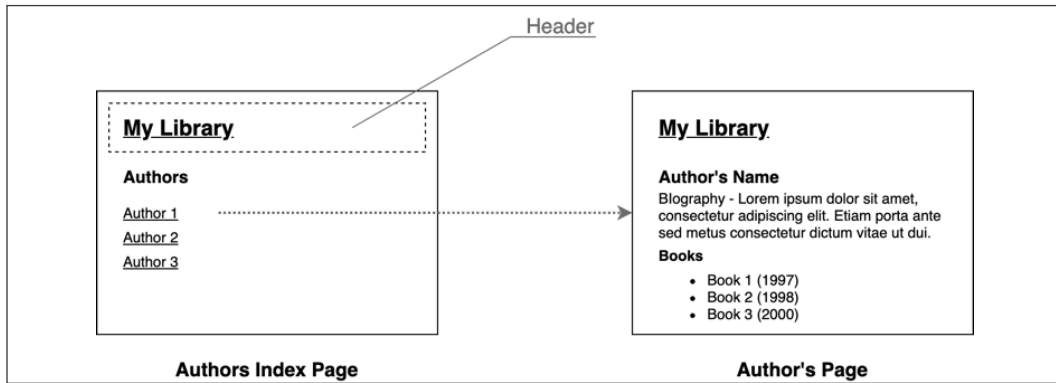


Figure 10.3: Application mockup

Figure 10.3 shows that our application will have two different types of page: an index page, where we list all the authors available in our data store, and then a page to visualize the details of a given author, where we will see their biography and some of their books.

These two types of page will only have a header in common. This will allow us to go back to the index page at any time.

We will be exposing the index page at the root path of our server (/), while we will be using the path /author/:authorId for the author's page.

Finally, we will also have a 404 page.

In terms of file structure, we will organize our project as follows:

```

src
├── data
│   └── authors.js          - data file
├── frontend
│   ├── App.js             - application component
│   ├── components
│   │   ├── Header.js      - header component
│   │   └── pages
│   │       ├── Author.js   - author page
│   │       ├── AuthorsIndex.js - index page
│   │       └── FourOhFour.js - 404 page
│   └── index.js           - project entry point

```

Let's start by writing the `index.js` module, which will serve as the entry point for loading our frontend application and attaching it to the DOM:

```
import react from 'react'
import ReactDOM from 'react-dom'
import htm from 'htm'
import { BrowserRouter } from 'react-router-dom'
import { App } from './App.js'

const html = htm.bind(react.createElement)

ReactDOM.render(
  html`<${BrowserRouter}><${App}/></>`,
  document.getElementById('root')
)
```

This code is quite simple as we are mainly importing the `App` component and attaching it to the DOM in an element with the ID equal to `root`. The only detail that stands out is that we are wrapping the application into a `BrowserRouter` component. This component comes from the `react-router-dom` library and it provides our app with client-side routing capabilities. Some of the components we will be writing next will showcase how to fully take advantage of these routing capabilities and how to connect different pages together using links. Later on, we will revisit this routing configuration to make it available on the server side as well.

Right now, let's focus on the source code for `App.js`:

```
import react from 'react'
import htm from 'htm'
import { Switch, Route } from 'react-router-dom'
import { AuthorsIndex } from './components/pages/AuthorsIndex.js'
import { Author } from './components/pages/Author.js'
import { FourOhFour } from './components/pages/FourOhFour.js'

const html = htm.bind(react.createElement)

export class App extends react.Component {
  render () {
    return html`
      <${Switch}>
        <${Route}
          path="/"
          exact=${true}
        >
```

```

        component=${AuthorsIndex}
      />
      <${Route}
        path="/author/:authorId"
        component=${Author}
      />
      <${Route}
        path="*"
        component=${FourOhFour}
      />
    </>
  }
}

```

As you can tell from this code, the App component is responsible for loading all the page components and configuring the routing for them.

Here, we are using the Switch component from react-router-dom. This component allows us to define Route components. Every Route component needs to have a path and a component prop associated with it. At render time, Switch will check the current URL against the paths defined by the routes, and it will render the component associated to the first Route component that matches.

As in a JavaScript switch statement, where the order of case statements is important, here, the order of the Route components is important too. Our last route is a *catch-all route*, which will always match if none of the previous routes matches..

Note also that we are setting the prop exact for the first Route. This is needed because react-router-dom will match based on prefixes, so a plain / will match any URL. By specifying exact: true, we are telling the router to only match this path if it is exactly / (and not if it just starts with /).

Let's now have a quick look at our Header component:

```

import react from 'react'
import htm from 'htm'
import { Link } from 'react-router-dom'

const html = htm.bind(react.createElement)

export class Header extends react.Component {
  render () {
    return html`<header>

```

```
    <h1>
      <${Link} to="/">My library</>
    </h1>
  </header>`
}
```

This is a very simple component that just renders an h1 title containing "My library." The only detail worth discussing here is that the title is wrapped by a `Link` component from the `react-router-dom` library. This component is responsible for rendering a clickable link that can interact with the application router to switch to a new route dynamically, without refreshing the entire page.

Now, we have to write, one by one, our page components. Let's start with the `AuthorsIndex` component:

```
import react from 'react'
import htm from 'htm'
import { Link } from 'react-router-dom'
import { Header } from '../Header.js'
import { authors } from '../../data/authors.js'

const html = htm.bind(react.createElement)

export class AuthorsIndex extends react.Component {
  render () {
    return html`<div>
      <${Header}/>
      <div>${authors.map((author) =>
        html`<div key=${author.id}>
          <p>
            <${Link} to="${`/author/${author.id}`}">
              ${author.name}
            </>
          </p>
        </div>`)}
      </div>
    </div>`
  }
}
```

Yet another very simple component. Here, we are rendering some markup dynamically based on the list of authors available in our data file. Note that we are using, once again, the Link component from react-router-dom to create dynamic links to the author page.

Now, let's have a look at the Author component code:

```
import react from 'react'
import htm from 'htm'
import { FourOhFour } from './FourOhFour.js'
import { Header } from '../Header.js'
import { authors } from '../../data/authors.js'

const html = htm.bind(react.createElement)

export class Author extends react.Component {
  render () {
    const author = authors.find(
      author => author.id === this.props.match.params.authorId
    )

    if (!author) {
      return html`<${FourOhFour} error="Author not found"/>`
    }
    return html`<div>
      <${Header}/>
      <h2>${author.name}</h2>
      <p>${author.bio}</p>
      <h3>Books</h3>
      <ul>
        ${author.books.map((book) =>
          html`<li key=${book.id}>${book.title} (${book.year})</li>`
        )}
      </ul>
    </div>`
  }
}
```

This component has a little bit of logic in it. In the render() method, we filter the authors dataset to find the current author. Notice that we are using props.match.params.authorId to get the current author ID. The match prop will be passed to the component by the router at render time and the nested params object will be populated if the current path has dynamic parameters.



It is common practice to memoize (nodejsdp.link/memoization) the result of any complex calculation performed in the `render()` method. This prevents the complex calculation from running again in case its inputs haven't changed since the last render. In our example, a possible target for this type of optimization is the call to `authors.find()`. We leave this to you as an exercise. If you want to know more about this technique take a look at nodejsdp.link/react-memoization.

There's a chance that we are receiving an ID that doesn't match any author in our dataset, so in this case, `author` will be undefined. This is clearly a 404, so instead of rendering the author data, we delegate the render logic to the `FourOhFour` component, which is responsible for rendering the 404 error page.

Finally, let's see the source code for the `FourOhFour` component:

```
import react from 'react'
import htm from 'htm'
import { Link } from 'react-router-dom'
import { Header } from '../Header.js'

const html = htm.bind(react.createElement)
export class FourOhFour extends react.Component {
  render () {
    return html`<div>
      <${Header}/>
      <div>
        <h2>404</h2>
        <h3>${this.props.error || 'Page not found'}</h3>
        <${Link} to="/">Go back to the home page</>
      </div>
    </div>`
  }
}
```

This component is responsible for rendering the 404 page. Note that we made the error message configurable through the `error` prop and also that we are using a `Link` from the `react-router-dom` library to allow the user to travel back to the home page when landing on this error page.

This was quite a lot of code, but we are finally ready to run our frontend-only React application: just type `npm start` in your console and you should see the application running in your browser. Pretty barebones, but if we did everything correctly, it should work as expected and allow us to see our favorite authors and their masterpieces.

It is worth using the app with the browser developer tools open so that we can verify that our dynamic routing is working correctly, that is, once the first page is loaded, transitions to other pages happen without any page refresh.



For a better understanding of what happens when you interact with a React application, you can install and use the React Developer Tools browser extension on Chrome (nodejsdp.link/react-dev-tools-chrome) or Firefox (nodejsdp.link/react-dev-tools-firefox).

Server-side rendering

Our application works and this is great news. However, the app is running only on the client side, which means that if we try to `curl` one of the pages, we will see something like this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My library</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/javascript" src="/main.js"></script></body>
</html>
```

No content whatsoever! There's only an empty container (the root `div`), which is where our application is mounted at runtime.

In this section, we will modify our application to be able to render the content also from the server.

Let's start by adding fastify and esm to our project:

```
npm install --save fastify fastify-static esm
```

Now, we can create our server application in `src/server.js`:

```
import { resolve, dirname } from 'path'
import { fileURLToPath } from 'url'
import react from 'react'
import reactServer from 'react-dom/server.js'
import htm from 'htm'
import fastify from 'fastify'
import fastifyStatic from 'fastify-static'
import { StaticRouter } from 'react-router-dom'
import { App } from './frontend/App.js'

const __dirname = dirname(fileURLToPath(import.meta.url))
const html = htm.bind(react.createElement)

// (1)
const template = ({ content }) => `<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My library</title>
  </head>
  <body>
    <div id="root">${content}</div>
    <script type="text/javascript" src="/public/main.js"></script>
  </body>
</html>`

const server = fastify({ logger: true }) // (2)

server.register(fastifyStatic, { // (3)
  root: resolve(__dirname, '..', 'public'),
  prefix: '/public/'
})

server.get('*', async (req, reply) => { // (4)
  const location = req.raw.originalUrl // (5)
```

```

const serverApp = html`
  <${StaticRouter} location=${location}>
    <${App}/>
  </>
`

const content = reactServer.renderToString(serverApp) // (6)
const responseHtml = template({ content })

reply.code(200).type('text/html').send(responseHtml)
})

const port = Number.parseInt(process.env.PORT) || 3000 // (7)
const address = process.env.ADDRESS || '127.0.0.1'

server.listen(port, address, function (err) {
  if (err) {
    console.error(err)
    process.exit(1)
  }
})

```

There's a lot of code here, so let's discuss step by step the main concepts introduced here:

1. Since we are not going to use the webpack dev server, we need to return the full HTML code of the page from our server. Here, we are defining the HTML template for all our pages using a function and a template literal. We will be passing the result of our server-rendered React application as content to this template to get the final HTML to return to the client.
2. Here, we create a Fastify server instance and enable logging.
3. As you might have noticed from our template code, our web application will load the script `/public/main.js`. This file is the frontend bundle that is generated by webpack. Here, we are letting the Fastify server instance serve all static assets from the public folder using the `fastify-static` plugin.
4. In this line, we define a catch-all route for every GET request to the server. The reason why we are doing a catch-all route is because the actual routing logic is already contained in the React application. When we render the React application, it will display the correct page component based on the current URL.

5. On the server side, we have to use an instance of `StaticRouter` from `react-router-dom` and wrap our application component with it. `StaticRouter` is a version of `React Router` that can be used for server-side rendering. This router, rather than taking the current URL from the browser window, allows us to pass the current URL directly from the server through the `location` prop.
6. Here, we can finally generate the HTML code for our `serverApp` component by using `React's renderToString()` function. The generated HTML is the same as the one generated by the client-side application on a given URL. In the next few lines, we wrap this code with our page layout using the `template()` function and finally, we send the result to the client.
7. In the last few lines of code, we tell our `Fastify` server instance to listen on a given address and port defaulting to `localhost:3000`.

Now, we can run `npm run build` to create the frontend bundle and finally, we can run our server, as follows:

```
node -r esm src/server.js
```

Let's open our browser on `http://localhost:3000/` and see if our app is still working as expected. All good, right? Great! Now, let's try to `curl` our home page to see if the server-generated code looks different:

```
curl http://localhost:3000/
```

This time, this is what we should see:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My library</title>
  </head>
  <body>
    <div id="root"><div><header><h1><a href="/">My library</a></h1></header><div><h2>Authors</h2><div><div><a href="/author/joyce"><p>James Joyce</p></a></div><div><a href="/author/h-g-wells"><p>Herbert George Wells</p></a></div><div><a href="/author/orwell"><p>George Orwell</p></a></div></div></div></div>
    <script type="text/javascript" src="/public/main.js"></script>
  </body>
</html>
```

Great! This time, our root container is not empty: we are rendering the list of authors directly from the server. You should also try some author pages and see that it works correctly for those as well. Mission complete! Well, almost... what happens if we try to render a page that does not exist? Let's have a look:

```
curl -i http://localhost:3000/blah
```

This will print:

```
HTTP/1.1 200 OK
content-type: text/html
content-length: 367
Date: Sun, 05 Apr 2020 18:38:47 GMT
Connection: keep-alive

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My library</title>
  </head>
  <body>
    <div id="root"><div><header><h1><a href="/">My library</a></h1></header><div><h2>404</h2><h3>Page not found</h3><a href="/">Go back to the home page</a></div></div></div>
    <script type="text/javascript" src="/public/main.js"></script>
  </body>
</html>
```

At first glance, this might seem correct because we are rendering our 404 page, but we are actually returning a 200 status code... not good!

We can actually fix this with just a little extra effort, so let's do it.

React `StaticRouter` allows us to pass a generic context prop that can be used to exchange information between the React application and the server. We can leverage this utility to allow our 404 page to inject some information into this shared context so that on the server side, we are aware of whether we should return a 200 or a 404 status code.

Let's update the catch-all route on the server side first:

```
server.get('*', async (req, reply) => {
  const location = req.raw.originalUrl
  const staticContext = {}
  const serverApp = html`
    <${StaticRouter}
      location=${location}
      context=${staticContext}
    >
    <${App}/>
  </>
  ,

  const content = reactServer.renderToString(serverApp)
  const responseHtml = template({ content })

  let code = 200
  if (staticContext.statusCode) {
    code = staticContext.statusCode
  }

  reply.code(code).type('text/html').send(responseHtml)
})
```

The changes from the previous version are highlighted in bold. As you can see, we create an empty object called `staticContext` and pass it to the router instance in the `context` prop. Later on, after the server-side rendering is completed, we check if `staticContext.statusCode` was populated during the rendering process. If it was, it will now contain the status code that we have to return to the client, together with the rendered HTML code.

Let's now change the `FourOhFour` component to actually populate this value. To do this, we just need to update the `render()` function with the following code before we return the elements to render:

```
if (this.props.staticContext) {
  this.props.staticContext.statusCode = 404
}
```

Note that the `context` prop passed to `StaticRouter` is passed only to direct children of `Route` components using the prop `staticContext`. For this reason, if we rebuild the frontend bundle and relaunch our server, this time, we will see a correct `404` status for `http://localhost:3000/blah`, but it won't work for URLs that match the author page such as `http://localhost:3000/author/blah`.

In order to make this work, we also need to propagate `staticContext` from the `Author` component into the `FourOhFour` component. To do this, in the `Author` component's `render()` method, we have to apply the following change:

```
if (!author) {  
  return html`<${FourOhFour}  
    staticContext=${this.props.staticContext}  
    error="Author not found"  
  />`  
}  
// ...
```

Now, the `404` status code will be returned correctly from the server, even on author pages for non-existent authors.

Great—we now have a fully functional React application that uses server-side rendering! But don't celebrate just yet, we still have some work to do...

Asynchronous data retrieval

Now, imagine for a second that we are asked to build the website for the Library of Trinity College in Dublin, one of the most famous libraries in the world. It has about 300 years of history and about 7 million books. Ok, now let's imagine we have to allow the users to browse this massive collection of books. Yes, all 7 million of them... a simple data file is not going to be a great idea here!

A better approach would be to have a dedicated API to retrieve the data about the books and use it to dynamically fetch only the minimum amount of data needed to render a given page. More data will be fetched as the user navigates through the various pages of the website.

This approach is valid for most web applications, so let's try to apply the same principle to our demo application. We will be using an API with two endpoints:

- `/api/authors`, to get the list of authors
- `/api/author/:authorId`, to get the information for a given author

For the sake of this demo application, we will keep things very simple. We only want to demonstrate how our application is going to change as soon as we introduce asynchronous data fetching, so we are not going to bother with using a real database to back our API or with introducing more advanced features like pagination, filtering, or search.

Since building such an API server leveraging our existing data file is a rather trivial exercise (one that doesn't add much value in the context of this chapter), we are going to skip the walkthrough of the API implementation. You can get the source code of the API server from the code repository of this book (nodejsdp.link/authors-api-server).



This simple API server runs independently from our backend server, so it uses another port (or potentially even on another domain). In order to allow the browser to make asynchronous HTTP requests to a different port or domain, we need our API server to support **cross-origin resource sharing** or **CORS** (nodejsdp.link/cors), a mechanism that allows secure cross-origin requests. Thankfully, enabling CORS with Fastify is as easy as installing the `fastify-cors` (nodejsdp.link/fastify-cors) plugin.

We are also going to need an HTTP client that works seamlessly on both the browser and Node.js. A good option is `superagent` (nodejsdp.link/superagent).

Let's install the new dependencies then:

```
npm install --save fastify-cors superagent
```

Now we are ready to run our API server:

```
node -r esm src/api.js
```

And let's try some requests with `curl`, for instance:

```
curl -i http://localhost:3001/api/authors
curl -i http://localhost:3001/api/author/joyce
curl -i http://localhost:3001/api/author/invalid
```

If everything worked as expected, we are now ready to update our React components to use these new API endpoints rather than reading directly from the authors dataset. Let's start by updating the `AuthorsIndex` component:

```
import react from 'react'
import htm from 'htm'
import { Link } from 'react-router-dom'
import superagent from 'superagent'
import { Header } from '../Header.js'

const html = htm.bind(react.createElement)
```

```

export class AuthorsIndex extends react.Component {
  constructor (props) {
    super(props)
    this.state = {
      authors: [],
      loading: true
    }
  }

  async componentDidMount () {
    const { body } = await superagent.get('http://localhost:3001/api/authors')
    this.setState({ loading: false, authors: body })
  }

  render () {
    if (this.state.loading) {
      return html`<${Header}/><div>Loading ...</div>`
    }

    return html`<div>
      <${Header}/>
      <div>${this.state.authors.map((author) =>
        html`<div key=${author.id}>
          <p>
            <${Link} to="${`/author/${author.id}`}">
              ${author.name}
            </>
          </p>
        </div>`)}
      </div>
    </div>`
  }
}

```

The main changes from the previous version are highlighted in bold. Essentially, we converted our React component into a stateful component. At construction time, we initialized the state to an empty array of authors and we set the loading flag to true. Then, we used the `componentDidMount` lifecycle method to load the authors data using the new API endpoint. Finally, we updated the `render()` method to display a loading message while the data was being loaded asynchronously.

Now, we have to update our Author component:

```
import react from 'react'
import htm from 'htm'
import superagent from 'superagent'
import { FourOhFour } from './FourOhFour.js'
import { Header } from '../Header.js'

const html = htm.bind(react.createElement)

export class Author extends react.Component {
  constructor (props) {
    super(props)
    this.state = {
      author: null,
      loading: true
    }
  }

  async loadData () {
    let author = null
    this.setState({ loading: false, author })
    try {
      const { body } = await superagent.get(
        `http://localhost:3001/api/author/${
          this.props.match.params.authorId
        }`)
      author = body
    } catch (e) {}
    this.setState({ loading: false, author })
  }

  componentDidMount () {
    this.loadData()
  }

  componentDidUpdate (prevProps) {
    if (prevProps.match.params.authorId !==
      this.props.match.params.authorId) {
      this.loadData()
    }
  }
}
```

```

render () {
  if (this.state.loading) {
    return html`<${Header}/><div>Loading ...</div>`
  }

  if (!this.state.author) {
    return html`<${FourOhFour}
      staticContext=${this.props.staticContext}
      error="Author not found"
    />`
  }

  return html`<div>
    <${Header}/>
    <h2>${this.state.author.name}</h2>
    <p>${this.state.author.bio}</p>
    <h3>Books</h3>
    <ul>
      ${this.state.author.books.map((book) =>
        html`<li key=${book.id}>
          ${book.title} (${book.year})
        </li>`
      )}
    </ul>
  </div>`
}
}

```

The changes here are quite similar to the ones we applied to the previous component. In this component, we also generalized the data loading operation into the `loadData()` method. We did this because this component implements not just the `componentDidMount()` but also the `componentDidUpdate()` lifecycle method. This is necessary because if we end up passing new props to the same component instance, we want the component to update correctly. This will happen, for instance, if we have a link in the author page that points to another author page, something that could happen if we implement a "related authors" feature in our application.

At this point, we are ready to try this new version of the code. Let's regenerate the frontend bundle with `npm run build` and start both our backend server and our API server, then point our browser to `http://localhost:3000/`.

If you navigate around the various pages, everything should work as expected. You might also notice that page content gets loaded interactively as you navigate through the pages.

But what happens to our server-side rendering? If we try to use `curl` on our home page, we should see the following HTML markup being returned:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My library</title>
  </head>
  <body>
    <div id="root"><div><header><h1><a href="/">My library</a></h1></div><div>Loading ...</div></div></div>
    <script type="text/javascript" src="/public/main.js"></script>
  </body>
</html>
```

Did you notice that there's no content anymore, but just a quite useless "Loading ..." indicator? This is not good. Also, this is not the only problem here. If you try to use `curl` on an invalid author page, you will notice that you will get the same HTML markup with the loading indicator and no content and that the returned status code is 200 rather than 404!



We don't see any real content on the server-side rendered markup because the `componentDidMount` lifecycle method is executed only on the browser while it's ignored by React during server-side rendering.

Furthermore, server-side rendering is a synchronous operation, so even if we move our loading code somewhere else, we still won't be able to perform any asynchronous data loading while rendering on the server.

In the next section of this chapter, we will explore a pattern that can help us to achieve full universal rendering and data loading.

Universal data retrieval

Server-side rendering is a synchronous operation and this makes it tricky to preload all the necessary data effectively. Being able to avoid the problems we underlined at the end of the previous section is not as straightforward as you might expect.

The root of the problem is that we are keeping our routing logic within the React application, so, on the server, we cannot know which page we are actually going to render before we call `renderToString()`. This is why the server cannot establish whether we need to preload some data for a particular page.

Universal data retrieval is still quite a nebulous area in React, and different frameworks or libraries that facilitate React server-side rendering have come up with different solutions to this problem.

As of today, the two patterns that we believe are worth discussing are **two-pass rendering** and **async pages**. These two techniques have different ways of figuring out which data needs to be preloaded. In both cases, once the data is fully loaded on the server, the generated HTML page will provide an inline script block to inject all the data into the global scope (the window object) so that when the application runs on the browser, the same data already loaded on the server won't have to be reloaded from the client.

Two-pass rendering

The idea of two-pass rendering is to use the React router static context as a vector to exchange information between React and the server. *Figure 10.4* shows us how this works:

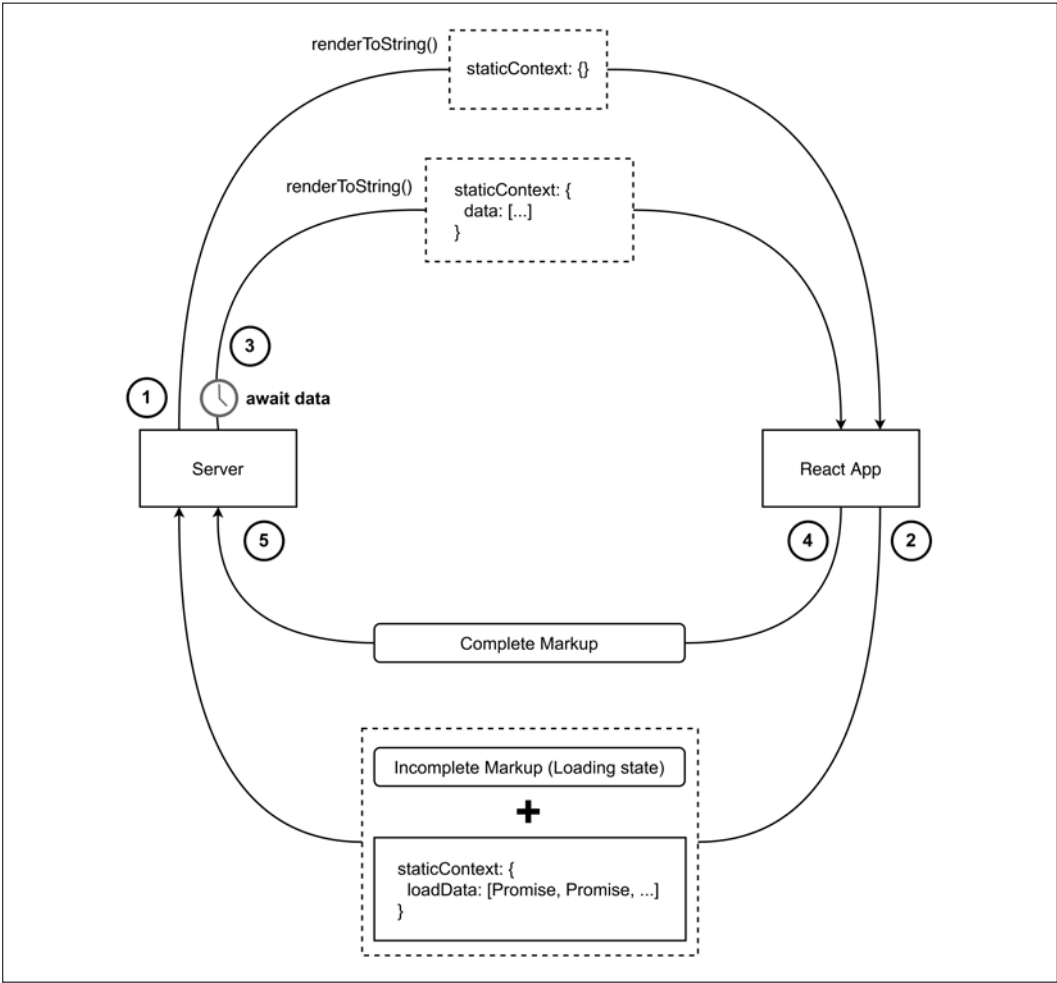


Figure 10.4: Two-pass rendering schematic

The steps of two-pass rendering are as follows:

1. The server calls `renderToString()`, passing the URL received from the client and an empty static context object to the React application.

2. The React application will perform the routing process and select the components that need to be rendered for the given URL. Every component that requires to load some data asynchronously will need to implement some extra logic to allow such data to be preloaded on the server as well. This can be done by attaching a promise representing the result of the data loading operation to the router static context. This way, at the end of the rendering process, the server will receive an incomplete markup (representing the current loading state) and the static context will contain a number of promises representing data loading operations.
3. At this point, the server can look at the static context and wait for all the promises to settle to make sure that all the data has been preloaded completely. During this process, the server builds a new static context that contains the results returned by the promises. This new static context is used for a second round of rendering. This is why this technique is called two-pass rendering.
4. Now, the ball is again on the React side of the field. The routing process should pick the same components used during the first rendering pass, since the URL has not changed. This time, the components that need data preloading should see that such data is already available in the static context and they can render the view straight away. This step produces a complete static markup that the server can now use.
5. At this point, the server has the complete markup and it uses it to render the final HTML page. The server can also include all the preloaded data in a script tag so that, on the browser, the data will be already available so there won't be any need to load it again while visiting the first page of the application.

This technique is very powerful and has some interesting advantages. For instance, it allows you to organize your React components tree in a very flexible way. You can have multiple components requesting asynchronous data, and they can be placed at any level of the components tree.

In more advanced use cases, you can also have data being loaded over multiple rendering passes. For instance, during the second pass, a new component in the tree might be rendered and this component might also need to load data asynchronously so it can just add new promises to the static context. To support this particular case, the server will have to continue the rendering loop until there are no more promises left in the static context. This particular variation of the two-pass rendering technique is referred to as **multi-pass rendering**.

The biggest disadvantage of this technique is that every call to `renderToString()` is not cheap and in real-life applications, this technique might force the server to go through multiple rendering passes, making the whole process very slow.

This might lead to severe performance degradation on the entire application, which can dramatically affect the user experience.

A simpler but potentially more performant alternative will be discussed in the next section.

Async pages

The technique we are going to describe here, which we are going to call "async pages," is based on a more constrained structure of the React application.

The idea is to structure the top layers of the application components tree in a very specific way. Let's have a look at a possible structure first, then it will be easier to discuss how this specific approach can help us with asynchronous data loading.

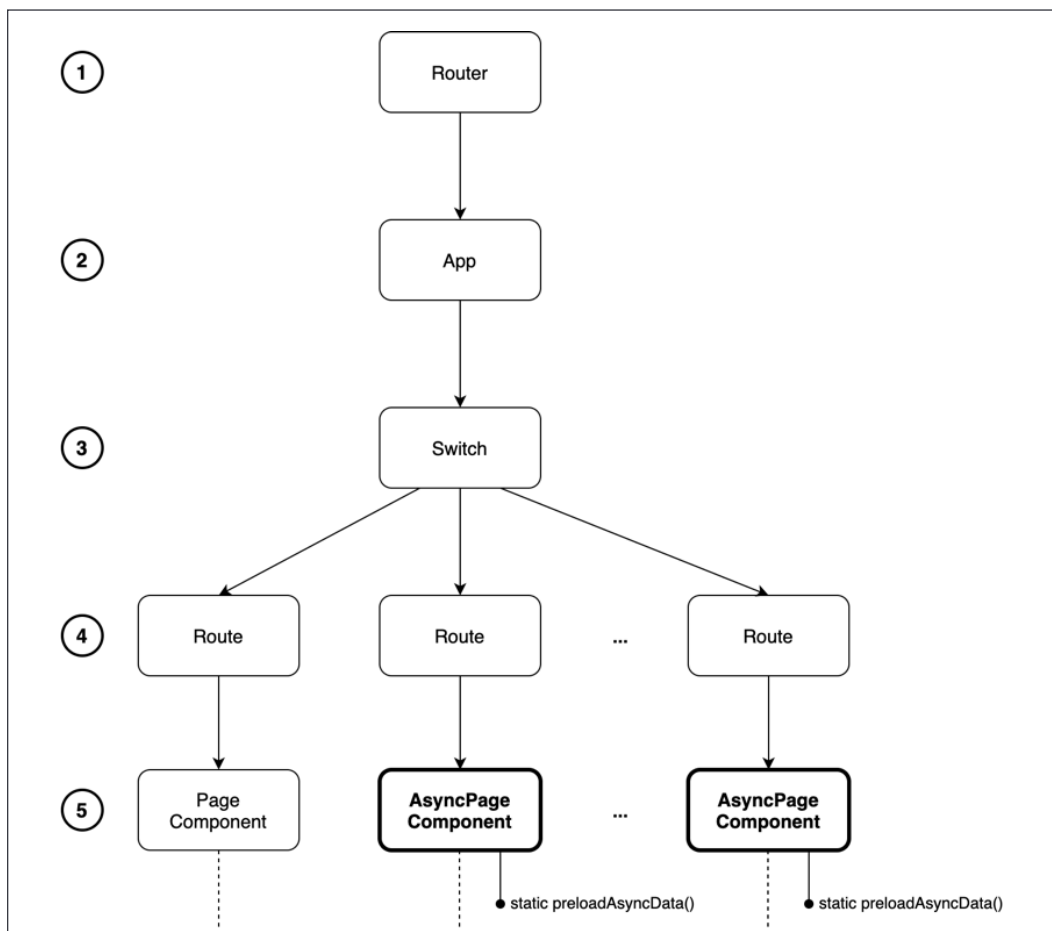


Figure 10.5: Async pages components tree structure

In Figure 10.5, we have represented the structure that allows us to apply the async pages technique. Let's discuss in detail the scope of every layer in the components tree:

1. The root of the application is always a Router component (`StaticRouter` on the server and `BrowserRouter` on the client).
2. The application component is the only child of the Router component.
3. The only child of the application component is a `Switch` component from the `react-router-dom` package.
4. The `Switch` component has one or more `Route` components as children. These are used to define all the possible routes and which component should be rendered for every route.
5. This is the most interesting layer as we actually introduce the concept of a "page component." The idea is that a page component is responsible for the look and feel of an entire page. A page component can have an arbitrary subtree of components used to render the current view; for instance, a header, a body, and a footer. We can have two types of page components: regular page components that behave as any other React components and `AsyncPage` components. Async pages are special *stateful* components that need to preload data for the page to be rendered both on the server- and the client side. They implement a special static method called `preloadAsyncData()` that contains the logic necessary to preload the data for the given page.

You can see that layers 1 to 4 are responsible for the routing logic, while level 5 is responsible for data loading and for actually rendering the current page. There are no other nested layers for additional routing and data loading.



Technically, there could be additional layers for routing and data loading after level 5, but those won't be universally available as they will be resolved only on the client side after the page has been rendered.

Now that we've discussed this more rigid structure, let's see how it can be useful to avoid multiple rendering passes and achieve universal data retrieval.

Here's the idea: if we have our routes defined in a dedicated file as an array of paths and components, we can easily reuse this file on the server side and determine, before the React rendering phase, which page component we will actually end up rendering.

Then, we can see if this page component is an `AsyncPage`. If it is, it means we have to preload some data on the server side before the rendering. We can do this by calling the `preloadAsyncData()` method from the given component.

Once the data has been preloaded, this can be added in the static context and we can render the entire application. During the rendering phase, the `AsyncPage` component will see that its data is already preloaded and available in the static context and it will be able to render straight away, skipping the loading state.

Once the rendering is finished, the server can add the same preloaded data in a script tag so that, on the browser side, the user won't have to wait for the data to be loaded again.



The Next.js framework (nodejsdp.link/nextjs) is a popular framework for Universal JavaScript applications and adopts a similar technique to the one described here, so it is a good example of this pattern in the wild.

Implementing async pages

Now that we know how to solve our data fetching problems, let's implement the async pages technique in our application.

Our components tree is already structured in a way that it's compliant to what's expected by this technique. Our pages are the `AuthorsIndex` component, the `Author` component, and the `FourOhFour` component. The first two require universal data loading, so we will have to convert them into async pages.

Let's start to update our application by extrapolating the route definitions into a dedicated file, `src/frontend/routes.js`:

```
import { AuthorsIndex } from './components/pages/AuthorsIndex.js'
import { Author } from './components/pages/Author.js'
import { FourOhFour } from './components/pages/FourOhFour.js'

export const routes = [
  {
    path: '/',
    exact: true,
    component: AuthorsIndex
  },
  {
    path: '/author/:authorId',
```

```

    component: Author
  },
  {
    path: '*',
    component: FourOhFour
  }
]

```

We want this configuration file to be the source of truth for the router configuration across the various parts of the application, so let's refactor the frontend App component to use this file as well:

```

// src/frontend/App.js
import React from 'react'
import htm from 'htm'
import { Switch, Route } from 'react-router-dom'
import { routes } from './routes.js'

const html = htm.bind(React.createElement)

export class App extends React.Component {
  render () {
    return html`<${Switch}>
      ${routes.map(routeConfig =>
        html`<${Route}
          key=${routeConfig.path}
          ...${routeConfig}
        />`
      )}
    </>`
  }
}

```

As you can see, the only change here is that, rather than defining the various Route components inline, we build them dynamically starting from the routes configuration array. Any change in the routes.js file will be automatically reflected in the application as well.

At this point, we can update the server-side logic in src/server.js.

The first thing that we want to do is import a utility function from the react-router-dom package that allows us to see if a given URL matches a given React router path definition. We also need to import the routes array from the new routes.js module.

```
// ...
import { StaticRouter, matchPath } from 'react-router-dom'
import { routes } from './frontend/routes.js'
// ...
```

Now, let's update our server-side HTML template generation function to be able to embed preloaded data in our page:

```
// ...
const template = ({ content, serverData }) => `<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>My library</title>
  </head>
  <body>
    <div id="root">${content}</div>
    ${serverData ? `<script type="text/javascript">
window.__STATIC_CONTEXT__=${JSON.stringify(serverData)}
</script>` : ''}
    <script type="text/javascript" src="/public/main.js"></script>
  </body>
</html>`
// ...
```

As you can see, our template now accepts a new argument called `serverData`. If this argument is passed to the template function, it will render a script tag that will inject this data into a global variable called `window.__STATIC_CONTEXT__`.

Now, let's get into the meaty bit; let's rewrite the server-side rendering logic:

```
// ...
server.get('*', async (req, reply) => {
  const location = req.raw.originalUrl
  let component // (1)
  let match
  for (const route of routes) {
    component = route.component
    match = matchPath(location, route)
    if (match) {
      break
    }
  }
}
```

```

let staticData // (2)
let staticError
let hasStaticContext = false
if (typeof component.preloadAsyncData === 'function') {
  hasStaticContext = true
  try {
    const data = await component.preloadAsyncData({ match })
    staticData = data
  } catch (err) {
    staticError = err
  }
}
const staticContext = {
  [location]: {
    data: staticData,
    err: staticError
  }
}

// (3)

const serverApp = html`
  <${StaticRouter}
    location=${location}
    context=${staticContext}
  >
  <${App}/>
</>
`

const content = reactServer.renderToString(serverApp)
const serverData = hasStaticContext ? staticContext : null
const responseHtml = template({ content, serverData })

const code = staticContext.statusCode
  ? staticContext.statusCode
  : 200
reply.code(code).type('text/html').send(responseHtml)
// ...

```

There are quite some changes here. Let's discuss the main blocks one by one:

1. The first change aims to detect which page will be rendered for the current URL. We loop through the defined routes and we use the `matchPath` utility to verify if location matches the current route definition. If it does, we stop the loop and record which component will be rendered in the component variable. We can be sure a component will be matched here because our last route (the 404 page) will always match. The `match` variable will contain information about the match. For instance, if the route contains some parameters, `match` will contain the path fragment that matched every parameter. For instance, for the URL `/author/joyce`, `match` will have the property `params` equal to `{ authorId: 'joyce' }`. This is the same prop that a page component will receive from the router when rendered.
2. In the second block of changes, we check if the selected component is an `AsyncPage`. We do that by checking if the component has a static method called `preloadAsyncData`. If that's the case, we invoke that function by passing an object that contains the `match` object as an argument (this way, we propagate any parameter that might be needed to fetch the data, such as `authorId`). This function should return a promise. If the promise resolves, we have successfully preloaded the data for this component. If it rejects, we make sure to record the error. Finally, we create the `staticContext` object. This object maps the preloaded data (or the rejection error) to the current location. The reason why we keep the location as a key is to be sure that if, for any reason, the browser renders another page from the one we preloaded (because of a programmatic error or because of a user action, like hitting the back button on the browser before the page is fully loaded), we won't end up using preloaded data that is not relevant to the current page on the browser.
3. In the last block of changes, we invoke the `renderToString()` function to get the rendered HTML of the application. Note that since we are passing a static context containing the preloaded data, we expect that the application will be able to completely render the page without returning a loading state view. This does not happen magically, of course. We will need to add some logic to our React component to check if the necessary data is already available in the static context. Once we have the generated HTML, we use our `template()` function to generate the complete page markup and we return it to the browser. We also make sure to respect the status code. For instance, if we ended up rendering the `FourOhFour` component, we will have the `statusCode` property in the static context changed, so if that's the case, we use that value for the final status code; otherwise, we default to `200`.

That's it for our server-side rendering.

Now, it's time to create the async page abstraction in our React application. Since we are going to have two distinct async pages, a good way to reuse some code is to create a base class and to use the Template pattern that we already discussed in *Chapter 9, Behavioral Design Patterns*. Let's define this class in `src/frontend/components/pages/AsyncPage.js`:

```
import react from 'react'

export class AsyncPage extends react.Component {
  static async preloadAsyncData (props) { // (1)
    throw new Error('Must be implemented by sub class')
  }

  render () {
    throw new Error('Must be implemented by sub class')
  }

  constructor (props) { // (2)
    super(props)
    const location = props.match.url
    this.hasData = false

    let staticData
    let staticError

    const staticContext = typeof window !== 'undefined'
      ? window.__STATIC_CONTEXT__ // client-side
      : this.props.staticContext // server-side

    if (staticContext && staticContext[location]) {
      const { data, err } = staticContext[location]
      staticData = data
      staticError = err
      this.hasStaticData = true

      typeof window !== 'undefined' &&
        delete staticContext[location]
    }

    this.state = {
      ...staticData,
      staticError,
      loading: !this.hasStaticData
    }
  }
}
```

```
    }  
  }  
  
  async componentDidMount () {                                // (3)  
    if (!this.hasStaticData) {  
      let staticData  
      let staticError  
      try {  
        const data = await this.constructor.preloadAsyncData(  
          this.props  
        )  
        staticData = data  
      } catch (err) {  
        staticError = err  
      }  
      this.setState({  
        ...staticData,  
        loading: false,  
        staticError  
      })  
    }  
  }  
}
```

This class provides helper code for building a stateful component that can handle three possible scenarios:

- We are rendering on the server and we already have the data preloaded (no need to load the data).
- We are rendering on the client and the data is already available in the page through the `__STATIC_CONTEXT__` variable (no need to load the data).
- We are rendering on the client and the data is not available (for instance, if this page was not rendered by the server, but a page the user navigated to after the first load). In this case, the data has to be dynamically loaded from the client when the component is mounted.

Let's review the main points of this implementation together:

1. This component class should not be instantiated directly but only extended when implementing async pages. When this class is extended, the async page component will need to implement the methods `static async preloadAsyncData(props)` and `render()`.

2. In the constructor, we have to initialize the component state. There are two possible outcomes here: the data is already available (so we can set it in the state) or the data is not available (so we need to set the state to "loading" and let the component load the data once it's mounted on the page). If we are on the browser and we load the data from the static context, we also make sure to delete this data from the context. This will allow the user to see fresh data if they happen to go back to this page during the navigation.
3. The method `componentDidMount()` is executed by React only on the browser. Here, we handle the case where the data was not preloaded and we have to dynamically load it at runtime.

Now that we have this useful abstraction in place, we can rewrite our `AuthorsIndex` and `Author` components and convert them into async pages. Let's start with `AuthorsIndex`:

```
import react from 'react'
import htm from 'htm'
import { Link } from 'react-router-dom'
import superagent from 'superagent'
import { AsyncPage } from './AsyncPage.js'
import { Header } from '../Header.js'

const html = htm.bind(react.createElement)

export class AuthorsIndex extends AsyncPage {
  static async preloadAsyncData (props) {
    const { body } = await superagent.get(
      'http://localhost:3001/api/authors'
    )
    return { authors: body }
  }

  render () {
    // unchanged...
  }
}
```

As you can see here, our `AuthorsIndex` component now extends `AsyncPage`. Since the `AsyncPage` template will take care of all the state management in its constructor, we don't need a constructor here anymore; we just need to specify the business logic to load the data in the `preloadAsyncData()` method.

If you compare this implementation with the previous one, you might notice that the logic of this method is almost the same as what we had previously in `componentDidMount()`. The method `componentDidMount()` has been removed from here because the one we inherit from `AsyncPage` will suffice. The only difference between the previous version of `componentDidMount()` and `preloadAsyncData()` is that in `preloadAsyncData()`, we don't set the internal state directly; we just need to return the data. The underlying code in `AsyncPage` will update the state as needed for us.

Let's now rewrite the `Author` component:

```
import react from 'react'
import htm from 'htm'
import superagent from 'superagent'
import { AsyncPage } from './AsyncPage.js'
import { FourOhFour } from './FourOhFour.js'
import { Header } from '../Header.js'

const html = htm.bind(react.createElement)

export class Author extends AsyncPage {
  static async preloadAsyncData (props) {
    const { body } = await superagent.get(
      `http://localhost:3001/api/author/${
        props.match.params.authorId
      }`
    )
    return { author: body }
  }

  render () {
    // unchanged...
  }
}
```

The changes here are perfectly in line with the changes we made for the `AuthorsIndex` component. We are only moving the data loading logic into `preloadAsyncData()` and letting the underlying abstraction manage the state transition for us.

Now, we can apply just a last small optimization in our `src/frontend/index.js` file. We can swap the `ReactDOM.render()` function call with `ReactDOM.hydrate()`. Since we will produce exactly the same markup from both the server side and the client side, this will make React a bit faster to initialize during the first browser load.

We are finally ready to try all these changes. Make sure to rebuild the frontend bundle and relaunch the server. Have a look at the application and the code that is generated by the server; it should contain all the preloaded data for every page. Also, 404 errors should be reported correctly for every 404 page, including the ones for missing authors.

Great! We finally managed to build an application that efficiently shares code, logic, and data between the client and the server: a true Universal JavaScript application!

Summary

In this chapter, we explored the innovative and fast-moving world of Universal JavaScript. Universal JavaScript opens up a lot of new opportunities in the field of web development and it can help you build single-page applications that load fast, are accessible, and are optimized for search engines.

In this chapter, we focused on introducing all the basics of this subject. We started from exploring module bundlers, why we need them, and how they work. We learned how to use webpack, and then we introduced React and discussed some of its functionality. We learned how to build component-oriented user interfaces and then started to build an application from scratch to explore universal rendering, universal routing, and universal data retrieval.

Even though we discussed a lot of topics, we barely scratched the surface of this wide topic, but you should have gained all the necessary knowledge to keep exploring this world on your own if you are interested in knowing more. Given that this field is still evolving quite rapidly, tools and libraries will probably change a lot in the next few years, but all the basic concepts should stay there, so don't be afraid to keep exploring and experimenting. Becoming an expert on this topic is now just a matter of using the acquired knowledge to build a first real-world app with real, business-driven use cases.

It's also worth underlining that the knowledge acquired here might be useful for projects that cross the boundaries of web development, like mobile app development. If you are interested in this topic, React Native might be a good starting point.

In the next chapter, we are going to take a problem-solution approach to explore some more advanced topics. Are you ready?

Exercises

- **10.1 A matter of style:** Our little library application looks very barebone. It could look a lot better with some style and images. Why don't you try to improve the look of the app? If you get stuck or need some inspiration, you can check our version of this exercise on GitHub (nodejsdp.link/univ).
- **10.2 Proper data management:** As we said, keeping a lot of data in a file is not a great idea. Why don't you try to move all the data into a real database backend of your choice? At this point, you might also want to take this application to the next level and write some script to import data from a big collection of books like the Open Library archive (nodejsdp.link/open-library-api).
- **10.3 Pagination and search:** Now that you have a more significant database, it's probably time to add some important features like search and pagination.
- **10.4 A Universal... Blog!:** Build a new universal JavaScript app from scratch that implements a blog. Then, try the same exercise using a framework like Next.js (nodejsdp.link/nextjs) or Gatsby (nodejsdp.link/gatsby).

11

Advanced Recipes

In this chapter, we'll take a problem-solution approach and, like in a cookbook, we'll present a set of ready-to-use *recipes* to solve some common Node.js programming problems.

You shouldn't be surprised by the fact that most of the problems presented in this chapter arise when we try to do things asynchronously. In fact, as we've seen repeatedly in the previous chapters of this book, tasks that are trivial in traditional synchronous programming can become more complicated when applied to asynchronous programming. A typical example is trying to use a component that requires an asynchronous initialization step. In this case, we have the inconvenience of delaying any attempt to use the component until the initialization completes. We'll show you how to solve this elegantly later.

But this chapter is not just about recipes involving asynchronous programming. You will also learn the best ways to run CPU-intensive tasks in Node.js.

These 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

One of the reasons for the existence of synchronous APIs in the Node.js core modules and many npm packages is because they are handy to use for implementing initialization tasks. For simple programs, using synchronous APIs at initialization time can streamline things a lot and the drawbacks associated with their use remain contained because they are used only once, which is when the program or a particular component is initialized.

Unfortunately, this is not always possible. A synchronous API might not always be available, especially for components using the network during their initialization phase to, for example, perform handshake protocols or to retrieve configuration parameters. This is the case for many database drivers and clients for middleware systems such as message queues.

The issue with asynchronously initialized components

Let's consider an example where a module called `db` is used to interact with a remote database. The `db` module will accept API requests only after the connection and handshake with the database server have been successfully completed. Therefore, no queries or other commands can be sent until the initialization phase is complete. The following is the code for such a sample module (the `db.js` file):

```
import { EventEmitter } from 'events'

class DB extends EventEmitter {
  connected = false

  connect () {
    // simulate the delay of the connection
    setTimeout(() => {
      this.connected = true
      this.emit('connected')
    }, 500)
  }

  async query (queryString) {
    if (!this.connected) {
      throw new Error('Not connected yet')
    }
  }
}
```

```

    }
    console.log(`Query executed: ${queryString}`)
  }
}

export const db = new DB()

```

This is a typical example of an asynchronously initialized component. Under these assumptions, we usually have two quick and easy solutions to this problem, which we can call *local initialization check* and *delayed startup*. Let's analyze them in more detail.

Local initialization check

The first solution makes sure that the module is initialized before any of its APIs are invoked; otherwise, we wait for its initialization. This check has to be done every time we want to invoke an operation on the asynchronous module:

```

import { once } from 'events'
import { db } from './db.js'

db.connect()

async function updateLastAccess () {
  if (!db.connected) {
    await once(db, 'connected')
  }

  await db.query(`INSERT (${Date.now()}) INTO "LastAccesses"`)
}

updateLastAccess()
setTimeout(() => {
  updateLastAccess()
}, 600)

```

As we already anticipated, any time we want to invoke the `query()` method on the `db` component, we have to check if the module is initialized; otherwise, we wait for its initialization by listening for the `'connected'` event. A variation of this technique performs the check inside the `query()` method itself, which shifts the burden of the boilerplate code from the consumer to the provider of the service.

Delayed startup

The second quick and dirty solution to the problem of asynchronously initialized components involves delaying the execution of any code relying on the asynchronously initialized component until the component has finished its initialization routine. We can see an example of such a technique in the following code fragment:

```
import { db } from './db.js'
import { once } from 'events'

async function initialize () {
  db.connect()
  await once(db, 'connected')
}

async function updateLastAccess () {
  await db.query(`INSERT (${Date.now()}) INTO "LastAccesses"`)
}

initialize()
  .then(() => {
    updateLastAccess()
    setTimeout(() => {
      updateLastAccess()
    }, 600)
  })
```

As we can see from the preceding code, we first wait for the initialization to complete, and then we proceed with executing any routine that uses the db object.

The main disadvantage of this technique is that it requires us to know, in advance, which components will make use of the asynchronously initialized component, which makes our code fragile and exposed to mistakes. One solution to this problem is delaying the startup of the entire application until all the asynchronous services are initialized. This has the advantage of being simple and effective; however, it can add a significant delay to the overall startup time of the application and moreover, it won't take into account the case in which the asynchronously initialized component has to be reinitialized.

As we will see in the next section, there is a third alternative that allows us to transparently and efficiently delay every operation until the asynchronous initialization step has completed.

Pre-initialization queues

Another recipe to make sure that the services of a component are invoked only after the component is initialized involves the use of queues and the Command pattern. The idea is to queue the method invocations (only those requiring the component to be initialized) received while the component is not yet initialized, and then execute them as soon as all the initialization steps have been completed.

Let's see how this technique can be applied to our sample db component:

```
import { EventEmitter } from 'events'

class DB extends EventEmitter {
  connected = false
  commandsQueue = []

  async query (queryString) {
    if (!this.connected) {
      console.log(`Request queued: ${queryString}`)

      return new Promise((resolve, reject) => {           // (1)
        const command = () => {
          this.query(queryString)
            .then(resolve, reject)
        }
        this.commandsQueue.push(command)
      })
    }

    console.log(`Query executed: ${queryString}`)
  }

  connect () {
    // simulate the delay of the connection
    setTimeout(() => {
      this.connected = true
      this.emit('connected')
      this.commandsQueue.forEach(command => command())    // (2)
      this.commandsQueue = []
    }, 500)
  }
}

export const db = new DB()
```


As we already mentioned, the technique described here consists of two parts:

1. If the component has not been initialized – which, in our case, is when the `connected` property is `false` – we create a command from the parameters received with the current invocation and push it to the `commandsQueue` array. When the command is executed, it will run the original `query()` method again and forward the result to the `Promise` we are returning to the caller.
2. When the initialization of the component is completed – which, in our case, means that the connection with the database server is established – we go through the `commandsQueue`, executing all the commands that have been previously queued.

With the `DB` class we just implemented, there is no need to check if the component is initialized before invoking its methods. In fact, all the logic is embedded in the component itself and any consumer can just transparently use it without worrying about its initialization status.

We can also go a step further and try to reduce the boilerplate of the `DB` class we just created and, at the same time, improve its modularity. We can achieve that by applying the State pattern, which we learned about in *Chapter 9, Behavioral Design Patterns*, with two states:

- The first state implements all the methods that require the component to be initialized, and it's activated only when there is a successful initialization. Each of these methods implements its own business logic without worrying about the initialization status of the `db` component
- The second state is activated before the initialization has completed and it implements the same methods as the first state, but their only role here is to add a new command to the queue using the parameters passed to the invocation.

Let's see how we can apply the structure we just described to our `db` component. First, we create the `InitializedState`, which implements the actual business logic of our component:

```
class InitializedState {  
  async query (queryString) {  
    console.log(`Query executed: ${queryString}`)  
  }  
}
```

As we can see, the only method that we need to implement in the `InitializedState` class is the `query()` method, which will print a message to the console when it receives a new query.

Next, we implement the `QueuingState`, the core of our recipe. This state implements the queuing logic:

```
const METHODS_REQUIRING_CONNECTION = ['query']
const deactivate = Symbol('deactivate')

class QueuingState {
  constructor (db) {
    this.db = db
    this.commandsQueue = []

    METHODS_REQUIRING_CONNECTION.forEach(methodName => {
      this[methodName] = function (...args) {
        console.log('Command queued:', methodName, args)
        return new Promise((resolve, reject) => {
          const command = () => {
            db[methodName](...args)
              .then(resolve, reject)
          }
          this.commandsQueue.push(command)
        })
      }
    })
  }

  [deactivate] () {
    this.commandsQueue.forEach(command => command())
    this.commandsQueue = []
  }
}
```

It's interesting to note how the `QueuingState` is mostly built dynamically at creation time. For each method that requires an active connection, we create a new method for the current instance, which queues a new command representing the function invocation. When the command is executed at a later time, when a connection is established, the result of the invocation of the method on the `db` instance is forwarded to the caller (through the returned promise).

The other important part of this state class is `[deactivate]()`. This method is invoked when the state is deactivated (which is when the component is initialized) and it executes all the commands in the queue. Note how we used a `Symbol` to name the method.

This will avoid any name clashes in the future if we add more methods to the state (for example, what if we need to decorate a hypothetical `deactivate()` method of the DB class?).

Now, it's time to reimplement the DB class using the two states we just described:

```
class DB extends EventEmitter {
  constructor () {
    super()
    this.state = new QueuingState(this)           // (1)
  }

  async query (queryString) {
    return this.state.query(queryString)          // (2)
  }

  connect () {
    // simulate the delay of the connection
    setTimeout(() => {
      this.connected = true
      this.emit('connected')
      const oldState = this.state                 // (3)
      this.state = new InitializedState(this)
      oldState[deactivate] && oldState[deactivate]()
    }, 500)
  }
}

export const db = new DB()
```

Let's further analyze the most important parts of the new DB class:

1. In the constructor, we initialize the current state of the instance. It's going to be the `QueuingState` as the asynchronous initialization of the component hasn't been completed yet.
2. The only method of our class implementing some (stub) business logic is the `query()` method. Here, all we have to do is invoke the homonymous method on the currently active state.
3. Finally, when we establish the connection with the database (initialization complete), we switch the current state to the `InitializedState` and we deactivate the old one. The effect of deactivating the `QueuedState`, as we've seen previously, is that any command that had been queued is now executed.

We can immediately see how this approach allows us to reduce the boilerplate and, at the same time, create a class that is purely business logic (the `InitializedState`) free from any repetitive initialization check.

The approach we've just seen will only work if we can modify the code of our asynchronously initialized component. In all those cases in which we can't make modifications to the component, we will need to create a wrapper or proxy, but the technique will be mostly similar to what we've seen here.

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 in order 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)**.

Take a 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. Take a look at the relevant code line at nodejsdp.link/pg-queue.

Asynchronous request batching and caching

In high-load applications, **caching** plays a critical role and it's used almost everywhere on the web, from static resources such as web pages, images, and stylesheets, to pure data such as the result of database queries. In this section, we are going to learn how caching applies to asynchronous operations and how a high request throughput can be turned to our advantage.

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. Take a look at the following diagram:

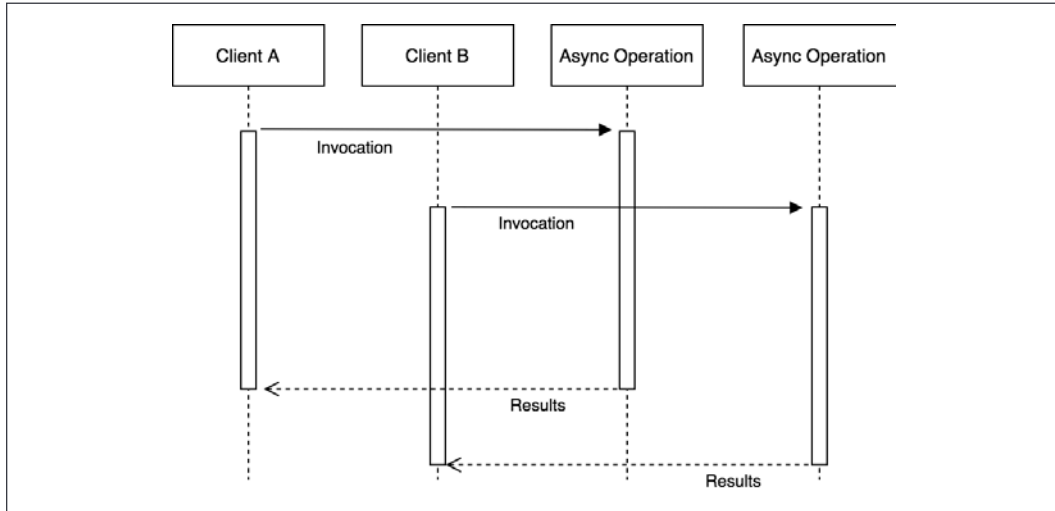


Figure 11.1: Two asynchronous requests with no batching

The previous diagram shows two clients invoking the same asynchronous operation with *exactly the same input*. Of course, the natural way to picture this situation is with the two clients starting two separate operations that will complete at two different moments.

Now, consider the following scenario:

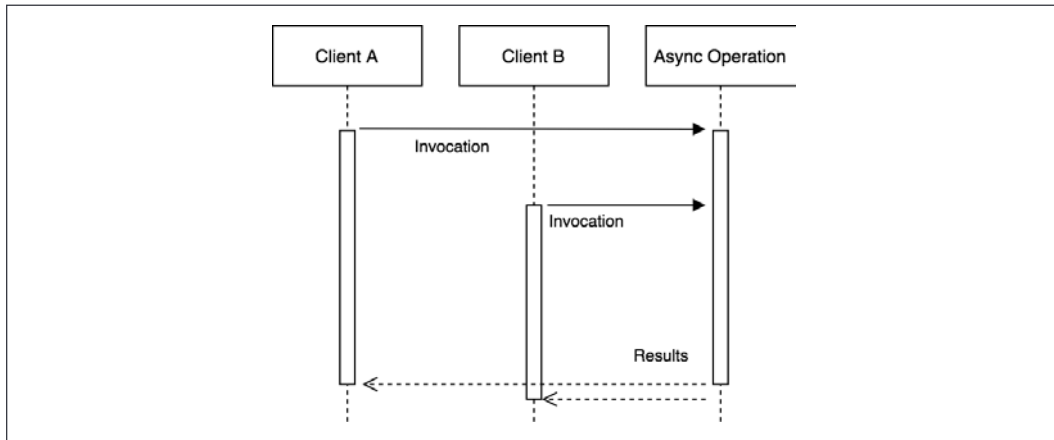


Figure 11.2: Batching of two asynchronous requests

Figure 11.2 shows us how two identical requests—which invoke the same API with the same input—can be batched, or in other words, appended to the same running operation. By doing this, when the operation completes, both clients are notified, even though the async operation is actually executed only once. This represents a simple, yet extremely powerful, way to optimize the load of an application while not having to deal with more complex caching mechanisms, which usually require an adequate memory management and invalidation strategy.

Optimal asynchronous request caching

Request batching is less effective if the operation is fast enough or if matching requests are spread across a longer period of time. Also, most of the time, we can safely assume that the result of two identical API invocations will not change so often, so simple request batching will not provide the best performance. In all these circumstances, the best candidate to reduce the load of an application and increase its responsiveness is definitely a more aggressive caching mechanism.

The idea is simple: as soon as a request completes, we store its result in the cache, which can be an in-memory variable or an item in a specialized caching server (such as Redis). Hence, the next time the API is invoked, the result can be retrieved immediately from the cache, instead of spawning another request.

The idea of caching should not be new to an experienced developer, but what makes this technique different in asynchronous programming is that it should be combined with request batching to be optimal. The reason for this is because multiple requests might run concurrently while the cache is not set and when those requests complete, the cache would be set multiple times.

Based on these assumptions, we can illustrate the Combined Request Batching and Caching pattern as follows:

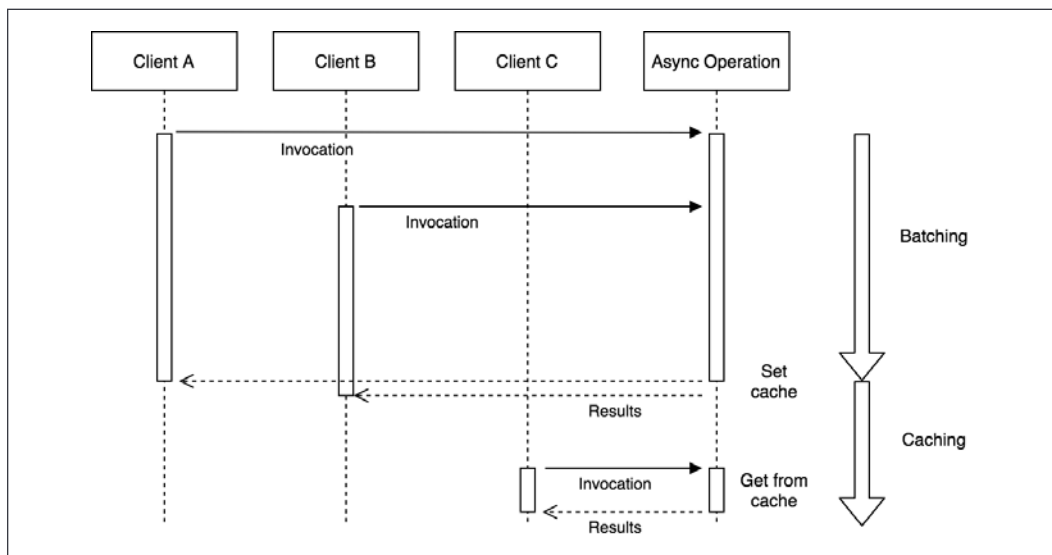


Figure 11.3: Combined batching and caching

The preceding figure shows the two phases of an optimal asynchronous caching algorithm:

- The first phase is totally identical to the batching pattern. Any request received while the cache is not set will be batched together. When the request completes, the cache is set, once.
- When the cache is finally set, any subsequent request will be served directly from it.

Another crucial detail to consider is the *Zalgo* anti-pattern (as we saw 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 LevelUP database through the `level` npm package (nodejssdp.link/level). The data model that we are going to use is a simple list of transactions stored in the `sales` sublevel (a subsection of the 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 really 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'
import sublevel from 'subleveldown'

const db = level('example-db')
const salesDb = sublevel(db, 'sales', { valueEncoding: 'json' })

export async function totalSales (product) {
  const now = Date.now()
  let sum = 0
  for await (const transaction of salesDb.createValueStream()) {
    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 sublevel 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 on. In a real-world application, we would have used an index to query the transactions by product or, even better, we could have used an incremental map/reduce algorithm to continuously calculate the sum for every product

We can now expose the `totalSales()` API through a simple HTTP server (the `server.js` file):

```
import { createServer } from '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'))
```

Before we start the server for the first time, we need to populate the database with some sample data. We can do this with the `populateDb.js` script, which can be found in this book's code repository in the folder dedicated to this section. This script creates 100,000 random sales transactions in the database so that our query spends some time crunching data:

```
node populateDb.js
```

Okay! Now, everything is ready. Let's start the server:

```
node server.js
```

To query the server, you can simply navigate with a browser to the following URL:

```
http://localhost:8000?product=book
```

However, to have a better idea of the performance of our server, we will need more than one request. So, we will use a small script named `loadTest.js`, which sends 20 requests at intervals of 200 ms. The script can be found in the code repository of this book and it's already configured to connect to the local URL of the server, so, to run it, just execute the following command:

```
node loadTest.js
```

We will see that the 20 requests will take a while to complete. Take note of the total execution time of the test. 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 in the total sales web server

Let's now add a batching layer on top of our `totalSales` API. The pattern we are going to use is very simple: if there is another identical request pending when the API is invoked, we will wait for that request to complete instead of launching a new one. As we will see, this can easily be implemented with promises. In fact, all we have to do is save the promise in a map, associating it to the specified request parameters (the product type, in our case) every time we launch a new request. Then, at every subsequent request, we check if there is already a promise for the specified product and if there is one, we just return it; otherwise, we launch a new request.

Now, let's see how this translates into code. Let's create a new module named `totalSalesBatch.js`. Here, we're going to 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(() => {
    runningRequests.delete(product)
  })

  return resultPromise
}
```

The `totalSales()` function of the `totalSalesBatch` module is 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. 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, thus 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) => {
  // ...
})
```

If we now try to start the server again and run the load test against it, the first thing we will see is that the requests are returned in *batches*. This is the effect of the recipe we just implemented, and it's a great practical demonstration of how it works.

Besides that, we should also observe a considerable reduction in the total time for executing the test. It should be at least four times faster than the original test performed against the plain `totalSales()` API!

This result substantiates the huge performance boost that we can obtain by just applying a simple batching layer, without all the complexity of managing a full-fledged cache and, more importantly, without worrying about 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.

Caching requests in the total sales web server

Adding a caching layer to our batching API is straightforward, thanks to promises. All we have to 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 code that enables caching is highlighted. All we have to 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, the server can be started again and profiled using the `loadTest.js` script, as we did in the previous examples. With the default test parameters, we should see a 10% reduction in the execution time compared to simple batching. Of course, this is highly dependent on a lot of factors; for example, the number of requests received and the delay between one request and the other. The advantages of using caching over batching will be much more substantial when the number of requests is higher and spans a longer period of time.

Notes about implementing caching mechanisms

We must remember that in real-life applications, we may want to use more advanced cache invalidation techniques and storage mechanisms. This is necessary for the following reasons:

- A large amount of cached values can easily consume a lot of memory. In this case, a **least recently used (LRU)** or a **first in first out (FIFO)** policy can be applied to maintain constant memory utilization.
- When the application is distributed across multiple processes, keeping the cache in memory may produce different results across each server instance. If that's undesired for the particular application we are implementing, the solution is to use a shared store for the cache. This is also more performant than a simple in-memory solution as the cache is shared across multiple instances. Popular caching solutions include Redis (nodejsdp.link/redis) and Memcached (nodejsdp.link/memcached).
- A manual cache invalidation (for example, when the related non-cached value changes), as opposed to a timed expiry, can enable a longer-living cache and at the same time provide more up-to-date data, but, of course, it would be a lot more complex to manage. Let's not forget the famous quote by Phil Karlton (principal engineer at Netscape, Silicon Graphics, and more): "There are only two hard things in Computer Science: cache invalidation and naming things."

With this, we conclude this section on request batching and caching. Next, we are going to learn how to tackle a tricky business: canceling asynchronous operations.

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'll be talking about canceling asynchronous operations and not about canceling promises, which is a different matter altogether. By the way, the Promises/A+ standard doesn't include an API for canceling promises. However, you can use a third-party promise library such as bluebird if you need such a feature (more at nodejsdp.link/bluebird-cancellation). Note that canceling a promise doesn't mean that the operation the promise refers to will also be canceled; in fact, bluebird offers an `onCancel` callback in the promise constructor, in addition to `resolve` and `reject`, which can be used to cancel the underlying async operation when the promise is canceled. This is actually what this section is about.

A basic recipe for creating cancelable functions

Actually, in asynchronous programming, the basic principle for canceling the execution of a function is very simple: we check if 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.

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.

The following code demonstrates how we can cancel the `cancelable()` function:

```
const cancelObj = { cancelRequested: false }
cancelable(cancelObj)
  .catch(err => {
    if (err instanceof CancelError) {
      console.log('Function canceled')
    } else {
      console.error(err)
    }
  })

setTimeout(() => {
  cancelObj.cancelRequested = true
}, 100)
```

As we can see, all we have to 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 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 cancelation 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 cancelWrapper (func, ...args) {
    if (cancelRequested) {
      return Promise.reject(new CancelError())
    }
    return func(...args)
  }

  return { cancelWrapper, cancel }
}
```

Our wrapper is created through a factory function called `createCancelWrapper()`. The factory returns two functions: the wrapper function (`cancelWrapper`) and a function to trigger the cancelation of the asynchronous operation (`cancel`). This allows us to create a wrapper function to wrap multiple asynchronous invocations and then use a single `cancel()` function to cancel all of them.

The `cancelWrapper()` function takes, as input, a function to invoke (`func`) and a set of parameters to pass to the function (`args`). The wrapper simply checks if a cancelation 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`.

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 (cancelWrapper) {
  const resA = await cancelWrapper(asyncRoutine, 'A')
  console.log(resA)
  const resB = await cancelWrapper(asyncRoutine, 'B')
```

```

    console.log(resB)
    const resC = await cancelWrapper(asyncRoutine, 'C')
    console.log(resC)
  }

  const { cancelWrapper, cancel } = createCancelWrapper()

  cancelable(cancelWrapper)
    .catch(err => {
      if (err instanceof CancelError) {
        console.log('Function canceled')
      } else {
        console.error(err)
      }
    })

  setTimeout(() => {
    cancel()
  }, 100)

```

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 generators

The cancelable wrapper function we just created is already a big step ahead compared to embedding the cancellation logic directly in our code. However, it's still not ideal for two reasons: it is error prone (what if we forget to wrap one function?) and it still affects the readability of our code, which makes it not ideal for implementing cancelable asynchronous operations that are already large and complex.

An even neater solution involves the use of generators. In *Chapter 9, Behavioral Design Patterns*, we introduced generators as a means to implement iterators. However, they are a very versatile tool and can be used to implement all sorts of algorithms. In this case, we will be using generators to build a supervisor to control the asynchronous flow of a function. The result will be an asynchronous function that is transparently cancelable, whose behavior resembles an async function in which the `await` instruction is replaced by `yield`.

Let's see how we can implement this cancelable function using generators (the `createAsyncCancelable.js` file):

```
import { CancelError } from './cancelError.js'

export function createAsyncCancelable (generatorFunction) { // (1)
  return function asyncCancelable (...args) {
    const generatorObject = generatorFunction(...args) // (3)
    let cancelRequested = false

    function cancel () {
      cancelRequested = true
    }

    const promise = new Promise((resolve, reject) => {
      async function nextStep (prevResult) { // (4)
        if (cancelRequested) {
          return reject(new CancelError())
        }

        if (prevResult.done) {
          return resolve(prevResult.value)
        }

        try { // (5)
          nextStep(generatorObject.next(await prevResult.value))
        } catch (err) {
          try { // (6)
            nextStep(generatorObject.throw(err))
          } catch (err2) {
            reject(err2)
          }
        }
      }

      nextStep({})
    })

    return { promise, cancel } // (2)
  }
}
```

The `createAsyncCancelable()` function may seem complex, so let's analyze it in more detail:

1. First, we should note that the `createAsyncCancelable()` function takes, as input, a generator function (the supervised function) and returns another function (`asyncCancelable()`) that wraps the generator function with our supervising logic. The `asyncCancelable()` function is what we will use to invoke the asynchronous operation.
2. The `asyncCancelable()` function returns an object with two properties:
 - a. The `promise` property, which contains the promise representing the eventual resolution (or rejection) of the asynchronous operation.
 - b. The `cancel` property, which is a function that can be used to cancel the supervised asynchronous flow.
3. When invoked, the first task of `asyncCancelable()` is to invoke the generator function with the arguments received as input (`args`) and obtain a generator object, which we can use to control the execution flow of the running coroutine.
4. The entire logic of the supervisor is implemented within the `nextStep()` function, which is responsible for iterating over the values yielded by the supervised coroutine (`prevResult`). Those can be actual values or promises. If a cancelation is requested, we throw the usual `CancelError`; otherwise, if the coroutine has been terminated (for example, `prevResult.done` is `true`), we immediately resolve the outer promise and complete the return.
5. The core part of the `nextStep()` function is where we retrieve the next value yielded by the supervised coroutine (which, let's not forget, it's a generator). We `await` on that value so we can make sure we get the actual resolution value in case we are dealing with a promise. This also makes sure that if `prevResult.value` is a promise and it rejects, we end up in the `catch` statement. We can end up in the `catch` statement even if the supervised coroutine actually throws an exception.
6. In the `catch` statement, we throw the caught error inside the coroutine. This is redundant if that error was already thrown by the coroutine, but not if it's the result of a promise rejection. Even if not optimal, this trick can simplify our code a bit for the sake of this demonstration. We invoke `nextStep()` using whatever value is yielded next after throwing it inside the coroutine, but if the result is another exception (for example, the exception is not caught inside the coroutine or another one is thrown), we immediately reject the outer promise and complete the asynchronous operation.

As we saw, there are a lot of moving parts in the `createAsyncCancelable()` function. But we should appreciate the fact that, in just a few lines of code, we were able to create a cancelable function that doesn't require any manual cancelation logic. As we will see now, the results are impressive.

Let's rewrite our sample asynchronous cancelable operation using the supervisor we implemented in the `createAsyncCancelable()` function:

```
import { asyncRoutine } from './asyncRoutine.js'
import { createAsyncCancelable } from './createAsyncCancelable.js'
import { CancelError } from './cancelError.js'

const cancelable = createAsyncCancelable(function * () {
  const resA = yield asyncRoutine('A')
  console.log(resA)
  const resB = yield asyncRoutine('B')
  console.log(resB)
  const resC = yield asyncRoutine('C')
  console.log(resC)
})

const { promise, cancel } = cancelable()
promise.catch(err => {
  if (err instanceof CancelError) {
    console.log('Function canceled')
  } else {
    console.error(err)
  }
})

setTimeout(() => {
  cancel()
}, 100)
```

We can immediately see that the generator wrapped by `createAsyncCancelable()` closely resembles an async function, but we are using `yield` instead of `await`. Also, there is no visible cancelation logic at all. The generator function maintains the excellent properties of async functions (for example, to make asynchronous code look like synchronous code), but unlike the async function and thanks to the supervisor introduced by `createAsyncCancelable()`, it's also possible to cancel the operation.

The second interesting aspect is that `createAsyncCancelable()` creates a function (called `cancelable`) that can be invoked like any other function but at the same time returns a promise representing the result of the operation and a function to cancel the operation.

This technique of using generators represents the best option we have to implement cancelable asynchronous operations.



For use in production, most of the time, we can rely on a widely used package from the Node.js ecosystem such as `caf` (the acronym means Cancelable Async Flows), which you can find at nodejsdp.link/caf.

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 a synchronous task that takes a long time to complete and that never gives back the 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 task behave in Node.js.

Solving the subset sum problem

Let's now choose a computationally expensive problem to use as a base for our experiment. A good candidate is the **subset sum** problem, which decides whether a set (or multiset) of integers contains a non-empty subset that has a sum equal to zero. For example, if we had as input the set `[1, 2, -4, 5, -3]`, the subsets satisfying the problem are `[1, 2, -3]` and `[2, -4, 5, -3]`.

The simplest algorithm is the one that checks every possible combination of subsets of any size and has a computational cost of $O(2^n)$, or in other words, it grows exponentially with the size of the input. This means that a set of 20 integers would require up to 1,048,576 combinations to be checked, which is not bad for testing our assumptions. For our example, we are going to consider the following variation of the subset sum problem: given a set of integers, we want to calculate all the possible combinations whose sum is equal to a given arbitrary integer, not just zero.

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`:

```
export class SubsetSum extends EventEmitter {
  constructor (sum, set) {
    super()
    this.sum = sum
    this.set = set
    this.totalSubsets = 0
  }
  //...
```

The `SubsetSum` class is extending `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.

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)
  }
}
```

Trivially, the `_processSubset()` method applies a `reduce` operation to the subset in order 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 have to expose the algorithm we just created over the network. As always, we can use a simple HTTP server for this task. In particular, 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 '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.write(`Match: ${JSON.stringify(match)}\n`)
  })
  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 than `/subsetSum`. We will use this for checking the responsiveness of our server, as we will see in a moment.

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"
```

After a few seconds, 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 actually 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 in order 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."

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 have to 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 is 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 have to 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 request.

The last missing bit is updating the `index.js` module so that it can use the new version of the `SubsetSum` API. This is actually a trivial change:

```
import { createServer } from '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 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.



Note 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 gives its best when running I/O-intensive applications such as web servers, which allows us to optimize resource utilization thanks to its asynchronous architecture. So, the best way we have to maintain the responsiveness of an application is to not run expensive CPU-bound tasks in the context of the main application and, instead, use separate processes. 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 like Go or Rust. Always use the best tool for the job!

Node.js has an ample toolbox of APIs for interacting with external processes. We can find all we need in the `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 also 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 **denial-of-service (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.



The purpose of a DoS attack is to make a machine unavailable to its users. This is usually achieved by exhausting the capacity of such a machine by exploiting a vulnerability or massively overloading it with requests (DDoS – distributed DoS).

Implementing a process pool

Let's start by building the `processPool.js` module piece by piece:

```
import { fork } from '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 `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 very simple and is explained as follows:

1. If we have a process in the pool ready to be used, we simply 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 have to 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 time 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 as the details we could add are really endless.

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 'events'
import { dirname, join } from 'path'
import { fileURLToPath } from 'url'
import { ProcessPool } from './processPool.js'

const __dirname = dirname(fileURLToPath(import.meta.url))
const workerFile = join(__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 if 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 nodejsdp.link/childprocess-send). This is actually 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 an event is received from the running algorithm, 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 simply have to replace the module used by the HTTP server (the `index.js` file):

```
import { createServer } from '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!`

More interestingly, we can also try to start two `SubsetSum` tasks concurrently and we will be able to see that they will use the full power of two different processors in order to run (if your system has more than one processor, of course). Instead, if we try to run three `SubsetSum` tasks concurrently, the result should be that the last one to start will hang. This is not because the event loop of the main process is blocked, but because we set a concurrency limit of two processes for the `SubsetSum` task, which means that the third request will be handled as soon as at least one of the two processes in the pool becomes available again.

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.

Even though worker threads are based on real threads, they don't allow the deep synchronization and sharing capabilities supported by other languages such as Java or Python. This is because JavaScript is a single-threaded language and it doesn't have any built-in mechanism to synchronize access to variables from multiple threads. JavaScript with threads simply wouldn't be JavaScript. The solution to bring all the advantages of threads within Node.js without actually changing the language is worker 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 similar to 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 '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 '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 module `subsetSumThreads.js` 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 'events'
import { dirname, join } from 'path'
import { fileURLToPath } from 'url'
import { ThreadPool } from './threadPool.js'

const __dirname = dirname(fileURLToPath(import.meta.url))
const workerFile = join(__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 full-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 '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.

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 from 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 weapons to our toolbelt, 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

In its early days, Node.js was just a non-blocking web server written in C++ and JavaScript and was called web.js. Its creator, Ryan Dahl, soon realized the potential of the platform and started extending it with tools to enable the creation of different types of server-side applications on top of JavaScript and the non-blocking paradigm.

The characteristics of Node.js are perfect for the implementation of distributed systems, ranging from a few nodes to thousands of nodes communicating through the network: Node.js was born to be distributed.

Unlike in other web platforms, scalability is a topic that gets explored rather quickly in Node.js while developing an application. This is often because of the single-threaded nature of Node.js, which is incapable of exploiting all the resources of a multi-core machine. But this is just one side of the coin. In reality, there are more profound reasons for talking about scalability with Node.js.

As we will see in this chapter, scaling an application does not only mean increasing its capacity, enabling it to handle more requests faster: it's also a crucial path to achieving high availability and tolerance to errors.

Sometimes, we even refer to scalability when we talk about ways to split the complexity of an application into more manageable pieces. Scalability is a concept with multiple faces, six to be precise, as many as there are faces on a cube — the **scale cube**.

In this chapter, you will learn the following topics:

- Why you should care about scalability
- What the scale cube is and why it is useful to understand scalability
- How to scale by running multiple instances of the same application
- How to leverage a load balancer when scaling an application
- What a service registry is and how it can be used
- Running and scaling Node.js applications using container orchestration platforms like Kubernetes
- How to design a microservice architecture out of a monolithic application
- How to integrate a large number of services through the use of some simple architectural patterns

An introduction to application scaling

Scalability can be described as the capability of a system to grow and adapt to ever-changing conditions. Scalability is not limited to pure technical growth; it is also dependent on the growth of a business and the organization behind it.

If you are building the next "unicorn startup" and you expect your product to rapidly reach millions of users worldwide, you will face serious scalability challenges. How is your application going to sustain ever-increasing demand? Is the system going to get slower over time or crash often? How can you store high volumes of data and keep I/O under control? As you hire more people, how can you organize the different teams effectively and make them able to work autonomously, without contention across the different parts of the codebase?

Even if you are not working on a high-scale project, that doesn't mean that you will be free from scalability concerns. You will just face different types of scalability challenges. Being unprepared for these challenges might seriously hinder the success of the project and ultimately damage the company behind it. It's important to approach scalability in the context of the specific project and understand the expectations for current and future business needs.

Since scalability is such a broad topic, in this chapter, we will focus our attention on discussing the role of Node.js in the context of scalability. We will discuss several useful patterns and architectures used to scale Node.js applications.

With these in your toolbelt and a solid understanding of your business context, you will be able to design and implement Node.js applications that can adapt and satisfy your business needs and keep your customers happy.

Scaling Node.js applications

We already know that most of the workload of a typical Node.js application runs in the context of a single thread. In *Chapter 1, The Node.js Platform*, we learned that this is not necessarily a limitation but rather an advantage, because it allows the application to optimize the usage of the resources necessary to handle concurrent requests, thanks to the non-blocking I/O paradigm. This model works wonderfully for applications handling a moderate number of requests per second (usually a few hundred per second), especially if the application is mostly performing I/O-bound tasks (for example, reading and writing from the filesystem and the network) rather than CPU-bound ones (for example, number crunching and data processing).

In any case, assuming we are using commodity hardware, the capacity that a single thread can support is limited. This is regardless of how powerful a server can be, so if we want to use Node.js for high-load applications, the only way is to scale it across multiple processes and machines.

However, workload is not the only reason to scale a Node.js application. In fact, with the same techniques that allow us to scale workloads, we can obtain other desirable properties such as high **availability** and **tolerance to failures**. Scalability is also a concept applicable to the size and complexity of an application. In fact, building architectures that can grow as much as needed over time is another important factor when designing software.

JavaScript is a tool to be used with caution. The lack of type checking and its many gotchas can be an obstacle to the growth of an application, but with discipline and accurate design, we can turn some of its downsides into precious advantages. With JavaScript, we are often pushed to keep the application simple and to split its components into small, manageable pieces. This mindset can make it easier to build applications that are distributed and scalable, but also easy to evolve over time.

The three dimensions of scalability

When talking about scalability, the first fundamental principle to understand is **load distribution**, which is the science of splitting the load of an application across several 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 proposes an ingenious model to represent them, called the **scale cube**. This model describes scalability in terms of the following three 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*:

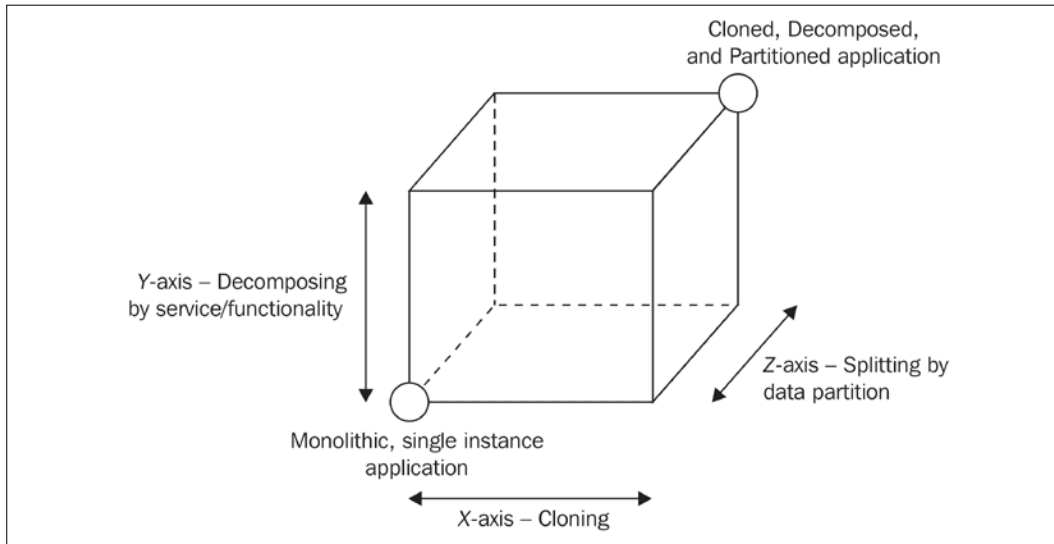


Figure 12.1: The scale cube

The bottom-left corner of the cube (that is, the intersection between the X-axis and the Y-axis) represents the application having all the functionality in a single code base and running on a single instance. This is what we generally call a **monolithic application**. This is a common situation for applications handling small workloads or at the early stages of their development. Given a monolithic application, there are three different strategies for scaling it. By looking at the scale cube, these strategies are represented as growth along the different axes of the cube: X, Y, and Z:

- **X-axis – Cloning:** The most intuitive evolution of a monolithic, unscaled application is moving right along the X-axis, which is simple, most of the time inexpensive (in terms of development cost), and highly effective. The principle behind this technique is elementary, that is, cloning the same application n times and letting each instance handle $1/n$ th of the workload.
- **Y-axis – Decomposing by service/functionality:** Scaling along the Y-axis means decomposing the application based on its functionalities, services, or use cases. In this instance, *decomposing* means creating different, standalone applications, each with its own codebase, possibly with its own dedicated database, and even with a separate UI.

For example, a common situation is separating the part of an application responsible for the administration from the public-facing product. Another example is extracting the services responsible for user authentication, thereby creating a dedicated authentication server.

The criteria to split an application by its functionalities depend mostly on its business requirements, the use cases, the data, and many other factors, as we will see later in this chapter. Interestingly, this is the scaling dimension with the biggest repercussions, not only on the architecture of an application but also on the way it is managed from a development and an operational perspective. As we will see, **microservice** is a term that is most commonly associated with a fine-grained Y-axis scaling.

- **Z-axis – Splitting by data partition:** The last scaling dimension is the Z-axis, where the application is split in such a way that each instance is responsible for only a portion of the whole data. This is a technique often used in databases, also known as **horizontal/vertical partitioning**. In this setup, there are multiple instances of the same application, each of them operating on a partition of the data, which is determined using different criteria.

For example, we could partition the users of an application based on their country (*list partitioning*), or based on the starting letter of their surname (*range partitioning*), or by letting a hash function decide the partition each user belongs to (*hash partitioning*).

Each partition can then be assigned to a particular instance of the application. The use of data partitions requires each operation to be preceded by a lookup step to determine which instance of the application is responsible for a given datum. As we said, data partitioning is usually applied and handled at the data storage level because its main purpose is overcoming the problems related to handling large monolithic datasets (limited disk space, memory, and network capacity). Applying it at the application level is worth considering only for complex, distributed architectures or for very particular use cases such as, for example, when building applications relying on custom solutions for data persistence, when using databases that don't support partitioning, or when building applications at Google scale. Considering its complexity, scaling an application along the Z-axis should be taken into consideration only after the X and Y axes of the scale cube have been fully exploited.

In the following sections, we will focus on the two most common and effective techniques used to scale Node.js applications, namely, **cloning** and **decomposing** by functionality/service.

Cloning and load balancing

Traditional, multithreaded web servers are usually only scaled horizontally when the resources assigned to a machine cannot be upgraded any more, or when doing so would involve a higher cost than simply launching another machine.

By using multiple threads, traditional web servers can take advantage of all the processing power of a server, using all the available processors and memory. Conversely, Node.js applications, being single-threaded, are usually scaled much sooner compared to traditional web servers. Even in the context of a single machine, we need to find ways to "scale" an application in order to take advantage of all the available resources.



In Node.js, **vertical scaling** (adding more resources to a single machine) and **horizontal scaling** (adding more machines to the infrastructure) are almost equivalent concepts: both, in fact, involve similar techniques to leverage all the available processing power.

Don't be fooled into thinking about this as a disadvantage. On the contrary, being almost forced to scale has beneficial effects on other attributes of an application, in particular, availability and fault-tolerance. In fact, scaling a Node.js application by cloning is relatively simple and it's often implemented even if there is no need to harvest more resources, just for the purpose of having a redundant, fail-tolerant setup.

This also pushes the developer to take into account scalability from the early stages of an application, making sure the application does not rely on any resource that cannot be shared across multiple processes or machines. In fact, an absolute prerequisite to scaling an application is that each instance does not have to store common information on resources that cannot be shared, such as memory or disk. For example, in a web server, storing the session data in memory or on disk is a practice that does not work well with scaling. Instead, using a shared database will ensure that each instance will have access to the same session information, wherever it is deployed.

Let's now introduce the most basic mechanism for scaling Node.js applications: the `cluster` module.

The cluster module

In Node.js, the simplest pattern to distribute the load of an application across different instances running on a single machine is by using the `cluster` module, which is part of the core libraries. The `cluster` module simplifies the forking of new instances of the same application and automatically distributes incoming connections across them, as shown in *Figure 12.2*:

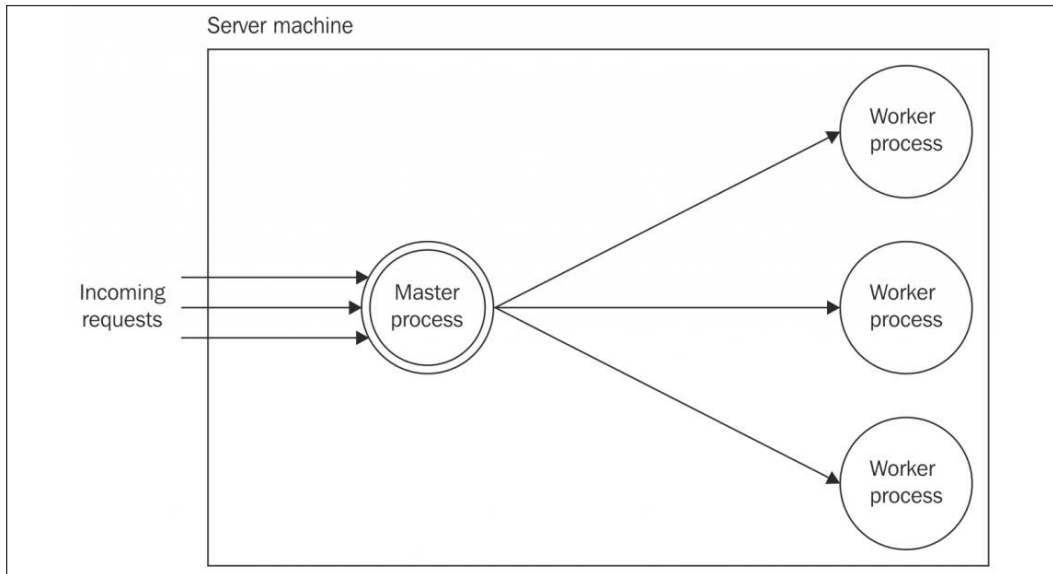


Figure 12.2: Cluster module schematic

The **master 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 `cluster` module uses an explicit round-robin load balancing algorithm. This algorithm is used inside the master process, which makes sure the requests are evenly distributed across all the workers. 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 servers on a rotational basis. The first request is forwarded to the first server, the second to the next server in the list, and so on. When the end of the list is reached, the iteration starts again from the beginning. In the `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 `cluster` module, every invocation to `server.listen()` in a worker process is delegated to the master process. This allows the master 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 master process. One way to overcome this limitation is to create the file descriptor in the master 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 master.
- `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 master 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 by 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 `cluster` module. First of all, 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 'http'

const { pid } = process
const server = createServer((req, res) => {
  // simulates CPU intensive work
  let i = 1e7; while (i > 0) { i-- }

  console.log(`Handling request from ${pid}`)
  res.end(`Hello from ${pid}\n`)
})

server.listen(8080, () => console.log(`Started at ${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 will 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 like 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 a network benchmarking tool such as `autocannon` (nodejssdp.link/autocannon):

```
npx autocannon -c 200 -d 10 http://localhost:8080
```

The preceding command will load the server with 200 concurrent connections for 10 seconds. As a reference, the result we got on our machine (a 2.5 GHz quad-core Intel Core i7 using Node.js v14) is in the order of 300 transactions per second.



Please remember that the load tests we will perform in this chapter are intentionally simple and minimal and are provided only for reference and learning purposes. Their results cannot provide a 100% accurate evaluation of the performance of the various techniques we are analyzing. When you are trying to optimize a real production application, make sure to always run your own benchmarks after every change. You might find out that, among the different techniques we are going to illustrate here, some can be more effective than others for your specific application.

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 `cluster` module:

```
import { createServer } from 'http'
import { cpus } from 'os'
import cluster from 'cluster'

if (cluster.isMaster) {                                     // (1)
  const availableCpus = cpus()
  console.log(`Clustering to ${availableCpus.length} processes`)
  availableCpus.forEach(() => cluster.fork())
} else {                                                    // (2)
  const { pid } = process
  const server = createServer((req, res) => {
    let i = 1e7; while (i > 0) { i-- }
    console.log(`Handling request from ${pid}`)
    res.end(`Hello from ${pid}\n`)
  })

  server.listen(8080, () => console.log(`Started at ${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 actually executing the master process. In this case, the `cluster.isMaster` 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.isMaster` 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 `cluster` module is based on a recurring pattern, which makes it very easy to run multiple instances of an application:

```
if (cluster.isMaster) {  
  // 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 master'))
```

Now, let's try to run our HTTP server in cluster mode. If our machine has more than one core, we should see a number of 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:

```
npx autocannon -c 200 -d 10 http://localhost:8080
```

This way, we should be able to discover the performance increase obtained by scaling our application across multiple processes. As a reference, in our machine, we saw a performance increase of about 3.3x (1,000 trans/sec versus 300 trans/sec).

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; however, it is the application's responsibility to manage the worker pool based on its own needs.

As we already mentioned, 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 pretty straightforward to implement using the `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
  setTimeout(
    () => { throw new Error('Oops') },
    Math.ceil(Math.random() * 3) * 1000
  )
// ...
```

With this change in place, our server exits with an error after a random number of seconds between 1 and 3. In a real-life situation, this would eventually cause our application to stop serving requests, unless we use some external tool to monitor its status and restart it automatically. However, if we only have one instance, there may be a non-negligible delay between restarts caused by the startup time of the application. This means that during those restarts, the application is not available. Having multiple instances instead will make sure we always have a backup process to serve an incoming request, even when one of the workers fails.

With the `cluster` module, all we have to do is spawn a new worker as soon as we detect that one is terminated with an error code. Let's modify `app.js` to take this into account:

```
// ...
if (cluster.isMaster) {
  // ...
  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 preceding code, as soon as the master process receives an 'exit' event, we check whether the process is terminated intentionally or as the result of an error. We do this by checking the status code and the flag `worker.exitedAfterDisconnect`, which indicates whether the worker was terminated explicitly by the master. If we confirm that the process was terminated because of an error, we start a new worker. It's interesting to note that while the crashed worker gets replaced, the other workers can still serve requests, thus not affecting the availability of the application.

To test this assumption, we can try to stress our server again using autocannon. When the stress test completes, we will notice that among the various metrics in the output, there is also an indication of the number of failures. In our case, it is something like this:

```
[...]
8k requests in 10.07s, 964 kB read
674 errors (7 timeouts)
```

This should amount to about 92% availability. Bear in mind that this result can vary a lot as it greatly depends on the number of running instances and how many times they crash during the test, but it should give us a good indicator of how our solution works. The preceding numbers tell us that despite the fact that our application is constantly crashing, we only had 674 failed requests over 8,000 hits.

In the example scenario that we just built, most of the failing requests will be caused by the interruption of already established connections during a crash. Unfortunately, there is very little we can do to prevent these types of failures, especially when the application terminates because of a crash. Nonetheless, our solution proves to be working and its availability is not bad at all for an application that crashes so often!

Zero-downtime restart

A Node.js application might also need to be restarted when we want to release a new version to our production servers. So, also in this scenario, having multiple instances can help maintain the availability of our application.

When we have to intentionally restart an application to update it, there is a small window in which the application restarts and is unable to serve requests. This can be acceptable if we are updating our personal blog, but it's not even an option for a professional application with a **service-level agreement (SLA)** or one that is updated very often as part of a continuous delivery process. The solution is to implement a **zero-downtime restart**, where the code of an application is updated without affecting its availability.

With the `cluster` module, this is, again, a pretty easy task: the pattern involves restarting the workers one at a time. This way, the remaining workers can continue to operate and maintain the services of the application available.

Let's add this new feature to our clustered server. All we have to do is add some new code to be executed by the master process:

```
import { once } from 'events'
// ...
if (cluster.isMaster) {
  // ...
  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()                                           // (2)
      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 restarting of the workers is triggered on receiving 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 a request, this won't be interrupted abruptly; instead, it will be completed. The worker exits only after the completion of all inflight 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.



`pm2` (nodejssdp.link/pm2) is a small utility, based on `cluster`, which offers load balancing, process monitoring, zero-downtime restarts, and other goodies.

Dealing with stateful communications

The `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 `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*:

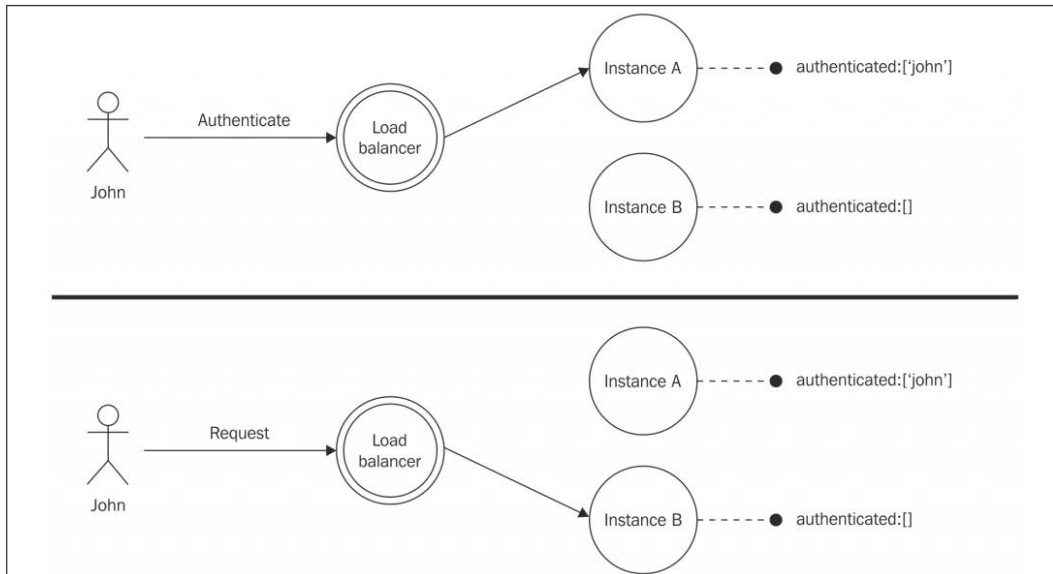


Figure 12.3: An example issue with a stateful application behind a load balancer

The user **John** initially sends a request to our application to authenticate himself, but the result of the operation is registered locally (for example, in memory), so only the instance of the application that receives the authentication request (**Instance A**) knows that John is successfully authenticated. When John sends a new request, the load balancer might forward it to a different instance of the application, which actually doesn't possess the authentication details of John, hence refusing to perform the operation. The application we just described cannot be scaled as it is, but luckily, there are two easy solutions we can apply to solve this problem.

Sharing the state across multiple instances

The first option we have to scale an application using stateful communications is sharing the state across all the instances.

This can be easily 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, even better, we can 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:

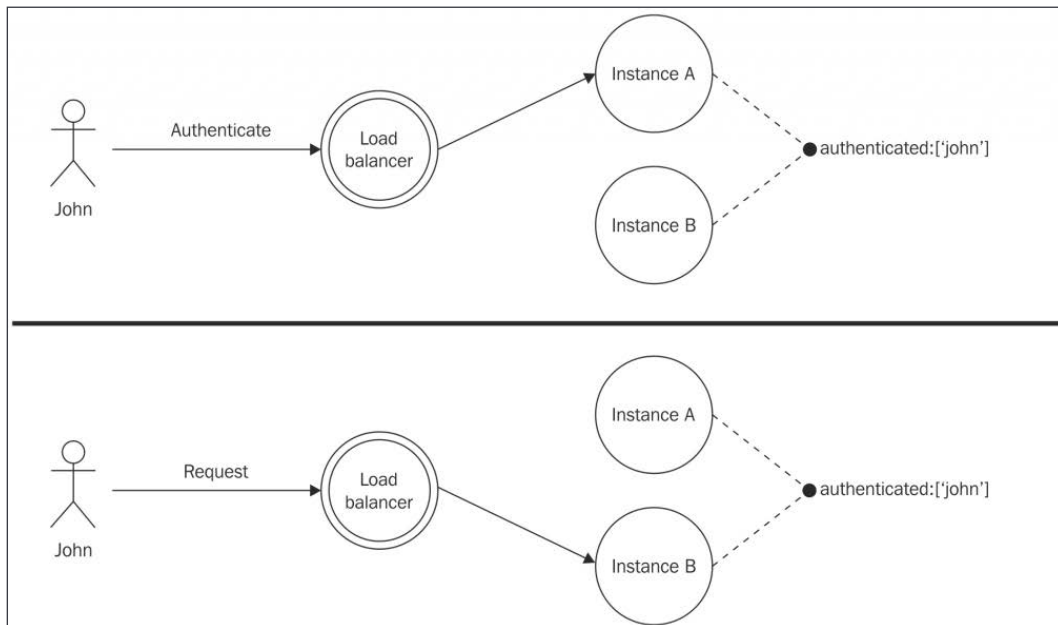


Figure 12.4: Application behind a load balancer using a shared data store

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 code base. For example, we might be using an existing library that keeps the communication state in memory, so we have to figure out how to configure, replace, or reimplement this library to use a shared store.

In cases where refactoring might not be feasible, for instance, because of too many changes required or stringent time constraints in making the application more scalable, we can rely on a less invasive solution: **sticky load balancing** (or **sticky sessions**).

Sticky load balancing

The other alternative we have to support stateful communications is having the load balancer always routing all of the requests associated with a session to the same instance of the application. This technique is also called **sticky load balancing**.

Figure 12.5 illustrates a simplified scenario involving this technique:

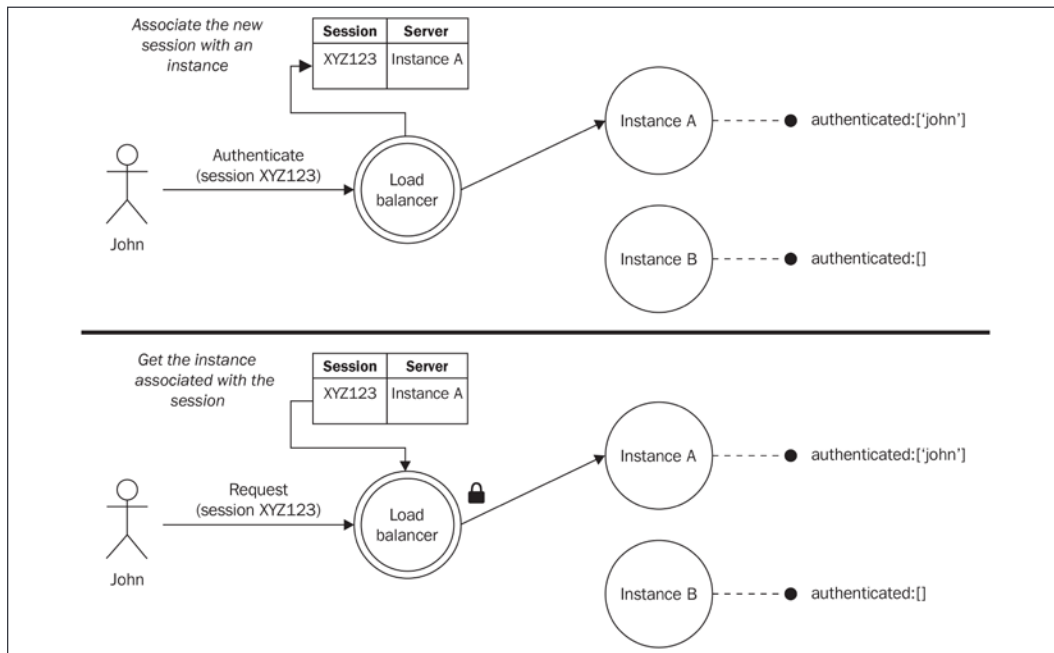


Figure 12.5: An example illustrating how sticky load balancing works

As we can see from Figure 12.5, when the load balancer receives a request associated with a new session, it creates a mapping with one particular instance selected by the load balancing algorithm. The next time the load balancer receives a request from that same session, it bypasses the load balancing algorithm, selecting the application instance that was previously associated with the session. The particular technique we just described involves inspecting the session ID associated with the requests (usually included in a cookie by the application or the load balancer itself).

A simpler alternative to associate a stateful connection to a single server is by using the IP address of the client performing the request. Usually, the IP is provided to a hash function that generates an ID representing the application instance designated to receive the request. This technique has the advantage of not requiring the association to be remembered by the load balancer. However, it doesn't work well with devices that frequently change IP, for example, when roaming on different networks.



Sticky load balancing is not supported by default by the `cluster` module, but it can be added with an npm library called `sticky-session` (nodejsdp.link/sticky-session).

One big problem with sticky load balancing is the fact that it nullifies most of the advantages of having a redundant system, where all the instances of the application are the same, and where an instance can eventually replace another one that stopped working. For these reasons, it is recommended to always try to avoid sticky load balancing and building applications that maintain session state in a shared store. Alternatively, where feasible, you can try to build applications that don't require stateful communications at all; for example, by including the state in the request itself.



For a real example of a library requiring sticky load balancing, we can mention `Socket.IO` (nodejsdp.link/socket-io).

Scaling with a reverse proxy

The `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 `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 was 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.

Figure 12.6 shows a typical multi-process, multi-machine configuration with a reverse proxy acting as a load balancer on the front:

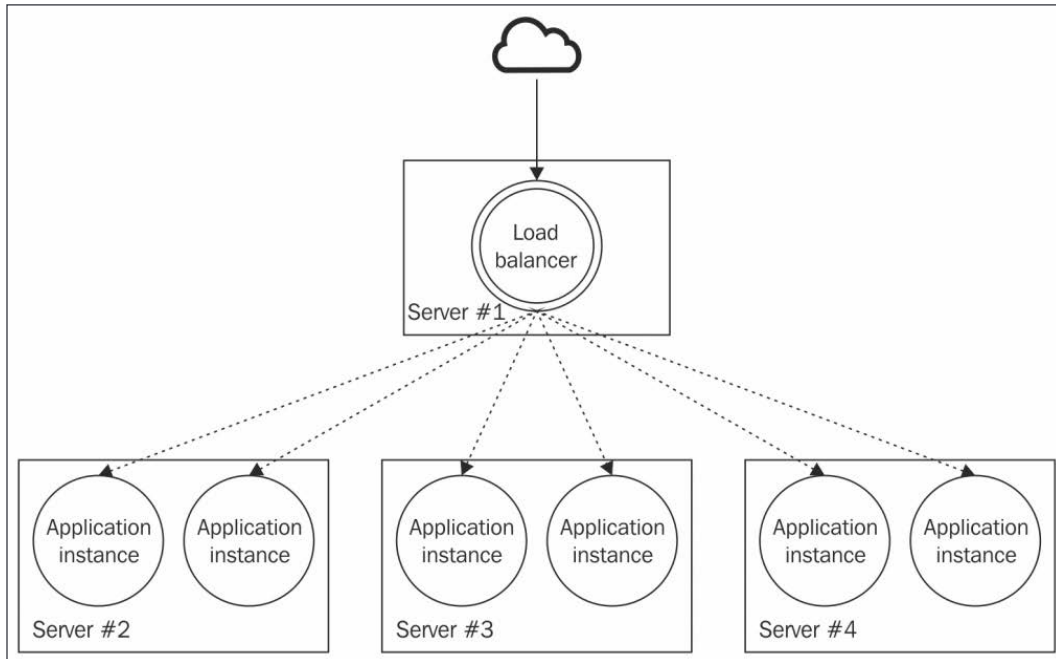


Figure 12.6: A typical multi-process, multi-machine configuration with a reverse proxy acting as a load balancer

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 `cluster` module could also be easily combined with a reverse proxy if necessary, for example, by using `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.
- **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.

In the next few sections of this chapter, we will analyze a sample configuration using Nginx. Later on, we will work on building our very own 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. We can do that by following the instructions at nodejsdp.link/nginx-install.



On the latest Ubuntu system, you can quickly install Nginx with the command `sudo apt-get install nginx`. On macOS, you can use `brew` (nodejsdp.link/brew): `brew install nginx`. Note that for the following examples, we will be using the latest version of Nginx available at the time of writing (1.17.10).

Since we are not going to use `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 'http'

const { pid } = process
const server = createServer((req, res) => {
  let i = 1e7; while (i > 0) { i-- }
  console.log(`Handling 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}`))
```

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 `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 **forever** (nodejsdp.link/forever) or **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 **forever**, which is the simplest and most immediate for us to use. We can install it globally by running the following command:

```
npm install forever -g
```

The next step is to start the four instances of our application, all on different ports and supervised by **forever**:

```
forever start app.js 8081
forever start app.js 8082
forever start app.js 8083
forever start app.js 8084
```

We can check the list of the started processes using the command:

```
forever list
```



You can use `forever stopall` to stop all the Node.js processes previously started with **forever**. Alternatively, you can use `forever stop <id>` to stop a specific process from the ones shown with `forever list`.

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 ok for the sake of this demo as we want to run just one application locally, but we advise you 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. 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 `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 this recipe:

1. Provision n backend servers running the Node.js application (running multiple instances with a service monitor like `forever` or by using the `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 like `autocannon`.



For simplicity, you can perform all these steps manually by spinning servers through your cloud provider 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** such as **Terraform** (nodejsdp.link/terraform), **Ansible** (nodejsdp.link/ansible), and **Packer** (nodejsdp.link/packer).

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

One important advantage of modern cloud-based infrastructure is the ability to dynamically adjust the capacity of an application based on the current or predicted traffic. This is also known as **dynamic scaling**. If implemented properly, this practice can reduce the cost of the IT infrastructure enormously while still keeping the application highly available and responsive.

The idea is simple: if our application is experiencing a performance degradation caused by a peak in traffic, the system automatically spawns new servers to cope with the increased load. Similarly, if we see that the allocated resources are underutilized, we can shut some servers down to reduce the cost of the running infrastructure. We could also decide to perform scaling operations based on a schedule; for instance, we could shut down some servers during certain hours of the day when we know that the traffic will be lighter, and restart them again just before the peak hours. These mechanisms require the load balancer to always be up-to-date with the current network topology, knowing at any time which server is up.

Using a service registry

A common pattern to solve this problem is to use a central repository called a **service registry**, which keeps track of the running 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:

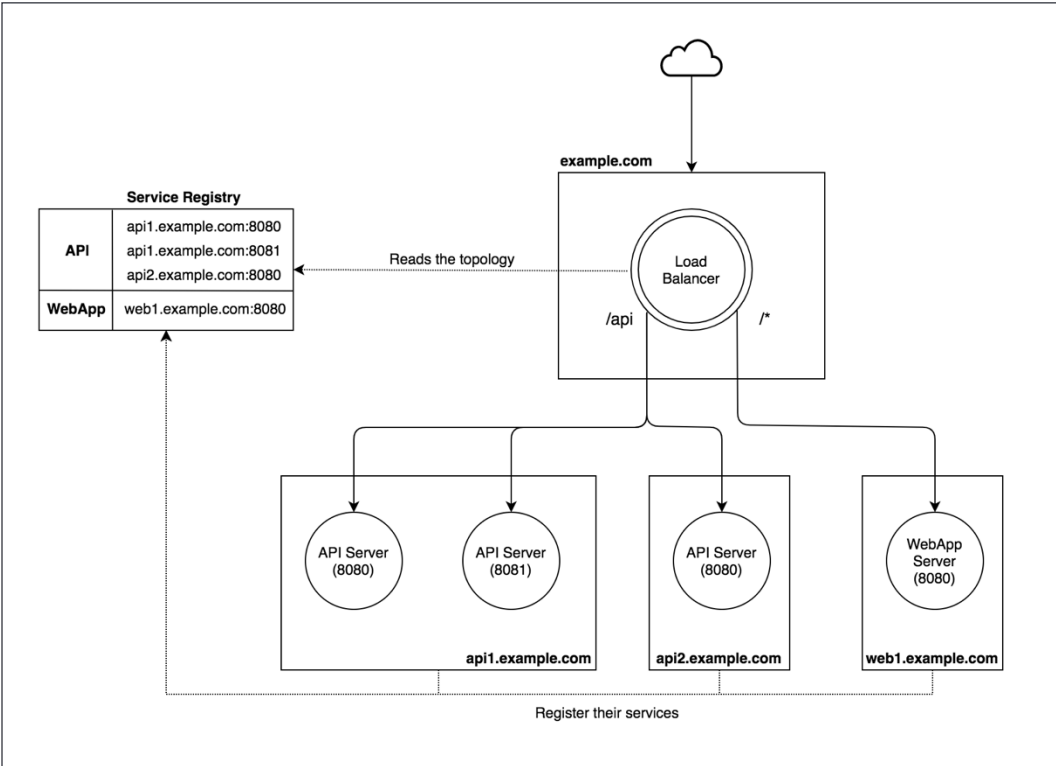


Figure 12.7: A multiservice architecture with a load balancer on the front, dynamically configured using a service registry

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 to 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. We can look at the Service Registry pattern as an implementation of the Service Locator Design pattern applied to network services.

Implementing a dynamic load balancer with http-proxy and Consul

To support a dynamic network infrastructure, we can use a reverse proxy such as **Nginx** or **HAProxy**: all we need to do is update their configuration using an automated service and then force the load balancer to pick the changes. For Nginx, this can be done using the following command line:

```
nginx -s reload
```

The same result can be achieved with a cloud-based solution, but we have a third and more familiar alternative that makes use of our favorite platform.

We all know that Node.js is a great tool for building any sort of network application and, as we said throughout this book, this is exactly one of its main design goals. So, why not build a load balancer using nothing but Node.js? This would give us much more freedom and power and would allow us to implement any sort of pattern or algorithm straight into our custom-built load balancer, including the one we are now going to explore: dynamic load balancing using a service registry. Furthermore, working on this exercise will definitely help us to understand even better how production-grade products such as Nginx and HAProxy actually work.

In this example, we are going to use **Consul** (nodejsdp.link/consul) as the service registry to replicate the multiservice architecture we saw in *Figure 12.7*. To do that, we are going to mainly use three npm packages:

- `http-proxy` (nodejsdp.link/http-proxy): 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
- `consul` (nodejsdp.link/consul-lib): To interact with Consul

Let's start by implementing our services. These are simple HTTP servers like the ones we have used so far to test cluster and Nginx, but this time, we want each server to register itself into the service registry the moment it starts.

Let's see how this looks (file `app.js`):

```
import { createServer } from 'http'
import consul from 'consul'
import portfinder from 'portfinder'
import { nanoid } from 'nanoid'

const serviceType = process.argv[2]
const { pid } = process

async function main () {
  const consulClient = consul()

  const port = await portfinder.getPortPromise() // (1)
  const address = process.env.ADDRESS || 'localhost'
  const serviceId = nanoid()

  function registerService () { // (2)
    consulClient.agent.service.register({
      id: serviceId,
      name: serviceType,
      address,
      port,
      tags: [serviceType]
    }, () => {
      console.log(`${serviceType} registered successfully`)
    })
  }
}
```

```

function unregisterService (err) {                                // (3)
  err && console.error(err)
  console.log(`deregistering ${serviceId}`)
  consulClient.agent.service.deregister(serviceId, () => {
    process.exit(err ? 1 : 0)
  })
}

process.on('exit', unregisterService)                             // (4)
process.on('uncaughtException', unregisterService)
process.on('SIGINT', unregisterService)

const server = createServer((req, res) => {                       // (5)
  let i = 1e7; while (i > 0) { i-- }
  console.log(`Handling request from ${pid}`)
  res.end(`${serviceType} response from ${pid}\n`)
})

server.listen(port, address, () => {
  registerService()
  console.log(`Started ${serviceType} at ${pid} on port ${port}`)
})
}

main().catch((err) => {
  console.error(err)
  process.exit(1)
})

```

In the preceding code, there are some parts that deserve our attention:

1. First, we use `portfinder.getPortPromise()` to discover a free port in the system (by default, `portfinder` starts to search from port 8000). We also allow the user to configure the address based on the environment variable `ADDRESS`. Finally, we generate a random ID to identify this service using `nanoid` (nodejsdp.link/nanoid).
2. Next, we declare the `registerService()` function, which uses the `consul` library to register a new service in the registry. The service definition needs several attributes: `id` (a unique identifier for the service), `name` (a generic name that identifies the service), `address` and `port` (to identify how to access the service), and `tags` (an optional array of tags that can be used to filter and group services). We are using `serviceType` (which we get from the command-line arguments) to specify the service name and to add a tag. This will allow us to identify all the services of the same type available in the cluster.

3. At this point, we define a function called `unregisterService()`, which allows us to remove the service we just registered in Consul.
4. We use `unregisterService()` as a cleanup function so that when the program is closed (either intentionally or by accident), the service is unregistered from Consul.
5. Finally, we start the HTTP server for our service on the port discovered by `portfinder` and the address configured for the current service. Note that when the server is started, we make sure to invoke the `registerService()` function to make sure that the service is registered for discovery.

With this script, we will be able to start and register different types of applications.

Now, it's time to implement the load balancer. Let's do that by creating a new module called `loadBalancer.js`:

```
import { createServer } from 'http'
import httpProxy from 'http-proxy'
import consul from 'consul'

const routing = [                                     // (1)
  {
    path: '/api',
    service: 'api-service',
    index: 0
  },
  {
    path: '/',
    service: 'webapp-service',
    index: 0
  }
]

const consulClient = consul()                         // (2)
const proxy = httpProxy.createProxyServer()

const server = createServer((req, res) => {
  const route = routing.find((route) =>               // (3)
    req.url.startsWith(route.path))
  consulClient.agent.service.list((err, services) => { // (4)
    const servers = !err && Object.values(services)
      .filter(service => service.Tags.includes(route.service))
```

```

    if (err || !servers.length) {
      res.writeHead(502)
      return res.end('Bad gateway')
    }

    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 })
  })
})

server.listen(8080, () => {
  console.log('Load balancer started on port 8080')
})

```

This is how we implemented our Node.js-based load balancer:

1. First, we define our load balancer routes. Each item in the routing array contains the service used to handle the requests arriving on the mapped path. The index property will be used to **round-robin** the requests of a given service.
2. We need to instantiate a consul client so that we can have access to the registry. Next, we instantiate an http-proxy server.
3. In the request handler of the server, the first thing we do is match the URL against our routing table. The result will be a descriptor containing the service name.
4. We obtain from consul the list of servers implementing the required service. If this list is empty or there was an error retrieving it, then we return an error to the client. We use the Tags attribute to filter all the available services and find the address of the servers that implement the current service type.
5. At last, we can route the request to its destination. We update route.index to point to the next server in the list, following a round-robin approach. We then use the index to select a server from the list, passing it to proxy.web(), along with the request (req) and the response (res) objects. This will simply forward the request to the server we chose.

It is now clear how simple it is to implement a load balancer using only Node.js and a service registry, as well as how much flexibility we can have by doing so.



Note that in order to keep the implementation simple, we intentionally left out some interesting optimization opportunities. For instance, in this implementation, we are interrogating `consul` to get the list of registered services for every single request. This is something that can add a significant overhead, especially if our load balancer receives requests with a high frequency. It would be more efficient to cache the list of services and refresh it on a regular basis (for instance, every 10 seconds). Another optimization could be to use the `cluster` module to run multiple instances of our load balancer and distribute the load across all the available cores in the machine.

Now, we should be ready to give our system a try, but first, let's install the Consul server by following the official documentation at nodejsdp.link/consul-install.

This allows us to start the Consul service registry on our development machine with this simple command line:

```
consul agent -dev
```

Now, we are ready to start the load balancer (using `forever` to make sure the application is restarted in case of a crash):

```
forever start 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 didn't start 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` and one `webapp-service`:

```
forever start --killSignal=SIGINT app.js api-service  
forever start --killSignal=SIGINT app.js api-service  
forever start --killSignal=SIGINT 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 this:

```
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:

```
api-service response from 6979
```



If you want to see the instances managed by `forever` and stop some of them you can use the commands `forever list` and `forever stop`. To stop all running instances you can use `forever stopall`. Why don't you try to stop one of the running instances of the `api-service` to see what happens to the whole application?

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!



Consul offers a convenient web UI available at `localhost:8500` by default. Check it out while playing with this example to see how services appear and disappear as they get registered or unregistered.

Consul also offers a health check feature to monitor registered services. This feature could be integrated within our example to make our infrastructure even more resilient to failures. In fact, if a service does not respond to a health check, it gets automatically removed from the registry and therefore, it won't receive traffic anymore. If you are curious to see how you can implement this feature, you can check out the official documentation for *Checks* at nodejsdp.link/consul-checks.

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, like peer-to-peer load balancing.

Peer-to-peer load balancing

Using a reverse proxy is almost a necessity when we want to expose a complex internal network architecture to a public network such as the Internet. It helps hide the complexity, providing a single access point that external applications can easily use and rely on. However, if we need to scale a service that is for internal use only, we can have much more flexibility and control.

Let's imagine having a service, **Service A**, that relies on **Service B** to implement its functionality. **Service B** is scaled across multiple machines and it's available only in the internal network. What we have learned so far is that **Service A** will connect to **Service B** using a load balancer, which will distribute the traffic to all the servers implementing **Service B**.

However, there is an alternative. We can remove the load balancer from the picture and distribute the requests directly from the client (**Service A**), which now becomes directly responsible for load balancing its requests across the various instances of **Service B**. This is possible only if **Service A** knows the details about the servers exposing **Service B**, and in an internal network, this is usually known information. With this approach, we are essentially implementing **peer-to-peer load balancing**.

Figure 12.8 compares the two alternatives we just described:

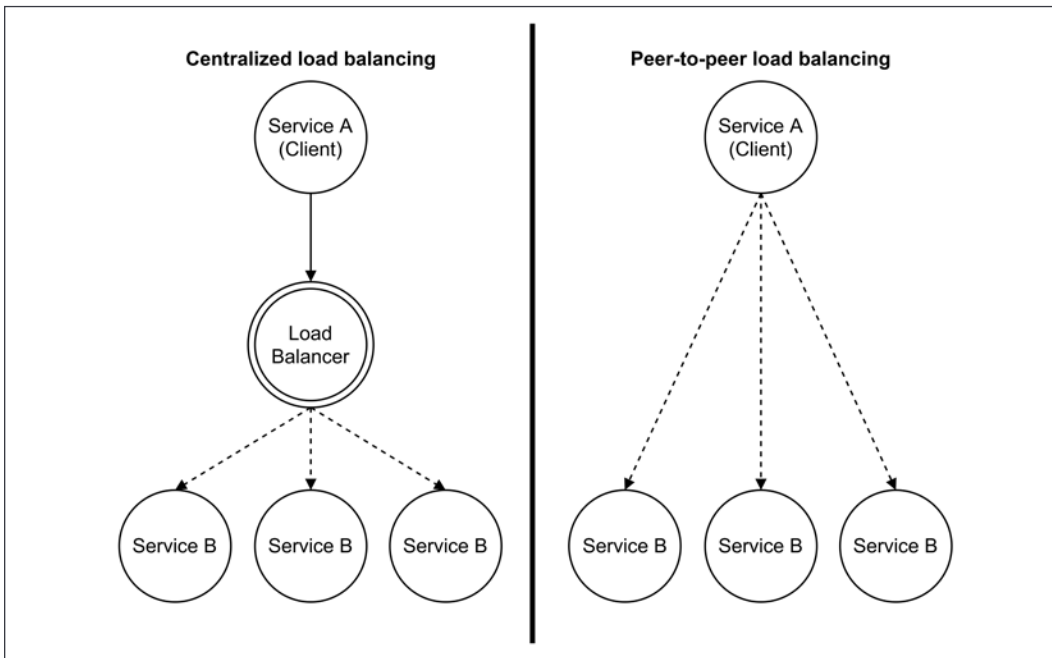


Figure 12.8: Centralized load balancing versus peer-to-peer load balancing

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:

- Reduces the infrastructure complexity by removing a network node
- Allows faster communications because messages will travel through one fewer node
- Scales better because performances are not limited by what the load balancer can handle

On the other hand, by removing the load balancer, we are actually exposing the complexity of its underlying infrastructure. Also, each client has to be smarter by implementing a load balancing algorithm and, possibly, also a way to keep its knowledge of the infrastructure up to date.



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 implementing peer-to-peer load balancing in an HTTP client.

Implementing an HTTP client that can balance requests across multiple servers

We already know how to implement a load balancer using only Node.js and distribute incoming requests across the available servers, so implementing the same mechanism on the client side should not be that different. All we have to do, in fact, is wrap the client API and augment it with a load balancing mechanism. Take a look at the following module (`balancedRequest.js`):

```
import { request } from 'http'
import getStream from 'get-stream'

const servers = [
  { host: 'localhost', port: 8081 },
  { host: 'localhost', port: 8082 }
]
let i = 0

export function balancedRequest (options) {
```



```
return new Promise((resolve) => {
  i = (i + 1) % servers.length
  options.hostname = servers[i].host
  options.port = servers[i].port

  request(options, (response) => {
    resolve(getStream(response))
  }).end()
})
}
```

The preceding code is very simple and needs little explanation. We wrapped the original `http.request` API so that it overrides the `hostname` and `port` of the request with those selected from the list of available servers using a round-robin algorithm. Note that, for simplicity, we used the module `get-stream` (nodejsdp.link/get-stream) to "accumulate" the response stream into a buffer that will contain the full response body.

The new wrapped API can then be used seamlessly (`client.js`):

```
import { balancedRequest } from './balancedRequest.js'

async function main () {
  for (let i = 0; i < 10; i++) {
    const body = await balancedRequest({
      method: 'GET',
      path: '/'
    })
    console.log(`Request ${i} completed:`, body)
  }
}

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

To run the preceding code, we have to start two instances of the sample server provided:

```
node app.js 8081
node app.js 8082
```

This is followed by the client application we just built:

```
node client.js
```

We should note that each request is sent to a different server, confirming that we are now able to balance the load without a dedicated load balancer!



An obvious 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 demonstrate how using containers and container orchestration platforms, such as Kubernetes, can help us to write simpler Node.js applications that can delegate most of the scaling concerns like load balancing, elastic scaling, and high availability to the underlying container platform.

Containers and container orchestration platforms constitute a quite broad topic, largely outside the scope of this book. For this reason, here, we aim to provide only some basic examples to get you started with this technology using Node.js. Ultimately, our goal is to encourage you to explore new modern patterns in order to run and scale Node.js applications.

What is a container?

A **container**, specifically a **Linux container**, as standardized by the **Open Container Initiative (OCI)** (nodejsdp.link/opencontainers), is defined as "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 other words, by using containers, you can seamlessly package and run applications on different machines, from a local development laptop on your desk to a production server in the cloud.

Other than being extremely portable, applications running as containers have the advantage of having very little overhead when executed. In fact, containers run almost as fast as running the native application directly on the operating system.

In simple terms, you can see a container as a standard unit of software that allows you to define and run an *isolated* process directly on a Linux operating system.

For their portability and performance, containers are considered a huge step forward when compared to **virtual machines**.

There are different ways and tools to create and run an OCI compliant container for an application. The most popular of them is **Docker** (nodejsdp.link/docker).

You can install Docker in your system by following the instructions for your operating system on the official documentation: nodejsdp.link/docker-docs.

Creating and running a container with Docker

Let's rewrite our simple web server application with some minor changes (`app.js`):

```
import { createServer } from 'http'
import { hostname } from 'os'

const version = 1
const server = createServer((req, res) => {
  let i = 1e7; while (i > 0) { i-- }
  res.end(`Hello from ${hostname()} (v${version})`)
})
server.listen(8080)
```

Compared to the previous versions of this web server, here, we send the machine hostname and the application version back to the user. If you run this server and make a request, you should get back something like this:

```
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"
  }
}
```

In order to *dockerize* our application, we need to follow a two-step process:

- Build a container image
- Run a container instance from the image

To create the **container image** for our application, we have to 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 (actually 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:14-alpine
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:14-alpine` 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 an image that already contains version 14 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 the 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 are able to 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:

```
Sending build context to Docker daemon 7.168kB
Step 1/5 : FROM node:14-alpine
----> ea308280893e
Step 2/5 : EXPOSE 8080
----> Running in 61c34f4064ab
Removing intermediate container 61c34f4064ab
----> 6abfcd0e750
Step 3/5 : COPY app.js package.json /app/
----> 9d498d7dbf8b
Step 4/5 : WORKDIR /app
----> Running in 70ea26158cbe
Removing intermediate container 70ea26158cbe
----> fc075a421b91
Step 5/5 : CMD ["npm", "start"]
----> Running in 3642a01224e8
Removing intermediate container 3642a01224e8
----> bb3bd34bac55
Successfully built bb3bd34bac55
```



Note that if you have never used the `node:14-alpine` 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 is the ID of our container image. We can use it to run an instance of the container with the following command:

```
docker run -it -p 8080:8080 bb3bd34bac55
```

This command is essentially telling Docker to run the application from image `bb3bd34bac55` 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 `localhost:8080`. So, if we use `curl` to send a request to the web server, we should get a response similar to the following:

```
Hello from f2ffa85c8ff8 (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, hooray! Even though this seems like a particularly exciting achievement, we have just scratched the surface here. The real power of containers comes out when building more complicated applications. For instance, when building applications composed by multiple independent services that needs to be deployed and coordinated across multiple cloud servers. In this situation, Docker alone is not sufficient anymore. We need a more complex system that allows us to orchestrate all the running container instances over the available machines in our cloud cluster: we need a container orchestration tool.

A container orchestration tool has a number of responsibilities:

- It allows us to join multiple cloud servers (nodes) into one logical cluster, where nodes can be added and removed dynamically without affecting the availability of the services running in every node.
- It makes sure that there is no downtime. If a container instance stops or becomes unresponsive to health checks, it will be automatically restarted. Also, if a node in the cluster fails, the workload running in that node will be automatically migrated to another node.
- Provides functionalities to implement service discovery and load balancing.
- Provides orchestrated access to durable storage so that data can be persisted as needed.
- Automatic rollouts and rollbacks of applications with zero downtime.
- Secret storage for sensitive data and configuration management systems.

One of the most popular container orchestration systems is Kubernetes ([nodejsdp.link/kubernetes](https://kubernetes.io)), originally open sourced by Google in 2014. The name Kubernetes originates from the Greek "κυβερνήτης", meaning "helmsman" or "pilot", but also "governor" or more generically, "the one in command". Kubernetes incorporates years of experience from Google engineers running workloads in the cloud at scale.

One of its peculiarities is the declarative configuration system that allows you to define an "end state" and let the orchestrator figure out the sequence of steps necessary to reach the desired state, without disrupting the stability of the services running on the cluster.

The whole idea of Kubernetes configuration revolves around the concept of "objects". An object is an element in your cloud deployment, which can be added, removed, and have its configuration changed over time. Some good examples of Kubernetes objects are:

- Containerized applications
- Resources for the containers (CPU and memory allocations, persistent storage, access to devices such as network interfaces or GPU, and so on)
- Policies for the application behavior (restart policies, upgrades, fault-tolerance)

A Kubernetes object is a sort of "record of intent", which means that once you create one in a cluster, Kubernetes will constantly monitor (and change, if needed) the state of the object to make sure it stays compliant with the defined expectation.

A Kubernetes cluster is generally managed through a command-line tool called `kubectl` (nodejsdp.link/kubectl-install).

There are several ways to create a Kubernetes cluster for development, testing, and production purposes. The easiest way to start experimenting with Kubernetes is through a local single-node cluster, which can be easily created by a tool called `minikube` (nodejsdp.link/minikube-install).

Make sure to install both `kubectl` and `minikube` on your system, as we will be deploying our sample containerized app on a local Kubernetes cluster in the next section!



Another great way to learn about Kubernetes is by using the official interactive tutorials (nodejsdp.link/kubernetes-tutorials).

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 Cloud Container Registry (nodejsdp.link/gc-container-registry), or Amazon Elastic Container Registry (nodejsdp.link/ecr). 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, in order 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 have to run together in the same Kubernetes node. Containers in the same pod have shared resources like 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:

```
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 actually 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:

```
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, in order to upgrade all the running pods to this new version, we have to run the following command:

```
kubectl set image deployment/hello-web hello-web=hello-web:v2 --record
```

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)
```

Note the **v2** flag there.

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 completes our mini Kubernetes tutorial. The lesson here is that, when using a container orchestrator platform like Kubernetes, we can keep our application code quite simple, as we won't have to include concerns such as scaling to multiple instances or deal with soft rollouts and application restarts. This is the major advantage of this approach.

Of course, this simplicity does not come for free. It is paid by having to learn and manage the orchestration platform. If you are running small applications in production, it is probably not worth to incur the complexity and the cost of having to install and manage a container orchestrator platform like Kubernetes. However, if you are serving millions of users every day, there is definitely a lot of value in building and maintaining such a powerful infrastructure.

Another interesting observation is that, when running containers in Kubernetes, containers are often considered "disposable," which basically means that they could be killed and restarted at any time. While this might seem like a non-relevant detail, you should actually take this behavior into account and try to keep your applications as stateless as possible. In fact, containers, by default, won't retain any change in the local filesystem, so every time you have to store some persistent information, you will have to rely on external storage mechanisms such as 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 have mainly focused our analysis on the X-axis of the scale cube. We saw how it represents the easiest and most immediate way to distribute the load and scale an application, also improving its availability. In the following section, we are going to focus on the Y-axis of the scale cube, where applications are scaled by **decomposing** them by functionality and service. As we will learn, this technique allows us to scale not only the capacity of an application, but also, and most importantly, its complexity.

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 with 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 30 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:

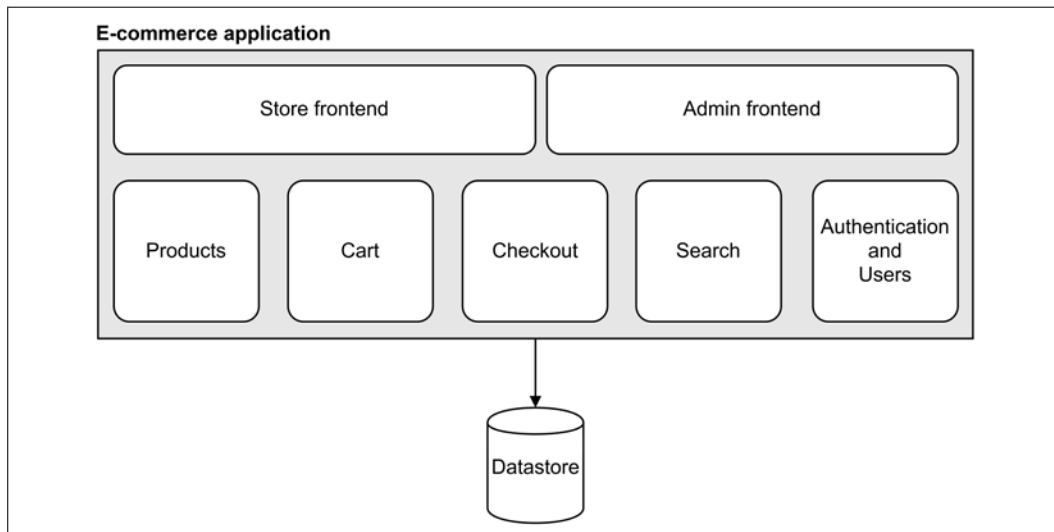


Figure 12.9: Example of a monolithic architecture

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 problem with this type of architecture is the interconnection between its modules; the fact that they all live inside the same application makes it very easy for a developer to build interactions and coupling between modules. For example, consider the use case of when a product is being purchased: the **Checkout** module has to update the availability of a **Product** object, and if those two modules are in the same application, it's too easy for a developer to just obtain a reference to a **Product** object and update its availability directly. Maintaining a low coupling between internal modules is very hard in a monolithic application, partly because the boundaries between them are not always clear or properly enforced.

A **high coupling** is often one of the main obstacles to the growth of an application and prevents its scalability in terms of complexity. In fact, an intricate dependency graph means that every part of the system is a liability, it has to be maintained for the entire life of the product, and any change should be carefully evaluated because every component is like a wooden block in a Jenga tower: moving or removing one of them can cause the entire tower to collapse. This often results in building conventions and development processes to cope with the increasing complexity of the project.

The microservice architecture

Now, we are going to reveal the most important pattern in Node.js for writing big applications: avoid writing big applications. This seems like a trivial statement, but it's an incredibly effective strategy to scale both the complexity and the capacity of a software system. So, what's the alternative to writing big applications? The answer is in the Y-axis of the scale cube: decomposition and splitting by service and functionality. The idea is to break down an application into its essential components, creating separate, independent applications. It is practically the opposite of a monolithic architecture. This fits perfectly with the Unix philosophy and the Node.js principles we discussed at the beginning of the book; in particular, the motto "make each program do one thing well."

Microservice architecture is, today, the main reference pattern for this type of approach, where a set of self-sufficient services replace big monolithic applications. The prefix "micro" means that the services should be as small as possible, but always within reasonable limits. Don't be misled by thinking that creating an architecture with a hundred different applications exposing only one web service is necessarily a good choice. In reality, there is no strict rule on how small or big a service should be. It's not the size that matters in the design of a microservice architecture; instead, it's a combination of different factors, mainly **loose coupling**, **high cohesion**, and **integration complexity**.

An example of a microservice architecture

Let's now see what the monolithic e-commerce application would look like using a microservice architecture:

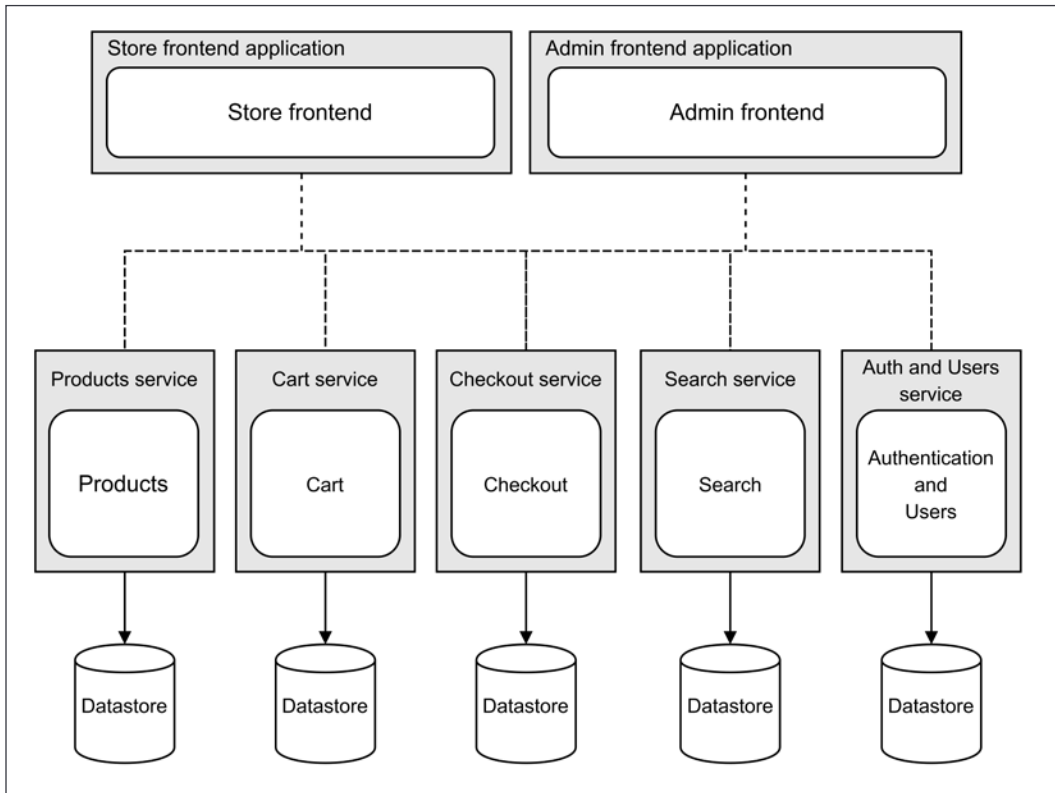


Figure 12.10: An example implementation of an e-commerce system using the Microservice pattern

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 has to 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 tells us that, in some way, they have to 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 generic. 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 nodejsdp.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** (nodejssdp.link/elasticsearch) is a great example of a reusable search service. **ORY** (nodejssdp.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 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 up 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 also, 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 has to 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 has to 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:

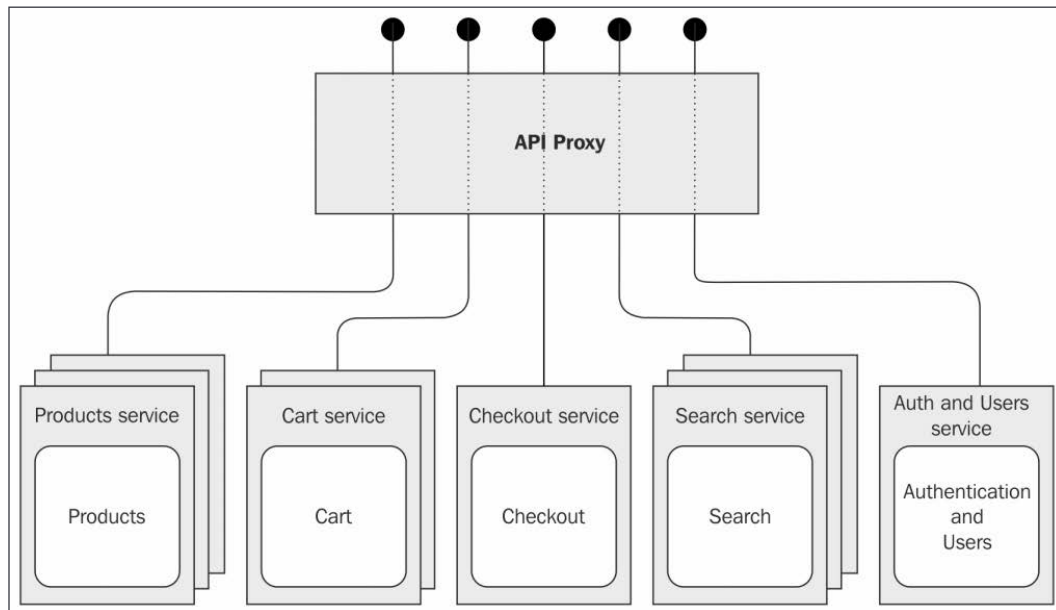


Figure 12.11: Using the API Proxy pattern in an e-commerce application

From the preceding diagram, it should be clear how an API proxy can hide the complexity of its underlying infrastructure. This is really 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*:

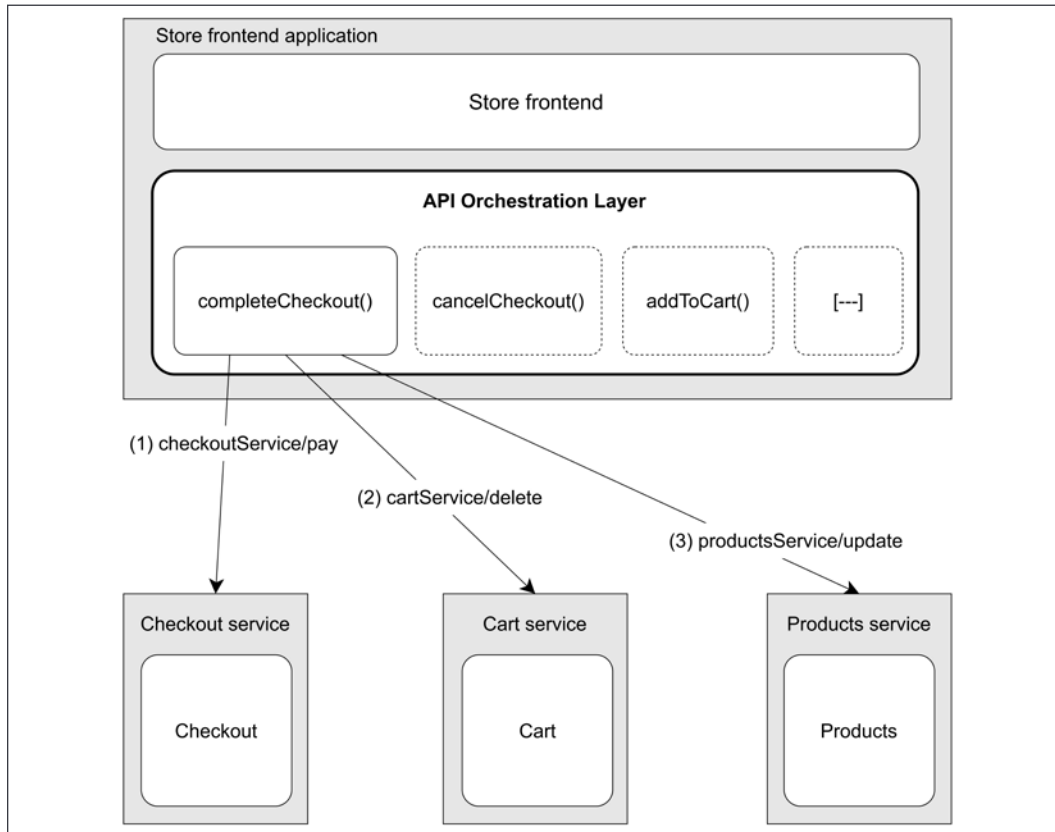


Figure 12.12: An example usage of an orchestration layer to interact with multiple microservices

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 is infinite, but the important pattern to remember is the role of the orchestration layer, which acts as an abstraction between a number of services and a specific application.

The orchestration layer is a great candidate for a 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:

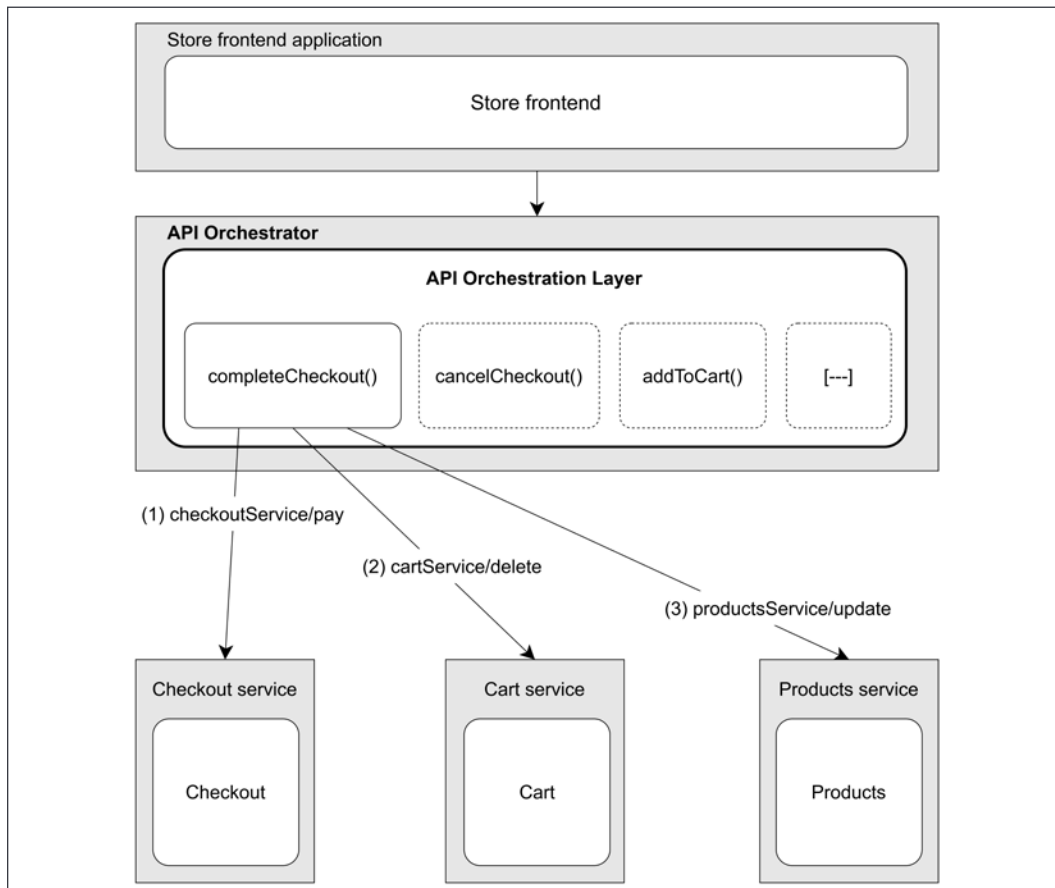


Figure 12.13: An application of the API Orchestrator pattern for our e-commerce example

Creating a standalone orchestrator, as shown in the previous figure, can help in decoupling the client application (in our case, the **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.

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 has to have a 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 **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:

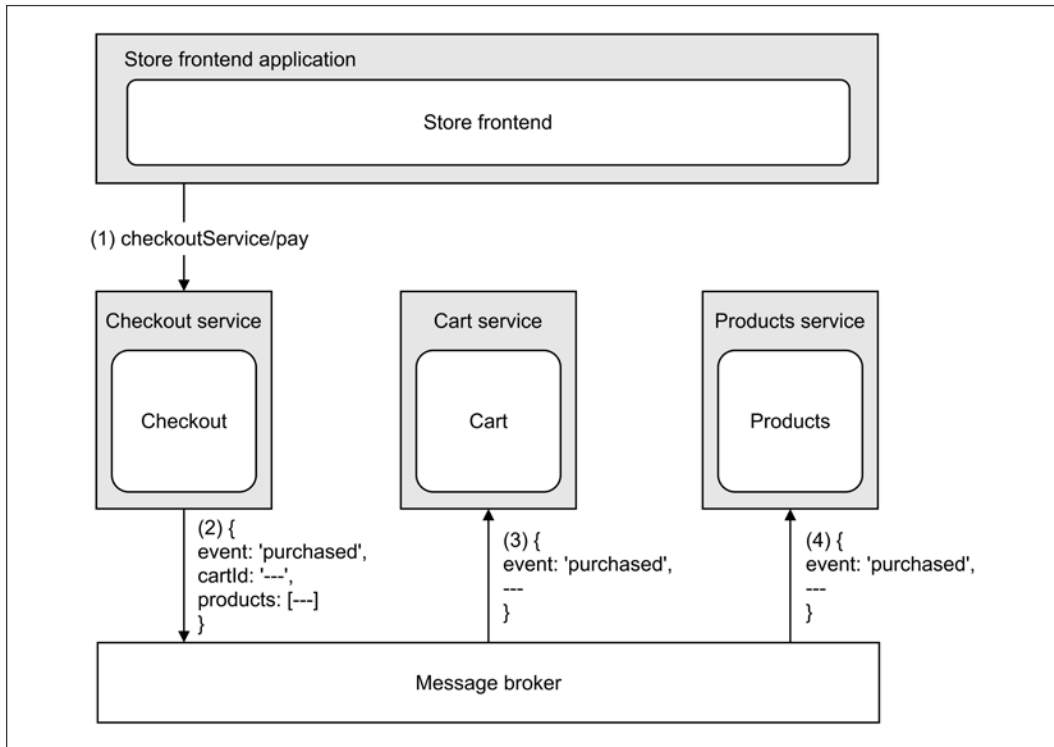


Figure 12.14: Using a message broker to distribute events in our e-commerce application

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 has to 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. The **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 into 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 has to know how to move the gears of the entire system, since each service is in charge of 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 learned how to design Node.js architectures that scale both in capacity and complexity. We saw how scaling an application is not only about handling more traffic or reducing the response time, but it's also a practice to apply whenever we want better availability and tolerance to failures. We saw how these properties often are on the same wavelength, and we understood that scaling early is not a bad practice, especially in Node.js, which allows us to do it easily and with few resources.

The scale cube taught us that applications can be scaled across three dimensions. Throughout this chapter, we focused on the two most important dimensions, the X-and Y-axes, allowing us to discover two essential architectural patterns, namely, load balancing and microservices. You should now know how to start multiple instances of the same Node.js application, how to distribute traffic across them, and how to exploit this setup for other purposes, such as fail tolerance and zero-downtime restarts. We also analyzed how to handle the problem of dynamic and auto-scaled infrastructures. With this, we saw that a service registry can really come in useful for those situations. We learned how to achieve these goals by using plain Node.js, external load balancers like Nginx, and service discovery systems like Consul. We also learned the basics of Kubernetes.

At this point, we should have got to grips with some very practical approaches to be able to face scalability much more fearlessly than before.

However, cloning and load balancing cover only one dimension of the scale cube, so we moved our analysis to another dimension, studying in more detail what it means to split an application by its constituent services by building a microservice architecture. We saw how microservices enable a complete revolution in how a project is developed and managed, providing a natural way to distribute the load of an application and split its complexity. However, we learned that this also means shifting the complexity from *how to build a big monolithic application* to *how to integrate a set of services*. This last aspect is where we focused the last part of our analysis, showing some of the architectural solutions to integrate a set of independent services.

In the next and last chapter of this book, we will have the chance to complete our *Node.js Design Patterns* journey by analyzing the messaging patterns we discussed in this chapter, in addition to more advanced integration techniques that are useful when implementing complex distributed architectures.

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 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 like `faker` (nodejsdp.link/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 names 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 have to 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 actually implement a minimal version of this idea with Node.js! If this turns out to be the next big startup idea and makes you a millionaire, well... don't forget to thank the authors of this book. :)

13

Messaging and Integration Patterns

If scalability is about distributing systems, integration is about connecting them. In the previous chapter, we learned how to distribute an application, fragmenting it across several processes and machines. For this to work properly, all those pieces have to communicate in some way and, hence, they have to be integrated.

There are two main techniques to integrate a distributed application: one is to use shared storage as a central coordinator and keeper of all the information, the other one is to use messages to disseminate data, events, and commands across the nodes of the system. This last option is what really makes the difference when scaling distributed systems, and it's also what makes this topic so fascinating and sometimes complex.

Messages are used in every layer of a software system. We exchange messages to communicate on the Internet; we can use messages to send information to other processes using pipes; we can use messages within an application as an alternative to direct function invocation (the Command pattern), and also device drivers use messages to communicate with the hardware. Any discrete and structured data that is used as a way to exchange information between components and systems can be seen as a *message*. However, when dealing with distributed architectures, the term **messaging system** is used to describe a specific class of solutions, patterns, and architectures that are meant to facilitate the exchange of information over the network.

As we will see, several traits characterize these types of systems. We might choose to use a broker versus a peer-to-peer structure, we might use a request/reply message exchange or one-way type of communication, or we might use queues to deliver our messages more reliably; the scope of the topic is really broad. The book *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf gives us an idea about the vastness of the topic. Historically, it is considered the *Bible* of messaging and integration patterns and has more than 700 pages describing 65 different integration patterns. In this final chapter, we will explore the most important of those well-known patterns – plus some more modern alternatives – considering them from the perspective of Node.js and its ecosystem.

To sum up, in this chapter, we will learn about the following topics:

- The fundamentals of a messaging system
- The Publish/Subscribe pattern
- Task distribution patterns and pipelines
- Request/reply patterns

Let's begin with the fundamentals.

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 in 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:

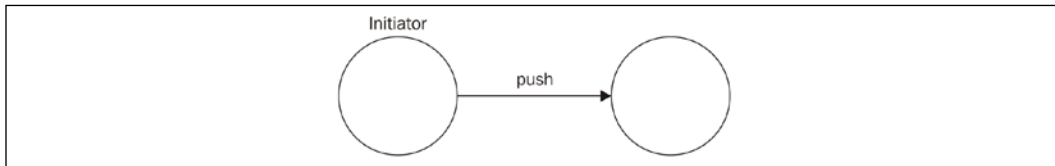


Figure 13.1: One-way communication

A typical example of one-way communication is an email or a web server that sends a message to a connected browser using WebSockets, 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:

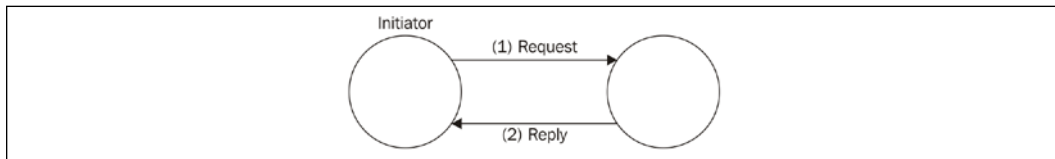


Figure 13.2: Request/Reply message exchange pattern

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. Take a look at the example represented in the next diagram:

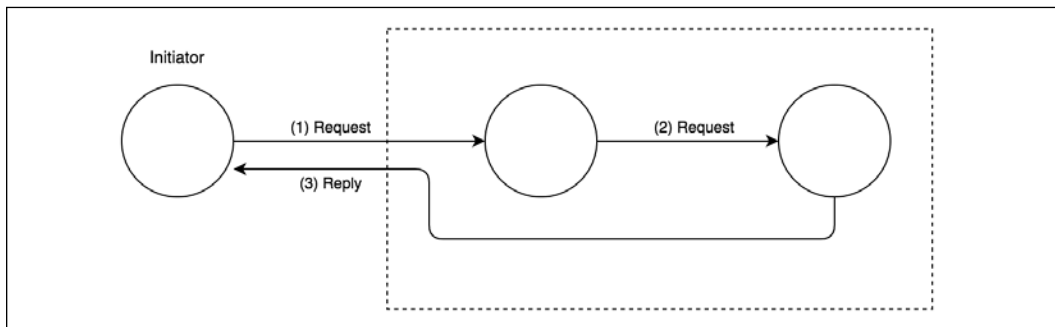


Figure 13.3: Multi-node request/reply communication

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 in *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 has to 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, 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 are using an Event Message when, for example, we leverage WebSockets to send notifications from the server to the client to communicate changes to some data or mutations in the state of the system.

Events are a very important integration mechanism in distributed applications, as they enable us to keep all the nodes of the system on the same page.

Document Messages

The **Document Message** is primarily meant to transfer data between components and machines. A typical example is a message used to transfer the results of a database query.

The main characteristic that differentiates a Document Message from a Command Message (which might also contain data) is that the message does not contain any information that tells the receiver what to do with the data. 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 with 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 semantic of the communication channel used to move our messages around.

Asynchronous messaging, queues, and streams

At this point in the book, you should already be familiar with the characteristics of an asynchronous operation. Well, it turns out that the same principles can be applied to messaging and communications.

We can compare synchronous communications to a phone call: the two peers must be connected to the same channel at the same time and they should exchange messages in real time. Normally, if we want to call someone else, we either need another phone or terminate the ongoing communication to start a new one.

Asynchronous communication is similar to an SMS: it doesn't require the recipient to be connected to the network the moment we send it; we might receive a response immediately or after an unknown delay, or we might not receive a response at all. We might send multiple SMSes to multiple recipients one after the other and receive their responses (if any) in any order. In short, we have better parallelism with the use of fewer resources.

Another important characteristic of asynchronous communications is that the messages can be stored and then delivered as soon as possible or at a later time. This can be useful when the receiver is too busy to handle new messages or when we want to guarantee delivery. In messaging systems, this is made possible using a **message queue**, a component that mediates the communication between the producer of the messages and the consumer, storing any message before it gets delivered to its destination, as shown in the following diagram:

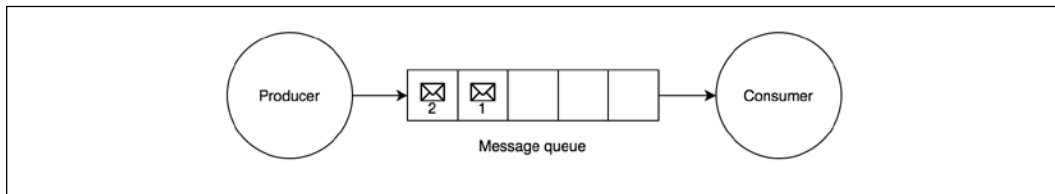


Figure 13.4: A message queue

If for any reason the consumer crashes, disconnects from the network, or experiences a slowdown, the messages are accumulated in the queue and dispatched as soon as the consumer comes back online. The queue can be located in the producer, or be split between the producer and the consumer (in peer-to-peer architectures), or live in a dedicated external system acting as middleware for the communication (**broker**).

Another data structure that has a similar (but not the same!) goal as a message queue is the **log**. A log is an append-only data structure, which is durable and whose messages can be read as they arrive or by accessing its history. In the context of messaging and integration systems, this is also known as a data **stream**.

Compared to a queue, in a stream, messages are not removed when they are retrieved or processed. This way, consumers can retrieve the messages as they arrive or can query the stream at any time to retrieve past messages. This means that a stream provides more freedom when it comes to accessing the messages, while queues usually expose only one message at a time to their consumers. Most importantly, a stream can be shared by more than one consumer, which can access the messages (even the same messages) using different approaches.

Figure 13.5 gives you an idea of the structure of a stream compared to that of a message queue:

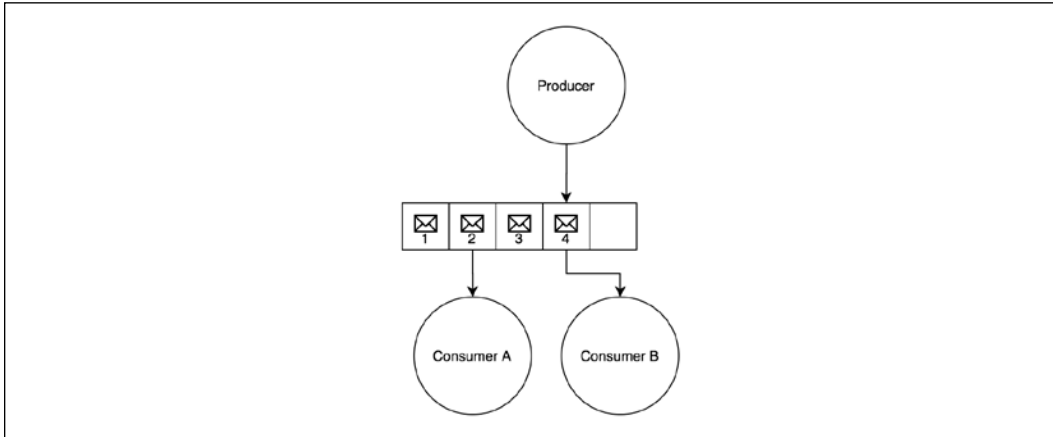


Figure 13.5: A stream

You will be able to better appreciate the difference between a queue and a stream later in the chapter when we implement a sample application using both approaches.

The final fundamental element to consider in a messaging system is the way the nodes of the system are connected together, which can be directly or through an intermediary.

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:

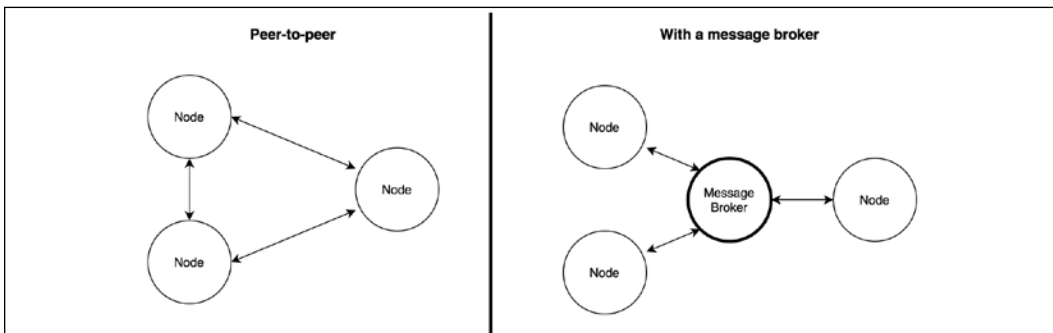


Figure 13.6: Peer-to-peer communication versus message brokering

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 have to 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 supporting different messaging protocols to interact.



MQTT (nodejsdp.link/mqtt) is a lightweight messaging protocol, specifically designed for machine-to-machine communications (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 has to 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 particular technology, protocol, or architecture.

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:

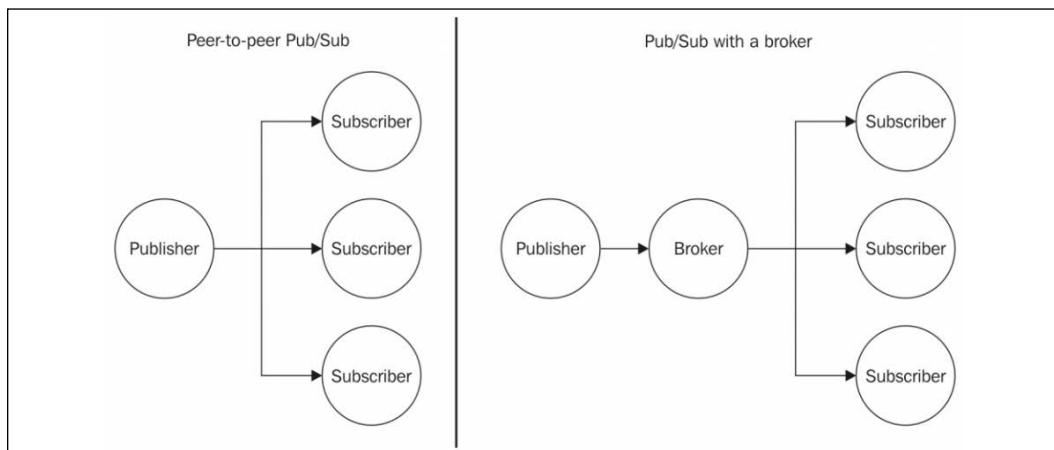


Figure 13.7: Publish/Subscribe messaging pattern

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 has to 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 a real-life example of how the Pub/Sub pattern can help us integrate a distributed architecture, we are now going to build a very basic real-time chat application using pure WebSockets. Then, we will scale it by running multiple instances, and finally, using a messaging system, we will build a communication channel between all the server instances.

Implementing the server side

Now, let's take 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 'http'
import staticHandler from 'serve-handler'
import ws from 'ws'

// serve static files
const server = createServer((req, res) => { // (1)
  return staticHandler(req, res, { public: 'www' })
})

const wss = new ws.Server({ 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 === ws.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 `www` 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 the magic of Node.js! 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 `www/index.html` as follows:

```

<!DOCTYPE html>
<html>
  <body>
    Messages:
    <div id="messages"></div>
    <form id="msgForm">
      <input type="text" placeholder="Send a message" id="msgBox"/>
      <input type="submit" value="Send"/>
    </form>
  </body>
</html>

```



```
</form>
<script>
  const ws = new WebSocket(
    `ws://${window.document.location.host}`
  )
  ws.onmessage = function (message) {
    const msgDiv = document.createElement('div')
    msgDiv.innerHTML = message.data
    document.getElementById('messages').appendChild(msgDiv)
  }
  const form = document.getElementById('msgForm')
  form.addEventListener('submit', (event) => {
    event.preventDefault()
    const message = document.getElementById('msgBox').value
    ws.send(message)
    document.getElementById('msgBox').value = ''
  })
</script>
</body>
</html>
```

The HTML page we just created doesn't really need many comments, it's just a piece of straightforward web development. We use the native WebSocket object to initialize a connection to our Node.js server, and then start listening for messages from the server, displaying them in new div elements as they arrive. For sending messages, instead, we use a simple textbox and a button within a form.



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:

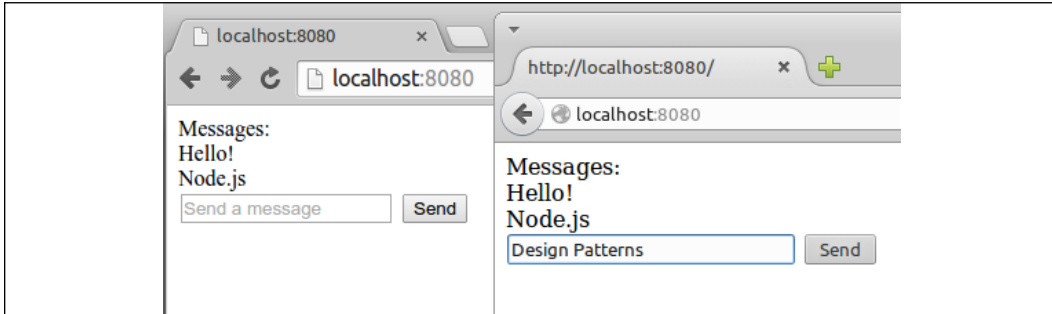


Figure 13.8: Our new chat application in action

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, should be able to 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 would 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 particular 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' 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 most simple and 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.

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 for 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:

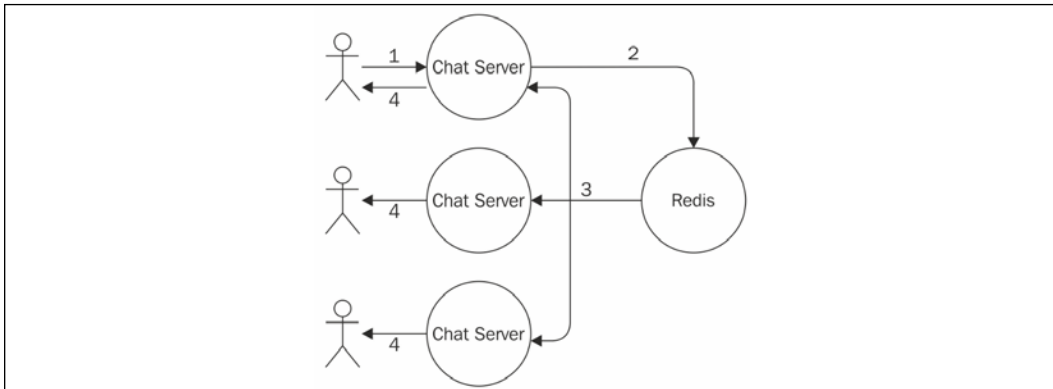


Figure 13.9: Using Redis as a message broker for our chat application

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 in practice how this works. Let's modify the server code by adding the publish/subscribe logic:

```
import { createServer } from 'http'
import staticHandler from 'serve-handler'
import ws from 'ws'
import Redis from 'ioredis' // (1)

const redisSub = new Redis()
const redisPub = new Redis()
```

```
// serve static files
const server = createServer((req, res) => {
  return staticHandler(req, res, { public: 'www' })
})

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) // (2)
  })
})

redisSub.subscribe('chat_messages') // (3)
redisSub.on('message', (channel, msg) => {
  for (const client of wss.clients) {
    if (client.readyState === ws.OPEN) {
      client.send(msg)
    }
  }
})

server.listen(process.argv[2] || 8080)
```

The changes that we made to our original chat server are highlighted in the preceding code. This 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, 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 also has to 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 a 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 a good candidate for a peer-to-peer architecture, one of the best solutions to evaluate is certainly **ZeroMQ** (nodejsdp.link/zeromq, also known as `zmq` or `ØMQ`). ZeroMQ is a networking library that provides the basic tools to build a large variety of messaging patterns. It is low-level, extremely fast, and has a minimalistic API, but it offers all the basic building blocks to create a solid messaging system, such as atomic messages, load balancing, queues, and many more. It supports many types of transport, such as in-process channels (`inproc://`), inter-process communication (`ipc://`), multicast using the PGM protocol (`pgm://` or `epgm://`), and, of course, the classic TCP (`tcp://`).

Among the features of ZeroMQ, we can also find tools to implement a Publish/Subscribe pattern, which is exactly what we need for our example. So, what we are going to do now is remove the broker (Redis) from the architecture of our chat application and let the various nodes communicate in a peer-to-peer fashion, leveraging the publish/subscribe sockets of ZeroMQ.



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 has to directly connect to the other available instances in order 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.

A subscription can have a *filter* that specifies what messages are delivered to the connected SUB sockets. The filter is a simple **binary buffer** (so it can also be a string), which will be matched against the beginning of the message (which is also a binary buffer). When a message is sent through the PUB socket it is broadcast to all the connected SUB sockets, but only after their subscription filters are applied. The filters will be applied to the publisher side only if a *connected* protocol is used, such as, for example, TCP.

The following diagram shows the pattern applied to our distributed chat server architecture (with only two instances, for simplicity):

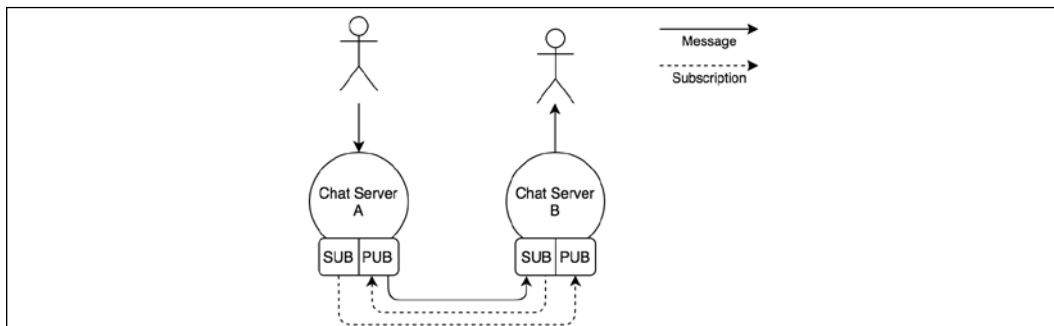


Figure 13.10: Chat server messaging architecture using ZeroMQ PUB/SUB sockets

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:

```
import { createServer } from 'http'
import staticHandler from 'serve-handler'
import ws from 'ws'
import yargs from 'yargs' // (1)
import zmq from 'zeromq'

// serve static files
const server = createServer((req, res) => {
  return staticHandler(req, res, { public: 'www' })
})

let pubSocket
async function initializeSockets () {
  pubSocket = new zmq.Publisher() // (2)
  await pubSocket.bind(`tcp://127.0.0.1:${yargs.argv.pub}`)

  const subSocket = new zmq.Subscriber() // (3)
  const subPorts = [].concat(yargs.argv.sub)
  for (const port of subPorts) {
    console.log(`Subscribing to ${port}`)
    subSocket.connect(`tcp://127.0.0.1:${port}`)
  }
  subSocket.subscribe('chat')

  for await (const [msg] of subSocket) { // (4)
    console.log(`Message from another server: ${msg}`)
    broadcast(msg.toString().split(' ')[1])
  }
}
```



```

initializeSockets()

const wss = new ws.Server({ server })
wss.on('connection', client => {
  console.log('Client connected')
  client.on('message', msg => {
    console.log(`Message: ${msg}`)
    broadcast(msg)
    pubSocket.send(`chat ${msg}`) // (5)
  })
})

function broadcast (msg) {
  for (const client of wss.clients) {
    if (client.readyState === ws.OPEN) {
      client.send(msg)
    }
  }
}

server.listen(yargs.argv.http || 8080)

```

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 import two new packages. First, we import `yargs` (nodejsdp.link/yargs), which is a command-line argument parser; we need this to easily accept named arguments. Secondly, we import the `zeromq` package (nodejsdp.link/zeromq), which is a Node.js client for ZeroMQ.
2. In the `initializeSockets()` function, we immediately 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` as a filter, which means that we will receive only the messages beginning with `chat`.

4. We start listening for messages arriving at our Subscriber socket using a `for await...of` loop, since `subSocket` is an async iterable. With each message we receive, we do some simple parsing to remove the chat prefix, and then 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. We use `chat` as a prefix followed by a space, so that the message will be published to all the subscriptions using `chat` as a filter.

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 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 is where the Publisher sockets of the other two instances should be listening at. 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 **message queue (MQ)**. With a message queue, 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 is opposed to 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 is able to 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 semantic** of a messaging system in three categories:

- **At most once:** Also known as *fire-and-forget*, the message is not persisted, and the delivery is not acknowledged. This means that the message can be lost in cases of crashes or disconnections of the receiver.
- **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 has to be persisted in the eventuality 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 an "exactly once" delivery semantic and to do that, the system has to use a message queue 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:

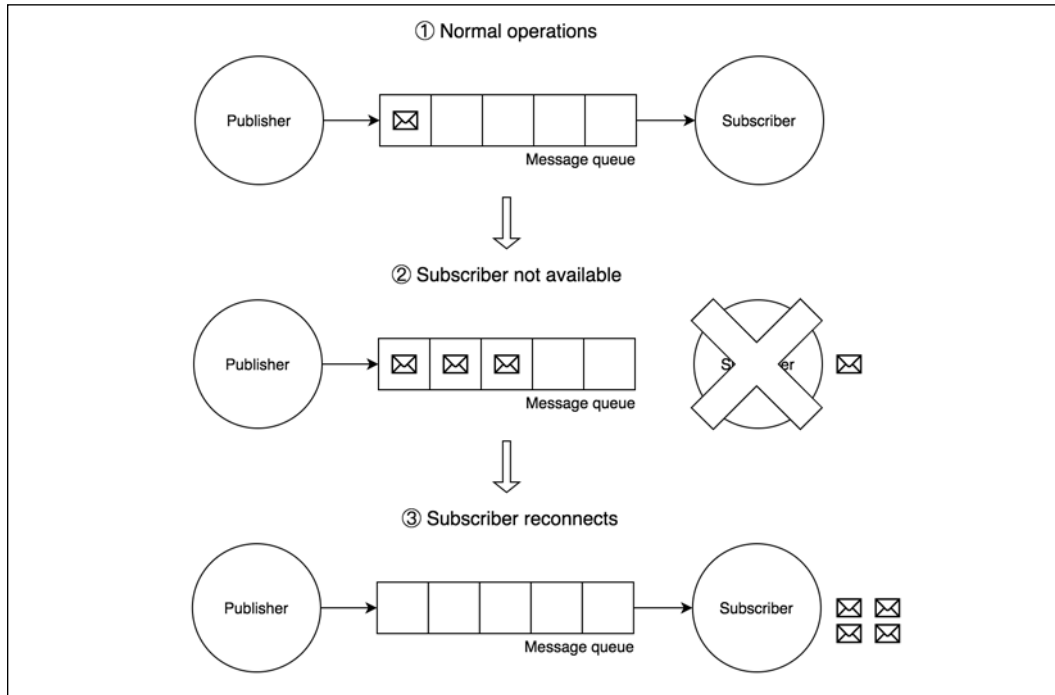


Figure 13.11: Example behavior of a messaging system backed by a queue

Figure 13.11 shows us 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 normally used in situations where messages must not be lost, which includes mission-critical applications such as banking systems, air traffic management and control systems, medical applications, and so on. This usually means that the typical enterprise-grade message queue is a very complex piece of software, which utilizes bulletproof protocols and persistent storage to guarantee the delivery of the message even in the presence of malfunctions. For this reason, enterprise messaging middleware has been, for many years, a prerogative of tech giants such as Oracle and IBM, each one of them usually implementing their own proprietary protocol, resulting in a strong customer lock-in. Fortunately, it's been a few years now since messaging systems entered the mainstream, thanks to the growth of open protocols such as AMQP, STOMP, and MQTT. Throughout the rest of the chapter we are going to use AMQP as the messaging protocol for our queuing system, so let's give it a proper introduction.

AMQP is an open standard protocol supported by many message-queuing systems. Besides defining a common communication protocol, it also provides a model to describe routing, filtering, queuing, reliability, and security.

The following diagram shows us all the AMQP components at a glance:

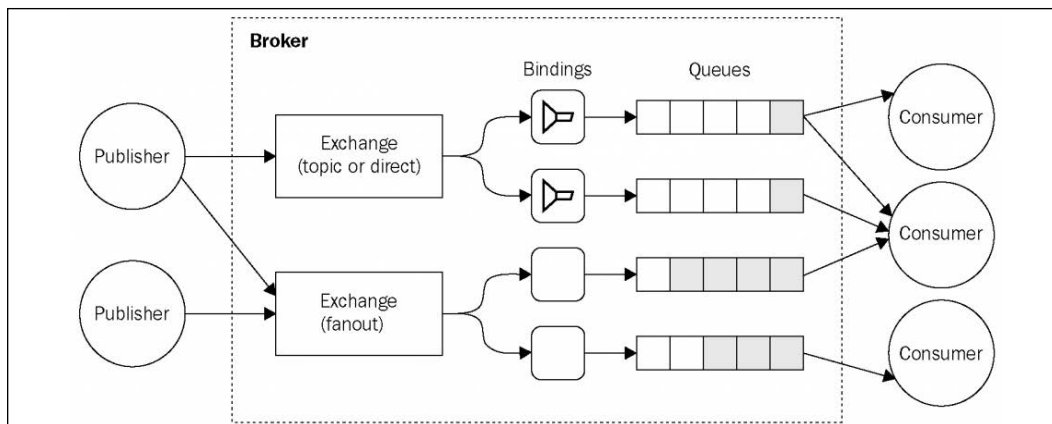


Figure 13.12: Example of an AMQP-based messaging system

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 imply 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 particular 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.`)
 - **Fanout 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 – which 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 nodejsdp.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 to not lose any message 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:

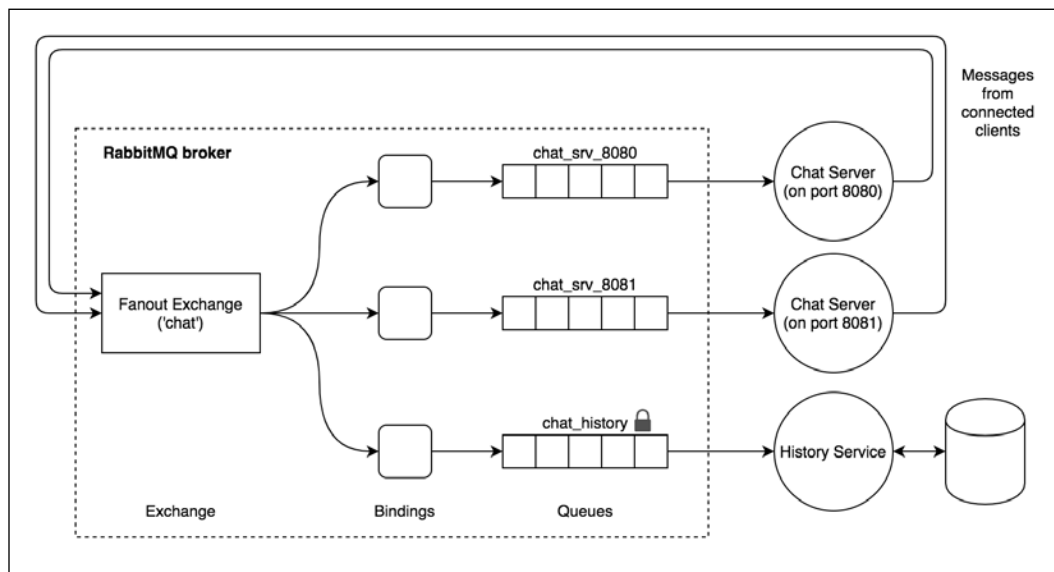


Figure 13.13: Architecture of our chat application with AMQP and history service

As shown in *Figure 13.13*, we are going to use a single fanout 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 has to 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 LevelUP 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.

Implementing a history service using AMQP

Let's now implement our history service! We are going to create a standalone application (a typical microservice), 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.

Let's see what this looks like in the code that follows:

```
import { createServer } from 'http'
import level from 'level'
import timestamp from 'monotonic-timestamp'
import JSONStream from 'JSONStream'
import amqp from 'amqplib'

async function main () {
  const db = level('./msgHistory')

  const connection = await amqp.connect('amqp://localhost') // (1)
  const channel = await connection.createChannel()
  await channel.assertExchange('chat', 'fanout') // (2)
  const { queue } = channel.assertQueue('chat_history') // (3)
  await channel.bindQueue(queue, 'chat') // (4)
```



```

channel.consume(queue, async msg => {                                     // (5)
  const content = msg.content.toString()
  console.log(`Saving message: ${content}`)
  await db.put(timestamp(), content)
  channel.ack(msg)
})

createServer((req, res) => {
  res.writeHead(200)
  db.createValueStream()
    .pipe(JSONStream.stringify())
    .pipe(res)
}).listen(8090)
}

main().catch(err => console.error(err))

```

We can immediately see that AMQP requires a little bit of setting up, which is necessary to create and connect all the components of the model. Let's see in detail how it works:

1. We first establish a connection with the AMQP broker, which in our case is RabbitMQ. Then, we create a channel, which is similar to a session that will maintain the state of our communications.
2. Next, we set up an exchange, named chat. As we already mentioned, it is a fanout exchange. The `assertExchange()` command will make sure that the exchange exists on the broker, otherwise it will create it.
3. 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.
4. Next, we bind the queue to the exchange we previously created. Here, we don't need any other particular option (such as a routing key or pattern), as the exchange is of the type fanout, so it doesn't perform any filtering.
5. Finally, we can begin to listen for messages coming from the queue we just created. We save every message that we receive in a LevelDB database using a monotonic timestamp as the key (see nodejsdp.link/monotonic-timestamp) to keep the messages sorted by date. It's also interesting to see 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.



If we are not interested in sending explicit acknowledgments, we can pass the `{ noAck: true }` option to the `channel.consume()` API.

Integrating the chat application with AMQP

To integrate the chat servers using AMQP, we have to 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 'http'
import staticHandler from 'serve-handler'
import ws from 'ws'
import amqp from 'amqplib'
import JSONStream from 'JSONStream'
import superagent from 'superagent'

const httpPort = process.argv[2] || 8080

async function main () {
  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(queue, msg => { // (2)
    msg = msg.content.toString()
    console.log(`From queue: ${msg}`)
    broadcast(msg)
  }, { noAck: true })

  // serve static files
  const server = createServer((req, res) => {
    return staticHandler(req, res, { public: 'www' })
  })

  const wss = new ws.Server({ server })
  wss.on('connection', client => {
    console.log('Client connected')
```

```

client.on('message', msg => {
  console.log(`Message: ${msg}`)
  channel.publish('chat', '', Buffer.from(msg)) // (3)
})

// query the history service
superagent // (4)
  .get('http://localhost:8090')
  .on('error', err => console.error(err))
  .pipe(JSONStream.parse('*'))
  .on('data', msg => client.send(msg))
})

function broadcast (msg) {
  for (const client of wss.clients) {
    if (client.readyState === ws.OPEN) {
      client.send(msg)
    }
  }
}

server.listen(httpPort)
}

main().catch(err => console.error(err))

```

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 fanout 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.

We can now run our new 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 in particular 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 message queues are **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 the context of system integration, a **stream** (or **log**) is an ordered, append-only, durable data structure. Messages – which in the context of streams would be more appropriately called **records** – are always added at the end of the stream and, unlike queues, they are not automatically deleted when they are consumed. Essentially, this characteristic makes a stream more similar to a data store than to a message broker. And like a data store, a stream can be queried to retrieve a batch of past records or replayed starting from a specific record.

Another important characteristic of streams is that records are pulled by the consumer from the stream. This intrinsically allows the consumer to process the records at its own pace without risking being overwhelmed.

Based on these features, a stream allows us to implement reliable message delivery out of the box, since no data is ever *lost* from the stream (even though data can still be removed explicitly or can be deleted after an optional retention period). In fact, as *Figure 13.14* shows, if a consumer crashes, all it has to do is start reading the stream from where it left off:

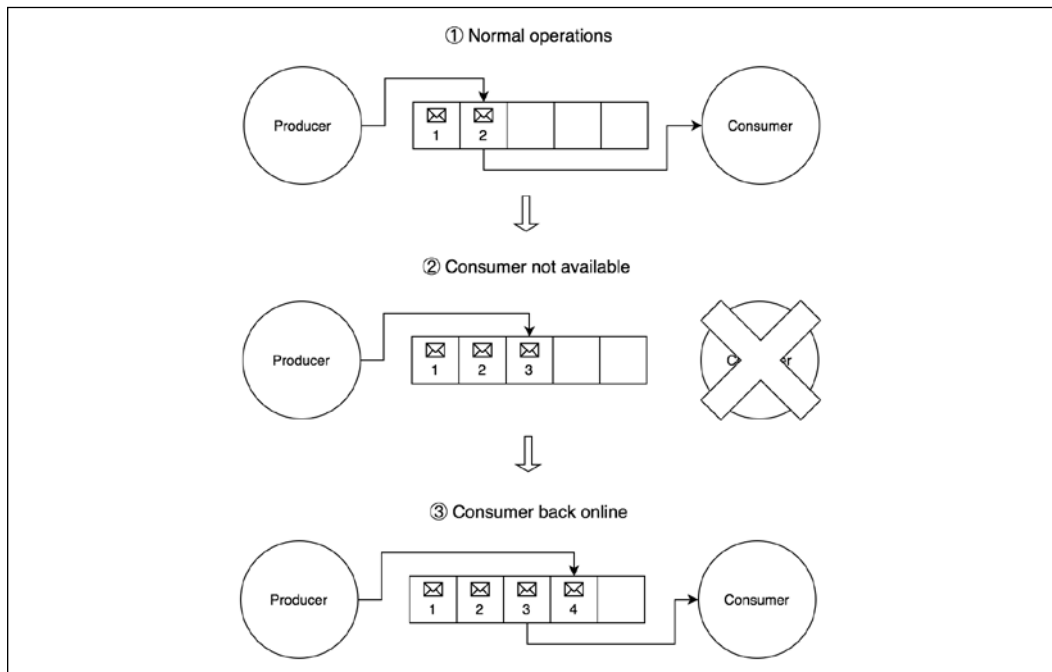


Figure 13.14: Reliable message delivery with streams

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. The main aspect of this mechanism is that it's very simple and barebone, but it's quite effective at making sure that no message is lost even when the consumer is not available.

Streams versus message queues

As we have seen so far, there are a lot of differences, but also a lot of similarities between a message queue and a stream. So, when should you use one in place of the other?

Well, the obvious use case for streams is when we have to process sequential data (streaming data) that may also require the consumer to process messages in batch or to look for correlations in past messages. Also, modern streaming platforms allow the ingestion of gigabytes of data per second and the distribution of both the data and the processing of the data across multiple nodes.

Both message queues and streams are well suited to implement simple Publish/Subscribe patterns, even with reliable message delivery. However, message queues are better suited for complex system integration tasks, since they provide advanced routing of messages and allow us to have different priorities for different messages (in streams, the order of the records is always preserved).

As we will see later, both can also be used to implement task distribution patterns, even though, in a standard architecture, message queues could be more suitable thanks to message priorities and more advanced routing mechanisms.

Implementing the chat application using Redis Streams

At the moment 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 a message queue 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 message queues, 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 'http'
import staticHandler from 'serve-handler'
import ws from 'ws'
import Redis from 'ioredis'

const redisClient = new Redis()
const redisClientXRead = new Redis()

// serve static files
const server = createServer((req, res) => {
  return staticHandler(req, res, { public: 'www' })
})

const wss = new ws.Server({ server })
wss.on('connection', async client => {
  console.log('Client connected')

  client.on('message', msg => {
    console.log(`Message: ${msg}`)
    redisClient.xadd('chat_stream', '*', 'message', msg) // (1)
  })

  // Load message history
  const logs = await redisClient.xrange( // (2)
    'chat_stream', '-', '+')
  for (const [, [, message]] of logs) {
    client.send(message)
  }
})

function broadcast (msg) {
  for (const client of wss.clients) {
    if (client.readyState === ws.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(message)
      lastRecordId = recordId
    }
  }
}

processStreamMessages().catch(err => console.error(err))

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 take a 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:
 - a. The name of the stream, which in our case is `chat_stream`.
 - b. 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 have to be monotonic to preserve the lexicographic order of the records and Redis takes care of that for us.
 - c. 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 into 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:
 - a. `BLOCK` means that we want the call to block until new messages arrive.
 - b. 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.
 - c. `STREAMS` is a keyword that tells Redis that we are now going to specify the details of the streams we want to read.
 - d. `chat_stream` is the name of the stream we want to read.
 - e. 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 the `propertyId`, we are simply keeping them out of the destructuring instruction. This particular destructuring, in combination with the `for...of` loop, is necessary to parse the data returned from the `xrange` command, 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 also a fairly complicated arguments list and return value that you can read more about at nodejsdp.link/xread.

Also, 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` commands (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 cannot be scaled beyond the boundaries of a single machine, so in this section, we are going to see how it's possible to use a similar pattern in a distributed architecture, using remote workers located anywhere in a network.

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:

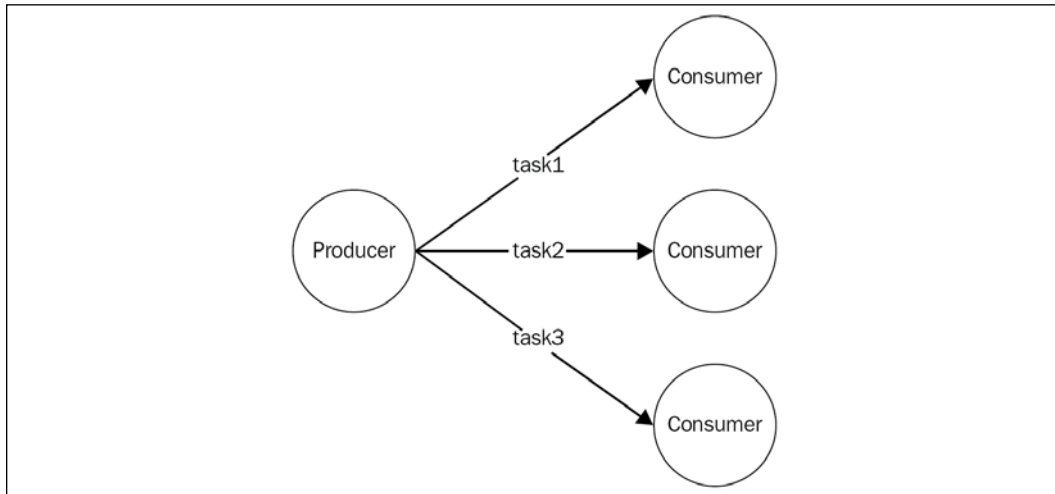


Figure 13.15: Distributing tasks to a set of consumers

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 similar to 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**, fanout distribution, or **ventilator**.

One important difference to the HTTP load balancers that 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 connects to the consumers, but the consumers themselves that connect to the task producer or to the task queue in order to receive new jobs. This is a great advantage in a scalable system as it allows us to seamlessly increase the number of workers without modifying the producer or adopting 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:

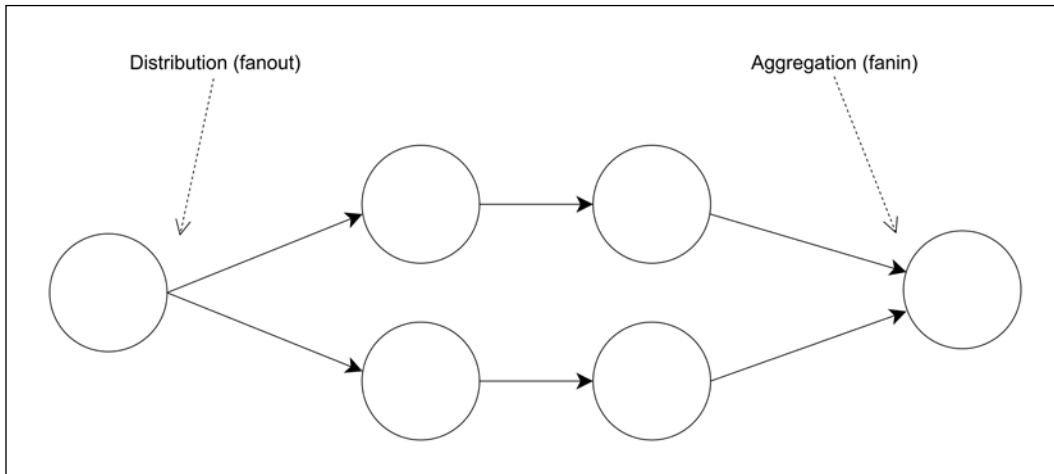


Figure 13.16: A messaging pipeline

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 (fanout), forwarded to other processing units, and then aggregated into a single node (fanin), 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 Fanout/Fanin 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 *connect* mode or *bind* mode. In other words, we can create a PUSH socket and bind it to a local port listening for the 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. The bind mode is the best solution for *durable* nodes, such as, for example, the task producer and the sink, while the 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 — 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 a 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 encrypt passwords as they are very 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:

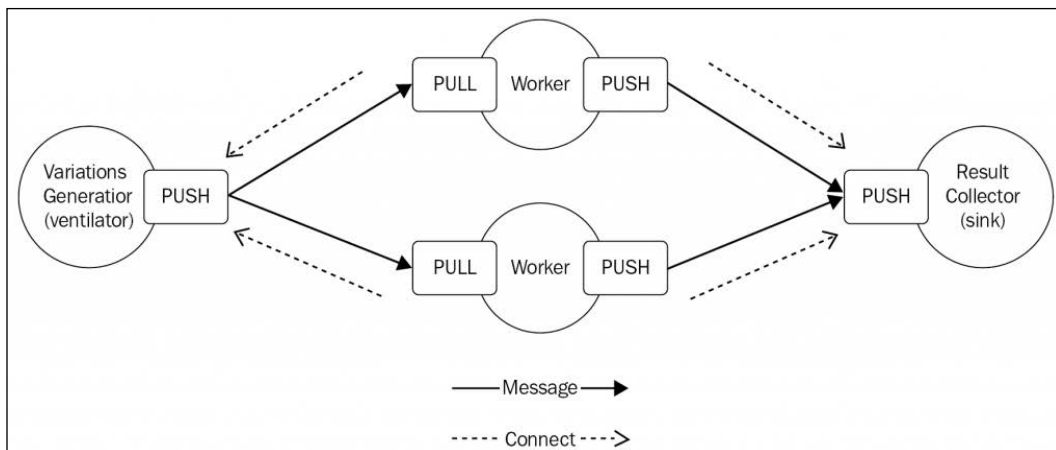


Figure 13.17: The architecture of a typical pipeline with ZeroMQ

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.

Implementing the producer

To represent intervals of variations, we are going to use indexed n-ary trees. If we imagine having a tree in which each node has exactly n children, where each child is one of the n elements of the given alphabet and we assign an index to each node in breadth-first order, then, given the alphabet $[a, b]$ we should obtain a tree such as the following:

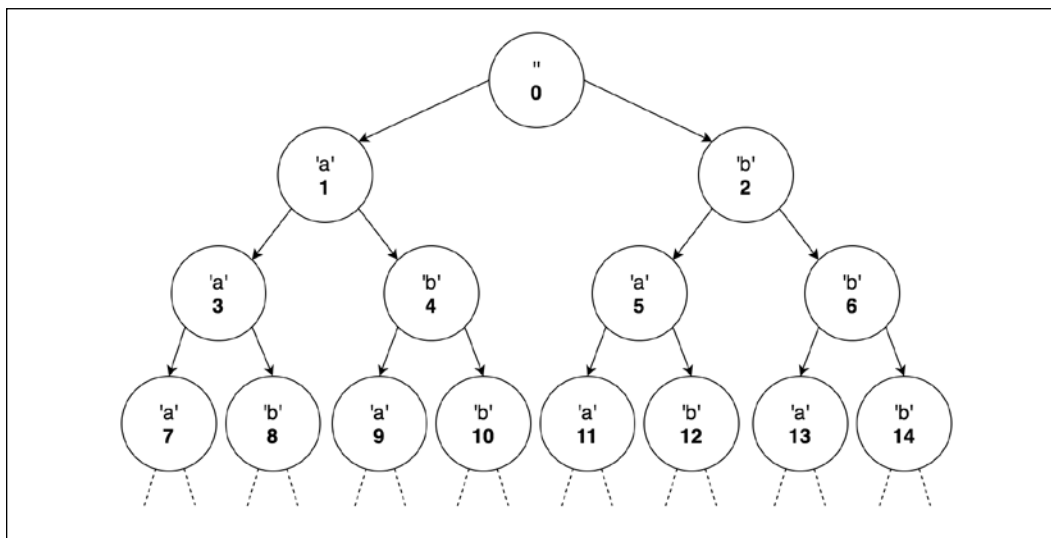


Figure 13.18: Indexed n-ary tree for alphabet $[a, b]$

It's then possible to obtain the variation corresponding to an index by traversing the tree from the root to the given index, appending the element of the nodes found along the way to the variation being calculated. For example, given the tree in *Figure 13.18*, the variation corresponding to the index 13 will be 'bba'.

We'll leverage the `indexed-string-variation` package (nodejsdp.link/indexed-string-variation) to aid us in calculating the corresponding variation given its index in the n-ary tree. This operation is done in the workers, so all we have to do in the ventilator is to produce intervals of indexes to give to the workers, which in turn will calculate all the variations of characters represented by those intervals.

Now, after the necessary theory, let's start to build our system by implementing the component responsible to generate the tasks to distribute (`generateTasks.js`):

```
export function * generateTasks (searchHash, alphabet,
  maxWordLength, batchSize) {
  let nVariations = 0
  for (let n = 1; n <= maxWordLength; n++) {
    nVariations += Math.pow(alphabet.length, n)
  }
  console.log('Finding the hashsum source string over ' +
    `${nVariations} possible variations`)

  let batchStart = 1
  while (batchStart <= nVariations) {
    const batchEnd = Math.min(
      batchStart + batchSize - 1, nVariations)
    yield {
      searchHash,
      alphabet: alphabet,
      batchStart,
      batchEnd
    }

    batchStart = batchEnd + 1
  }
}
```

The `generateTasks()` generator creates intervals of integers of `batchSize` size, starting from 1 (we exclude 0, which is the root of the tree, corresponding to the empty variation) and ending at the largest possible index (`nVariations`) for the given alphabet and the maximum word length provided (`maxLength`). Then, we pack all the data about the task into an object and yield it to the caller.



Please consider that to generate longer strings it may be necessary to switch to `BigInt` ([nodejsdp.link/bigint](https://nodejs.org/en/docs/next/api/bigint)) to represent their indexes, since the maximum safe integer manageable by JavaScript is currently $2^{53} - 1$, which is the value of `Number.MAX_SAFE_INTEGER`. Note that using very large integers may have a negative impact on the performances of the variations generator.

Now, we need to implement the logic of our producer, which is responsible for distributing the tasks across all workers (in the `producer.js` file):

```
import zmq from 'zermq'
import delay from 'delay'
import { generateTasks } from './generateTasks.js'

const ALPHABET = 'abcdefghijklmnopqrstuvwxyz'
const BATCH_SIZE = 10000

const [, , maxLength, searchHash] = process.argv

async function main () {
  const ventilator = new zmq.Push() // (1)
  await ventilator.bind('tcp://*:5016')
  await delay(1000) // wait for all the workers to connect

  const generatorObj = generateTasks(searchHash, ALPHABET,
    maxLength, BATCH_SIZE)
  for (const task of generatorObj) {
    await ventilator.send(JSON.stringify(task)) // (2)
  }
}

main().catch(err => console.error(err))
```

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. We then wait 1 second for all the workers to connect: we do this because if the producer starts while the workers are already running, the workers may connect at different times (because of their timer-based reconnection algorithm) and that may cause the first connecting worker to receive most of the tasks.
2. For each generated task, we stringify it and 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.

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'
import { createHash } from 'crypto'

export function processTask (task) {
  const variationGen = isv.generator(task.alphabet)
  console.log('Processing from ' +
    `${variationGen(task.batchStart)} (${task.batchStart})` +
    `to ${variationGen(task.batchEnd)} (${task.batchEnd})`)

  for (let idx = task.batchStart; idx <= task.batchEnd; idx++) {
    const word = variationGen(idx)
    const shasum = createHash('sha1')
    shasum.update(word)
    const digest = shasum.digest('hex')

    if (digest === task.searchHash) {
      return word
    }
  }
}
```

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 (`word`). Next, it calculates the SHA1 checksum for the word and it tries to match it against the `searchHash` passed within the task object. If the two digests match, then it returns the source word to the caller.

Now we are ready to implement the main logic of our worker (`worker.js`):

```
import zmq from 'zeromq'
import { processTask } from './processTask.js'

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

main().catch(err => console.error(err))

```

As we said, our worker represents a transient node in our architecture, therefore, its sockets should connect to a remote node instead of listening for the 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, and if a match is found, we send 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'

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())
  }
}

main().catch(err => console.error(err))

```

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 4 lowercase letters (because our alphabet comprises only lowercase letters) and we also provide a sample SHA1 checksum that corresponds to a secret 4-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.

Another known limitation of this implementation is the fact that the workers won't stop processing tasks if a match is found. This feature was intentionally left out to make the examples as focused as possible on the pattern being discussed. You can try adding this "stopping" mechanism as an exercise.

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 like AMQP, we have to 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 are able to send a set of messages directly to a single queue, we are already half-way 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 fanout 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 to 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:

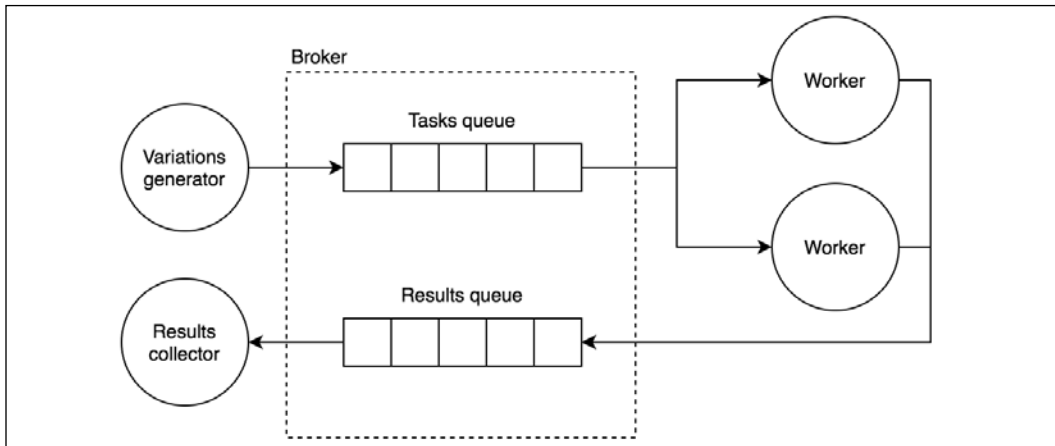


Figure 13.19: Task distribution architecture using a message queue broker

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 actually 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'
import { generateTasks } from './generateTasks.js'

const ALPHABET = 'abcdefghijklmnopqrstuvwxyz'
const BATCH_SIZE = 10000

const [, , maxLength, searchHash] = process.argv

async function main () {
  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) {
        channel.sendToQueue('tasks_queue',           // (2)
            Buffer.from(JSON.stringify(task)))
    }

    await channel.waitForConfirms()
    channel.close()
    connection.close()
}

main().catch(err => console.error(err))

```

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 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 actually new to us. As its name says, that's the API responsible for delivering a message straight to a queue—the `tasks_queue` in our example—bypassing any exchange or routing.

Implementing the worker

On the other side of the `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'
import { processTask } from './processTask.js'

async function main () {
    const connection = await amqp.connect('amqp://localhost')
    const channel = await connection.createChannel()
    const { queue } = await channel.assertQueue('tasks_queue')

```

```

channel.consume(queue, async (rawMessage) => {
  const found = processTask(
    JSON.parse(rawMessage.content.toString()))
  if (found) {
    console.log(`Found! => ${found}`)
    await channel.sendToQueue('results_queue',
      Buffer.from(`Found: ${found}`))
  }

  await channel.ack(rawMessage)
})
}

main().catch(err => console.error(err))

```

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 the `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 result 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'

async function main () {
  const connection = await amqp.connect('amqp://localhost')
  const channel = await connection.createChannel()
  const { queue } = await channel.assertQueue('results_queue')
  channel.consume(queue, msg => {
    console.log(`Message from worker: ${msg.content.toString()}`)
  })
}

main().catch(err => console.error(err))

```


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 f8e966d1e207d02c44511a58dccff2f5429e9a3b
```

With this, we implemented a message pipeline and the Competing Consumers pattern using AMQP.



It's interesting to note that our new version of the hashsum cracker based on AMQP takes slightly longer (compared to the ZeroMQ-based version) to execute all the tasks and find a match. This is a practical demonstration of how a broker can actually introduce a negative performance impact, compared to a more low-level peer-to-peer approach. However, let's not forget that with AMQP we are getting much more out of the box compared to our ZeroMQ implementation. For example, with the AMQP implementation, if a worker crashes, the messages it was processing won't be lost and will eventually be passed to another worker. So, remember to always look at the bigger picture when choosing the right approach to use for your application: a small delay may mean nothing compared to a massive increase in the overall complexity of the system or to a lack of some important features.

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, which comprises a set of consumers identified by a name. When the consumers in the group try to read the stream, they will receive the records in a round-robin configuration.

Each record has to 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.

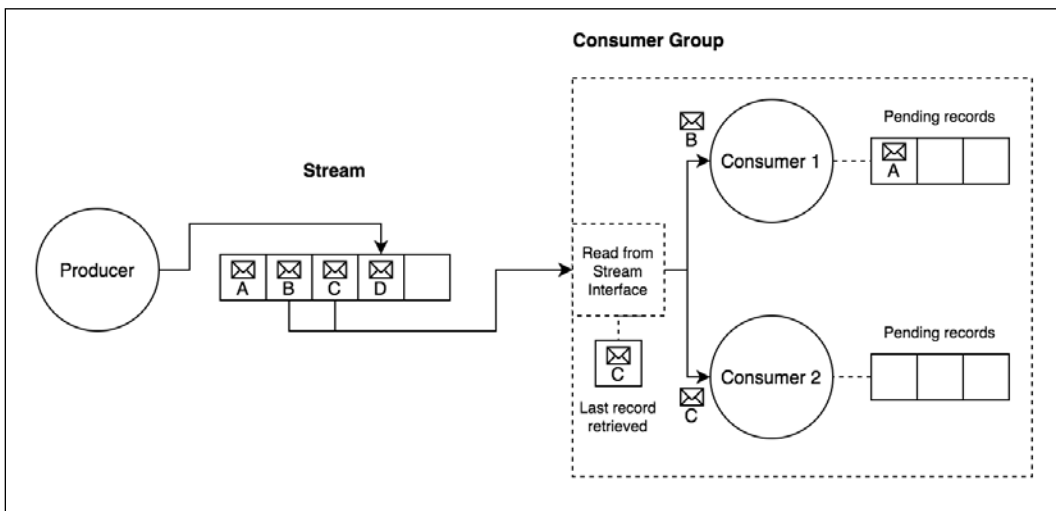


Figure 13.20: A Redis Stream consumer group

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's the next record to read. 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 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 resemble closely 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 the `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'
import { generateTasks } from './generateTasks.js'

const ALPHABET = 'abcdefghijklmnopqrstuvwxyz'
const BATCH_SIZE = 10000
const redisClient = new Redis()

const [, , maxLength, searchHash] = process.argv

async function main () {
  const generatorObj = generateTasks(searchHash, ALPHABET,
    maxLength, BATCH_SIZE)
  for (const task of generatorObj) {
    await redisClient.xadd('tasks_stream', '*',
      'task', JSON.stringify(task))
  }

  redisClient.disconnect()
```

```

}

main().catch(err => console.error(err))

```

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 have to 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'
import { processTask } from './processTask.js'

const redisClient = new Redis()
const [, , consumerName] = process.argv

async function main () {
  await redisClient.xgroup('CREATE', 'tasks_stream',           // (1)
    '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)
}

main().catch(err => console.error(err))
```

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:
 - a. `'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.
 - b. `'tasks_stream'` is the name of the stream we want to read from.
 - c. `'workers_group'` is the name of the consumer group.
 - d. 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.
 - e. `'MKSTREAM'` is an extra parameter that instructs Redis to create the stream if it doesn't exist already.
2. Next, we read all the pending records belonging to the current consumer. Those are the leftover 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 most likely be empty. As we already mentioned, each consumer has access only to its own pending records. We retrieve this list with a `xreadgroup` command and the following arguments:

- a. '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.
 - b. Then we specify the stream we would like to read with 'STREAMS', 'tasks_stream'.
 - c. 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:
 - a. 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.
 - b. 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.
 - c. The next two arguments, 'COUNT' and '1', tell Redis that we are interested in getting one record per call.
 - d. Next, we specify the stream we want to read from with 'STREAMS', 'tasks_stream'.
 - e. 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, we check if we have a match and if that's the case, we append a new record to the `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 amount of arguments required by the various Redis commands.



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 in particular, consumer groups. Check out the official Redis introduction to Streams for more details at nodejsdp.link/redis-streams.

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 communications can give us great advantages in terms of parallelism and efficiency, but alone they are not able to solve all our integration and communication problems. Sometimes, a good old request/reply pattern might just be the perfect tool for the job. But, there are situations in which all we have is an asynchronous one-way channel. It's therefore important to know the various patterns and approaches required to build an abstraction that would allow us to exchange messages in a request/reply fashion on top of a one-way channel. That's exactly what we are going to learn next.

Correlation Identifier

The first Request/Reply pattern that we are going to learn is called the **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 take a look at the example in the following diagram:

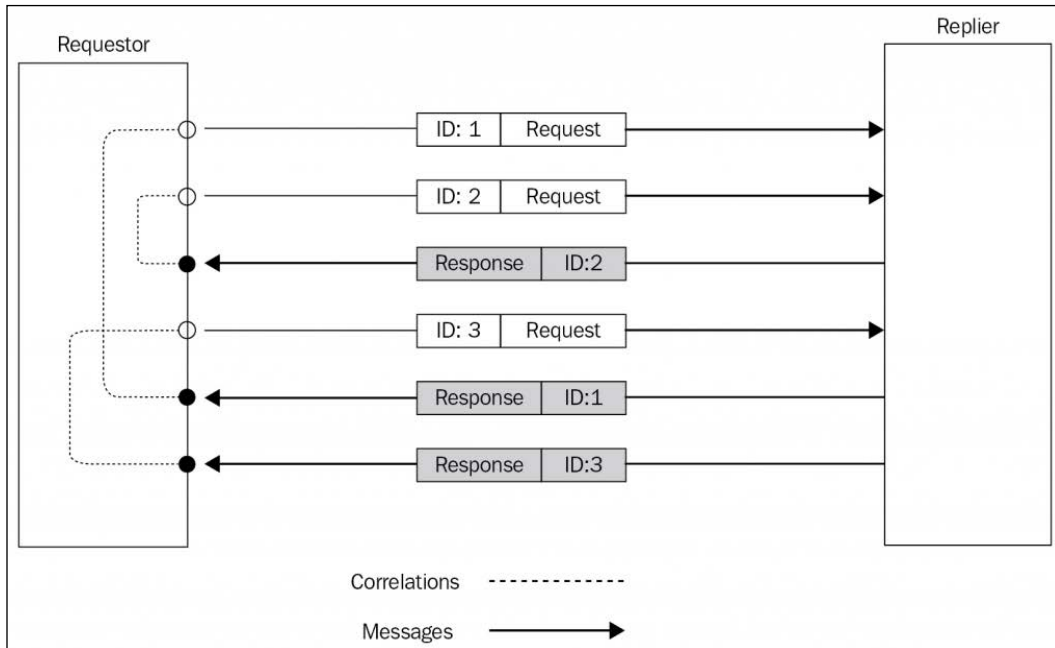


Figure 13.21: Request/reply message exchange using correlation identifiers

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

Let's now start working on an example by choosing the simplest type of one-way channel; one that is point-to-point (which directly connects two nodes of the system) and fully duplex (messages can travel in both directions).

In this *simple channel* category, we can find, for example, WebSockets: they establish a point-to-point connection between the server and browser, and the messages can travel in any direction. Another example is the communication channel that is created when a child process is spawned using `child_process.fork()` (we already met this API in *Chapter 11, Advanced Recipes*). This channel too is asynchronous, point-to-point, and duplex since it connects the parent only with the child process and it allows messages to travel in any direction. This is probably the most basic channel of this category, so that's what we are going to use in the next example.

The plan for the next application is to build an abstraction in order to wrap the channel created between the parent process and the child process. This abstraction should provide a request/reply communication channel by automatically marking each request with a correlation identifier and then matching the ID of any incoming reply against the list of request handlers awaiting a response.

From *Chapter 11, Advanced Recipes*, we should remember that the parent process can send a message to a child with `child.send(message)`, while receiving messages is possible with the `child.on('message', callback)` event handler.

In a similar way, the child process can send a message to the parent process using `process.send(message)` and receive messages with `process.on('message', callback)`.

This means that the interface of the channel available in the parent process is identical to the one available in the child. This will allow us to build a common abstraction that can be used from both ends of the channel.

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'

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 callback = correlationMap.get(message.inReplyTo)
  if (callback) {
    callback(message.data)
  }
})

return sendRequest
}

```

This is how our request abstraction works:

1. The `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.
2. The `sendRequest()` function is used to send new requests. Its job is to generate a 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 the `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) // (1)
      channel.send({                                // (2)
        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 between 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 'child_process'
import { dirname, join } from 'path'
import { fileURLToPath } from 'url'
import { once } from 'events'
import { createRequestChannel } from './createRequestChannel.js'

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

async function main () {
  const channel = fork(join(__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 })           // (3)
    .then(res => {
        console.log(`Reply: 1 + 2 = ${res.sum}`)
    })

    const p2 = request({ a: 6, b: 1, delay: 100 })          // (4)
    .then(res => {
        console.log(`Reply: 6 + 1 = ${res.sum}`)
    })

    await Promise.all([p1, p2])                             // (5)
  } finally {
    channel.disconnect()                                     // (6)
  }
}

main().catch(err => console.error(err))

```

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 similar to 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 is the fundamental pattern for creating a request/reply communication on top of a one-way channel. However, it's not enough when our messaging architecture has more than one channel or queue, or when there can be potentially more than one requestor. In these situations, in addition to a correlation ID, we also need to know the **return address**, a piece of information that allows the replier to send the response back to the original sender of the request.

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 has to 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:

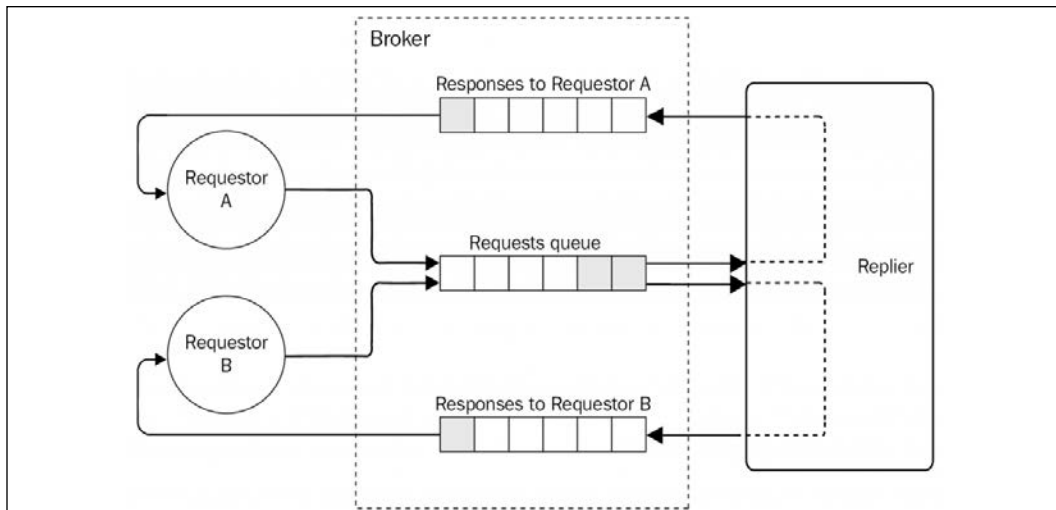


Figure 13.22: Request/reply messaging architecture using AMQP

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 has to 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:

```
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(this.replyQueue, msg => { // (2)  
    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 have to be delivered straight into our response queue. In the second part of the function (2), we start to consume the messages from the `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(queue, // (3)
      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 to 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.

It's important to note that we are using the `channel.sendToQueue()` API instead of `channel.publish()` to send the message. 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'  
  
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 the `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'

async function main () {
  const reply = new AMQPReply('requests_queue')
  await reply.initialize()

  reply.handleRequests(req => {
    console.log('Request received', req)
    return { sum: req.a + req.b }
  })
}

main().catch(err => console.error(err))

```

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 delay from 'delay'

async function main () {
  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 delay(1000)
  }

  request.destroy()
}

main().catch(err => console.error(err))
```

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 and this 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 resembles very closely 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. Here, you learned the most important messaging and integration patterns and the role they play in the design of distributed systems. You should now have mastered the three most important types of message exchange patterns: Publish/Subscribe, Task Distribution, and Request/Reply, implemented either on top of a peer-to-peer architecture or using a broker. We analyzed the pros and cons of each pattern and architecture, and we saw that by using a broker (implementing either a message queue or data stream), it's possible to implement reliable and scalable applications with little effort, but at the cost of having one more system to maintain and scale.

You have also learned how ZeroMQ allows you to build distributed systems where you can have total control over every aspect of the architecture, fine tuning its properties around your very own requirements.

Ultimately, both approaches will give you all the tools that you need to build any type of distributed systems, from basic chat applications to web-scale platforms used by millions of people.

This chapter also closes the book. By now, you should have a toolbelt full of patterns and techniques that you can go and apply in your projects. You should also have a deeper understanding of how Node.js development works and what its strengths and weaknesses are. Throughout the book, you also had the chance to work with a myriad of packages and solutions developed by many extraordinary developers. In the end, this is the most beautiful aspect of Node.js: its people, a community where everybody plays their part in giving something back.

We hope you enjoyed our small contribution and we look forward to seeing yours.

Sincerely, Mario Casciaro and Luciano Mammino.

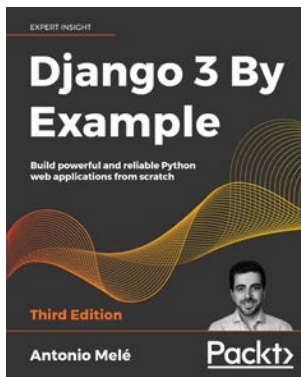
Exercises

- **13.1 History service with streams:** In our publish/subscribe example with Redis Stream, we didn't need a history service (as we did instead 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.

- **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 we 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 current 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 client to 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 (nodejsdp.link/kafka) instead of Redis Streams.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Django 3 By Example - Third Edition

Antonio Melé

ISBN: 9781838981952

- Build real-world web applications
- Learn Django essentials, including models, views, ORM, templates, URLs, forms, and authentication
- Implement advanced features such as custom model fields, custom template tags, cache, middleware, localization, and more

- Create complex functionalities, such as AJAX interactions, social authentication, a full-text search engine, a payment system, a CMS, a RESTful API, and more
- Integrate other technologies, including Redis, Celery, RabbitMQ, PostgreSQL, and Channels, into your projects
- Deploy Django projects in production using NGINX, uWSGI, and Daphne



Responsive Web Design with HTML5 and CSS - Third Edition

Ben Frain

ISBN: 9781839211560

- Integrate CSS media queries into your designs; apply different styles to different devices
- Load different sets of images depending upon screen size or resolution
- Leverage the speed, semantics, and clean markup of accessible HTML patterns
- Implement SVGs into your designs to provide resolution-independent images
- Apply the latest features of CSS like custom properties, variable fonts, and CSS Grid
- Add validation and interface elements like date and color pickers to HTML forms
- Understand the multitude of ways to enhance interface elements with filters, shadows, animations, and more

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

Symbols

@@iterator method 322

@@name convention

reference link 322

A

adaptee 295

Adapter pattern 294

LevelUP, using through filesystem

API 295-298

real-world examples 298

addCps() function 65

Amazon Kinesis

reference link 573

Advanced Message Queuing

Protocol (AMQP) 548, 564

chat application, integrating with 569, 570

competing consumers 588

durable subscribers 566

history service, implementing 567, 568

pipelines 588

point-to-point communications 588

reference link 548

return address pattern,

implementing 605, 606

AMQP, components

binding 565

exchange 565

queue 564

AMQP model

reference link 565

Ansible

URL 501

ANSI escape sequences

reference link 372

Apache Kafka

reference link 573

API orchestration 532-535

URL 532

API proxy 531, 532

API server

without, asynchronous request

batching 439-441

without, asynchronous request

caching 439-441

application scaling 476

archiver

using, reference link 197

async/await 141

limited parallel execution

pattern 149-152

parallel execution 147-149

sequential execution 145-147

sequential iteration 145-147

used, for error handling 143

using, with Array.forEach for

serial execution 147

async functions 141, 142

async generator function 334

async generator objects 334

async generators 334, 335

asynchronicity

with deferred execution 72, 73

asynchronous CPS

versus synchronous CPS 67

asynchronous invocations

cancelable async functions, with

generators 449-453

wrapping 447-449

asynchronously initialized components

- code execution, delaying 430
- dealing with 428
- issue 428, 429
- local initialization check 429
- Mongoose 435
- pre-initialization queues 431-435

asynchronous messaging 546

asynchronous module definition (AMD) 19

asynchronous operations

- cancelable functions, creating 446, 447
- canceled 445

asynchronous programming 63

- issues 90

asynchronous request batching 435-437

- in total sales web server 442, 443
- with promises 441

asynchronous request caching 435

- implementation 445
- in total sales web server 443, 444
- with promises 441

async imports 45-47

async iterables 331, 332

async iterators 178, 331-336

- and Node.js streams 335, 336

async library 119, 120

autocannon

- URL 483

await expression 141, 142

awilix

- reference link 265

B

Babel

- reference link 363

backpressure 176, 181

batch operations

- URL 293

behavioral design patterns

- Command pattern 347
- Iterator pattern 319
- Middleware pattern 337
- State pattern 308
- Strategy pattern 302
- Template pattern 315

binary buffer 558

bound function 349

Brotli

- using 195

browser

- code, sharing with 358, 359

Browserify

- URL 19, 362

Brunch

- reference link 362

buffered API

- using, for Gzipping 162

buffering

- versus streaming 160, 161

Builder pattern 241-244, 248

- URL object builder, implementing 244-247

build-time code branching 374-376

built-in Proxy object 277, 278

- capabilities and limitations 278, 279

C

callback discipline 95

- applying 95-98

callback hell 93, 94

callback pattern 64

- continuation-passing style (CPS) 64

callbacks 63

- best practices 94
- combining, with events 86, 87

chain-of-responsibility pattern 338

- reference link 338

chalk

- reference link 377

Change Observer pattern

- creating, with Proxy pattern 282-285

chat application

- history service, designing for 566, 567
- implementing, with Redis Streams 573-577
- integrating, with AMQP 569, 570

chat server

- peer-to-peer architecture, designing for 558

child process

- communicating with 464, 465

child processes 460

circular dependencies 29-32

- circular dependency resolution, ECMAScript modules (ESM)**
 - evaluation phase 54
 - instantiation phase 53, 54
 - parsing phase 52, 53
- cloning 479**
- closures 64**
 - URL 64
- cloud-based proxies 496**
- cluster module 480**
 - behavior, notes of 481, 482
 - edge cases 482
 - resiliency, and availability with 486-488
 - scaling with 484-486
 - used, for building HTTP server 482-484
 - zero-downtime restart 488-490
- code**
 - sharing, with browser 358, 359
- code minification 360**
- combined streams**
 - implementing 217, 218
- Command Message 544**
- Command pattern 347**
 - client 348
 - command 348
 - complex command 349-353
 - invoker 348
 - target 348
 - Task pattern 349
 - using 348, 349
- CommonJS**
 - URL 19
- CommonJS modules 22**
 - circular dependencies 29-32
 - defining 24
 - homemade module loader 22-24
 - module cache 28
 - module.exports, versus exports 25
 - require function is synchronous 26
 - resolving algorithm 26, 28
 - versus ESM 60
- CommonJS modules, concepts**
 - exports and module.exports variables 22
 - require function 22
- competitive race 108**
- complex applications**
 - decomposing 523

- component 379**
- concurrency**
 - limiting 112
 - limiting, globally 113
- concurrent tasks**
 - race conditions, fixing 108-110
- conditional rendering 388**
- constructor injection 265**
- Consul**
 - URL 504
 - used, for implementing dynamic load balancer 503-509
- consumer groups 593**
- container image 515**
- containers 513**
 - creating, with Docker 514-517
 - running, with Docker 514-517
 - used, for scaling applications 513
- content delivery network (CDN) 370**
- context 302**
- continuation-passing style (CPS) 64**
 - asynchronous CPS 65-67
 - non-CPS callbacks 67
 - synchronous CPS 65
- control flow patterns 94**
- core-js**
 - URL 280
- Correlation Identifier 598**
- CouchDB**
 - URL 491
- CPU-bound tasks**
 - executing 453
 - executing, in production 472
 - external processes, using 460
 - interleaving approach, interleaving 459
 - interleaving, with setImmediate 457
 - subset sum algorithm, interleaving 457, 458
 - subset sum problem, solving 453-457
- cross-origin resource sharing (CORS)**
 - reference link 406
- cross-platform context**
 - JavaScript modules 359, 360
- cross-platform development**
 - build-time code branching 374-376
 - design patterns 378
 - fundamentals 371
 - module swapping 377, 378

runtime code branching 372

csv-parse module

URL 190

D

databases

reference link 337

data ownership 527

Decorator pattern 285, 286

implementation techniques 286

used, for decorating LevelUP database 290

versus Proxy pattern 294

Decorator pattern implementation

techniques 286

composition, using 286, 288

object augmentation (decoration) 288, 289

Proxy object, using 289, 290

default exports 42, 43

default imports 42, 43

delegates

URL 275

demultiplexing 223

denial-of-service (DoS) attack 111, 461

dependency graph 48, 363

dependency hoisting 256

dependency injection container 265

Dependency Injection (DI) 261-264

design patterns, cross-platform development

adapter 378

dependency injection 379

proxy 378

service locator 379

strategy 378

template 378

developer experience (DX) 435

directed graph

URL 48

direct style 65

distributed hashsum cracker, building with

ZeroMQ 580, 581

application, running 587

producer, implementing 582-584

result collector, implementing 586

worker, implementing 585, 586

Docker

URL 514

used, for creating container 514-517

used, for running container 514-517

Dockerfile 515

Document Message 545

Don't Repeat Yourself (DRY) 3

duck typing 128

reference link 239

duplexer2

URL 215

duplexify

URL 215

Duplex streams 185

durable subscriber 562

durable subscribers

with AMQP, and RabbitMQ 566

dynamic horizontal scaling 501

dynamic load balancer, implementing with
Consul 503-509

dynamic load balancer, implementing with
http-proxy 503-509

service registry, using 501-503

dynamic load balancer

implementing, with Consul 503-509

implementing, with http-proxy 503-509

dynamic scaling 501

E

early return principle 96

ECMAScript modules (ESM) 38, 39

async imports 45-47

circular dependency resolution 50, 51

default exports and imports 42, 43

loading 48

loading phases 48, 49

missing references 60, 61

mixed exports 43, 44

modifying 56-59

module identifiers 45

named exports and imports 39-42

read-only live bindings 49, 50

running, in strict mode 60

using, in Node.js 39

versus CommonJS modules 60

Elasticsearch

URL 529

encapsulation 18, 236

entry point 48
ErrorBoundary component
 reference link 388
error handling, async/await
 return, versus return await 144, 145
 try...catch 143, 144
esx
 reference link 384
EventEmitter
 versus callbacks 85, 86
event loop 9
Event Message 545
event notification interface 7
Express
 Middleware pattern 337
 reference link 337, 347
external process
 subset sum task, delegating to 461

F

Factory pattern 234
 code profiler, building 238-240
 encapsulation mechanism 236
 object creation and implementation,
 decoupling 235
fail fast approach 77
Fastify 293
 URL 293, 391
fastify-cors
 reference link 406
first in first out (FIFO) 445
flow control 203
flowing mode 173
function injection 265
functional reactive programming (FRP) 285
FuseBox
 reference link 362

G

garbage collection, JavaScript
 URL 83
generator delegation 331
generator function 327
generator iterator
 controlling 328-330

generator object 327
generators 178, 326
 using, instead of iterators 330, 331
get trap 279
global scope
 modifying 37, 38
glob package
 URL 87
God object 536
Gzipping
 with buffered API 162
 with streams 163

H

HAProxy 496
 URL 496
**hashsum cracker implementation,
 with AMQP** 588, 589
 application, running 592
 producer, implementing 589, 590
 result collector, implementing 591
 worker, implementing 590, 591
**hashsum cracker implementation, with
 Redis Streams** 594
 producer, implementing 594
 worker, implementing 595-598
has trap 279
Hello React 381, 382
high coupling 526
history service
 designing, for chat application 566, 567
 implementing, with AMQP 567, 568
homemade module loader 22-24
horizontal scaling 480
horizontal/vertical partitioning 479
htm
 reference link 384
HTTP/2 Server Push
 reference link 360
http-proxy
 used, for implementing dynamic
 load balancer 503-509
HTTP server
 building 482-484
hydration
 reference link 380

I

Immediately Invoked Function Expression (IIFE) 21, 368

inconsistently asynchronous function 68

inconsistentRead() function

fixing 72, 73

infinite recursive promise resolution chains

issues 152-155

infrastructure as code 501

injector 261

intercepting filter pattern 338

reference link 338

interoperability 61, 62

invertify

reference link 265

Inversion of Control 265

I/O completion port (IOCP) 11

I/O starvation 73

iterable protocol 322-324

iterables 322

as native JavaScript interface 324-326

Iterator pattern 319

async generators 334

async iterators 331

generator function 327, 328

generators 327

iterable protocol 322-324

iterator protocol 319-321

iterator protocol 319-322

iterator result 319

iterators 178

as native JavaScript interface 324-326

reference link 321

J

JavaScript, in Node.js 13

accessing, operating system services 14

module system 14

native code, running 15, 16

running 13, 14

JavaScript modules

in cross-platform context 359, 360

Jest

URL 59

json-socket module

URL 293

JSX

reference link 383

JugglingDB 299

URL 299

K

Keep It Simple, Stupid (KISS) 4

KISS principle (Keep It Simple, Stupid) 90

Knex

reference link 241

Koa

reference link 347

kubecti

URL 519

Kubernetes 517-519

application, deploying and

scaling on 519, 520

rollouts 523

URL 518

Kubernetes deployment

creating 520, 521

scaling 521, 522

L

lazy initialization 280

lazy streams 197, 198

reference link 198

least recently used (LRU) 445

LevelDB 290

level-filesystem

URL 299

LevelGraph

URL 290

level package

URL 291

LevelUP 290

ecosystem, URL 290

URL 290

using, through filesystem API 295-298

LevelUP database

decorating, with Decorator pattern 290

LevelUP plugin

implementing 291-293

level-inverted-index, URL 293

levelplus, URL 293

- libuv 11**
 - URL 12
- links**
 - sequential crawling 101, 102
- Linux container 513**
- load balancer, implementing with reverse proxy**
 - options 496
- load balancing 479**
 - with Nginx 496-501
- load distribution 477**
- loading phases, ECMAScript modules (ESM)**
 - construction phase 48
 - evaluation phase 49
 - instantiation phase 49
- lodash**
 - URL 45
- log 546, 571**
- logging Writable stream**
 - creating, in Proxy pattern 281, 282
- long-term support (LTS) 14**
- LoopBack 285**
 - URL 285

M

- Memcached**
 - URL 491
- memory leaks**
 - observer pattern 82
- message 544**
 - Command Message 544
 - Document Message 545
 - Event Message 545
- message broker 547**
 - integrating with 536-538
 - Redis, using as 554-556
- message queue (MQ) 546, 562**
 - versus streams 573
- Message Queue Telemetry Transport (MQTT) 548**
 - reference link 548
- messaging system**
 - fundamentals 542
- Microbundle**
 - reference link 362

- microservice architecture 526**
 - advantages 528
 - disadvantages 528
 - example 526-528
 - integration patterns 530
- microservice architecture, advantages and disadvantages**
 - across platforms and languages, reusability 529
 - application, scaling 529
 - challenges 530
 - service, expendable 529
- microservice architecture, factors**
 - high cohesion 526
 - integration complexity 526
 - loose coupling 526
- microservice architecture, integration patterns**
 - API orchestration 532-535
 - API proxy 531, 532
 - integrating, with message broker 536-538
- microtasks 73**
- middleware framework**
 - creating, for ØMQ 340
- middleware framework, for ØMQ**
 - creating 340
 - middleware functions, creating to process messages 342, 343
 - Middleware Manager 340-342
- Middleware pattern 337**
 - implementing 339
 - in Express 337
 - Middleware Manager 339
 - using 338
- Middy**
 - reference link 347
- minikube**
 - URL 519
- mixed exports 43, 44**
- mkdirp 90**
- MobX 285**
 - URL 285
- mocku module**
 - URL 59
- module bundler 360**
 - dependency resolution 363-366
 - packing 367, 368

- working 363
- module cache 28**
- module definition patterns 33**
 - class, exporting 35
 - function, exporting 34
 - instance, exporting 36
 - named exports 33
- module identifiers 45**
- module objects**
 - reference link 59
- module systems 20**
 - in JavaScript 19, 20
 - in Node.js 19, 20
 - modifying 37, 38
 - need for 18
- MongoDB 435**
 - URL 491
- monkey patching 37, 275**
- monolithic application 478**
- monolithic architecture 524, 525**
- monolithic kernels 524**
- multi-pass rendering 413**
- multiplexing 223**
- multi-process approach**
 - considerations 467
- mux/demux application**
 - running 228

N

- named exports 39-42**
- named imports 39-42**
- namespace import 41**
- nanoSQL 299**
 - URL 299
- Next.js framework**
 - reference link 416
- Nginx**
 - load balancing with 496-500
- Nginx**
 - URL 496
- nock**
 - URL 38
- Node.js**
 - callback conventions 73
 - core 2
 - ECMAScript modules (ESM), using 39

- event demultiplexing 7-9
- I/O, blocking 5, 6
- I/O, disadvantages 5
- I/O engine, libuv of 11
- I/O, non-blocking 6, 7
- modules, advantages 3
- modules, using 2, 3
- philosophy 2
- philosophy, reference link 2
- reactor pattern 9-11
- recipe for 12
- simplicity and pragmatism 4
- surface area 3
- working 5

Node.js applications

- scaling 477
- scaling, with containers 513

Node.js-based proxies 496

Node.js callback conventions 73-77

- errors, propagating 74, 75
- uncaught exceptions 75-77

Node.js release cycles

- reference link 14

non-flowing mode 171, 172

nunjucks

- reference link 370

O

Object.defineProperty() method

- URL 280

object streams

- demultiplexing 229
- multiplexing 229

observer pattern 77

- asynchronous event 83-85
- errors, propagating 80
- EventEmitter 78
- EventEmitter, and memory leaks 82, 83
- EventEmitter, creating 79, 80
- EventEmitter, methods 78
- EventEmitter, using 79, 80
- observable object, creating 80-82
- synchronous event 83-85

ØMQ

- middleware framework, creating 340
- reference link 340

ØMQ middleware framework

- client-side, creating 345, 346
- reference link 344
- server-side, creating 344, 345
- using 344

one way pattern

- versus request/reply patterns 542, 543

Open Container Initiative (OCI)

- URL 513

operational transformation (OT) 348

- reference link 349

optimal asynchronous request

caching 437, 438

- phases 438

ORY

- URL 529

P

Packer

- URL 501

packet switching 224

parallel execution 104, 105

- limited parallel execution 110, 111
- pattern 108

parallel pipeline 579

parallel-transform

- URL 212

parcel

- reference link 362

passport 308

- reference link 308
- supported authentication providers 308

PassThrough streams 193

- late piping 194-196
- observability 193, 194

peer-to-peer architecture

- designing, for chat server 558

peer-to-peer communication 548

peer-to-peer load balancing 510, 511

- HTTP client, implementing that can balance requests across multiple servers 511-513

Peer-to-Peer Publish/Subscribe

- with ZeroMQ 557

pipeline

- processing 338

pipeline()

- used, for error handling 201, 202

pipes

- used, for connecting streams 198, 199

polyfill 280

PostgreSQL

- URL 491

PouchDB

- URL 290

private class fields

- reference link 237

private variables

- reference link 237

process identifier (PID) 483

process pool module

- implementing 461-463

promise 124-127

- creating 130
- limited parallel execution 137
- parallel execution flow 136, 137
- sequential execution 133-136
- sequential iteration 133-136
- used, for implementing TaskQueue class 138, 139

promise API 128, 129

Promises/A 127

Promises/A+ 124, 128

- URL 128

promisification 131, 132

property injection 265

proxy 269

Proxy pattern 269, 270

- implementing, techniques 271
- logging Writable stream, creating 281, 282
- used, for creating Change Observer pattern 282-285
- uses, example 270
- using, in projects 285
- versus Decorator pattern 294

Proxy pattern implementation

techniques 271, 272

- built-in Proxy object 277, 278
- comparing 280, 281
- object augmentation 275, 276
- object composition 272-275

Publish/Subscribe pattern 549

- real-time chat application, building 550

pumpify

URL 216

pyramid of doom 93

Q

queues 113, 114

R

RabbitMQ

durable subscribers 566

reference link 548

race conditions

fixing, with concurrent tasks 108-110

reachability

URL 83

React 379, 380

reference link 391

stateful components 385-390

react.createElement

alternatives 383-385

React Hardware

reference link 380

reactive programming (RP) 285

React Native

reference link 380

reactor pattern 9, 11

using 10

React PIXI

reference link 380

React Router

reference link 391

react-three-fiber

reference link 380

Readable streams

approaches 171

async iterators 174

flowing mode 173

from iterables 178, 179

implementing 174-177

non-flowing mode 171, 172

simplified construction 177

real-time chat application

building 550

client side, implementing 551, 552

running 553

scaling 553

server side, implementing 550, 551

records 571

Redis

reference link 554

URL 491

using, as message broker 554-556

Redis consumer groups 593

Redis Streams

chat application, implementing 573-577

tasks, distributing 592

reliable message delivery

with queues 562, 563

reliable messaging

with streams 571

remote procedure call (RPC) 348, 544

request/reply abstraction, implementing with correlation identifiers 599, 600

reply, abstracting 602

request, abstracting 600, 601

request/reply cycle

trying 603, 604

request/reply patterns 598

versus one way pattern 542, 543

require function is synchronous 26

requirejs

URL 19

resiliency 486

resolving algorithm 26-28

URL 27

resolving algorithm, branches

core modules 27

file modules 27

package modules 27

return address 605

return address pattern, AMQP

implementing 605

replier, implementing 609-611

reply abstraction, implementing 608, 609

request abstraction, implementing 606, 607

requestor, implementing 609-611

Revealing Constructor pattern 249, 250, 253

immutable buffer, building 250-252

revealing module pattern 20-22

reverse proxy

scaling with 494-496

rollup

reference link 362

runtime code branching 372
challenges 373, 374

S

scalability

three dimensions 477, 478

scalability, three dimensions

X-axis 478

Y-axis 478, 479

Z-axis 479

scale cube 475, 477

scaling

with cluster module 484-486

with reverse proxy 494-496

semicoroutines 326

sequential crawling

of links 101, 102

sequential execution flow 98

sequential iteration 100

pattern 103

service-level agreement (SLA) 488

service locator 265

service locator pattern

reference link 368

service registry

using 501-503

set of tasks

executing, in sequence 99

Simple/Streaming Text Orientated Messaging Protocol (STOMP) 548

reference link 548

single-page applications (SPAs) 380

single-responsibility principle (SRP) 35

Singleton dependencies 258-261

Singleton pattern 253-257

Socket.IO

URL 494

SQLite 258

sqlite3

reference link 258

stateful communications

dealing with 490, 491

state across multiple instances,
sharing 491, 492

sticky load balancing 492-494

State pattern 308, 309

used, for implementing failsafe

socket 310-314

state transition 310

sticky load balancing 492-494

URL 494

Store front-end 535

strategies 302

Strategy pattern 302-304

approaches 307

multi-format configuration objects

example 304-307

Passport 308

streaming

versus buffering 160, 161

streaming platform

characteristics 571, 572

streams 546, 547, 571

anatomy 170, 171

asynchronous control flow patterns with 203

combining 214-216

composability 167

connecting, with pipes 198, 199

data, handling in sequence 203-205

demultiplexing 223

error handling, with pipeline() 201, 202

forking 219, 220

for reliable messaging 571

importance, discovering 160

merging 221

multiple checksum generator,
implementing 220

multiplexing 223

mux/demux application, running 228

ordered parallel execution 212, 213

pipes and error handling 200

pipng patterns 214

remote logger, building 224

spatial efficiency 161, 162

text files, merging 221, 222

time efficiency 163-167

unordered limited parallel execution 210-212

used, for Gzipping 163

used, for implementing unordered
parallel tasks 206

versus message queues 573

working with 170

streams, composability
client-side encryption, adding 167, 168
server-side decryption, adding 169, 170

streams, demultiplexing
server side 226-228

streams, multiplexing
client side 224-226

streams, operating modes
binary mode 170
object mode 170

structural design patterns
Adapter pattern 294
Decorator pattern 285
Proxy pattern 269

stubs 317

subset sum problem 453

subset sum task
delegating, to external process 461
executing, in worker threads 469-472

substack pattern
URL 34

superagent 90
reference link 248, 406

surrogate 269

symbols
reference link 237

synchronous APIs
using 70, 71

synchronous CPS
versus asynchronous CPS 67

T

task distribution patterns 577-579

Task pattern 349

TaskQueue 115, 116

TaskQueue class
implementing, with promise 138, 139

tasks
distributing, with Redis Streams 592

template methods 315

Template pattern 315, 318
configuration manager template 316-318
purpose 315

ternary-stream package
URL 229

Terraform
URL 501

Terser
reference link 376

thenables 127, 128

Transform streams 185, 186
data, aggregating with 189-192
data, filtering with 189-192
implementing 186-188
simplified construction 188, 189

transpilation (transcompilation) 280

trap methods 277

URLs 279

tree shaking
reference link 366

U

Universal data retrieval 411
async pages 414-416
async pages implementation 416-424
two-pass rendering 412, 413

Universal JavaScript app
asynchronous data retrieval 405-411
creating 391
frontend only-app 392-399
server-side rendering 399-404

Universal Module Definition (UMD)
URL 19

unordered parallel stream
implementing 206-208

URL status monitoring application
implementing 208-210

userland 2

V

variable 302

vertical scaling 480

virtual DOM
reference link 379

virtual machines 514

Vue.js
version 3 285
version 3, URL 285

W

WeakMaps

reference link 237

WebAssembly

reference link 16

webpack

reference link 362

URL 19

using 369-371

webpack, alternatives

browserify 362

parcel 362

rollup 362

web spider

creating 90-92

updating 139, 140

web spider version 2 100

web spider version 3 106, 107

web spider version 4 116-119

wiring modules 257

worker module

implementing 466

worker threads 468

subset sum task, executing 469-472

using 468

Writable streams 179

backpressure 181, 182

data, pushing 179-181

implementing 182, 183

simplified construction 184

Z

Zalgo 70

unleashing 68-70

zero-downtime restart 488-490

ZeroMQ 340, 557

distributed hashsum cracker,
building 580, 581

reference link 314, 557

URL 511

ZeroMQ Fanout/Fanin pattern 579

PUSH/PULL sockets 580

ZeroMQ package

reference link 337

ZeroMQ PUB/SUB sockets

using 559-561

ZMQ 340

