

Node.js Web Development

Fifth Edition

Server-side web development made easy with Node 14 using practical examples

A photograph of a field filled with small, vibrant yellow and orange flowers, possibly marigolds or cosmos, growing in green grass. Sunlight filters through the leaves, creating bright rays of light that fan out across the scene.

David Herron

Packt

www.packt.com

Node.js Web Development

Fifth Edition

Server-side web development made easy with Node 14
using practical examples

David Herron



BIRMINGHAM - MUMBAI

Node.js Web Development

Fifth 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 author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

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

Commissioning Editor: Ashwin Nair

Acquisition Editor: Larissa Pinto

Content Development Editor: Aamir Ahmed

Senior Editor: Hayden Edwards

Technical Editor: Deepesh Patel

Copy Editor: Safis Editing

Project Coordinator: Kinjal Bari

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Joshua Misquitta

First published: August 2011

Second edition: July 2013

Third edition: June 2016

Fourth edition: May 2018

Fifth edition: July 2020

Production reference: 1300720

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83898-757-2

www.packtpub.com

I wish to thank my mother, Evelyn, for everything, and my father, Jim, my sister, Patti, and my brother, Ken, for having launched me on this journey in life. To my partner, Maggie, may we have many years of mutual encouragement together. I wish to thank Dr. Kubota, of the University of Kentucky, for believing in me, for giving me a start in the computer industry, and for all the other students he nurtured in the same way. I am grateful to the Node.js core team members and the programming platform they've created.

- David Herron



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
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at 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 author

David Herron is a software engineer living in Silicon Valley who has worked on projects ranging from an X.400 email server to being part of the team that launched the OpenJDK project, to Yahoo's Node.js application-hosting platform, and a solar array performance monitoring service. That took David through several companies until he grew tired of communicating primarily with machines, and developed a longing for human communication. Today, David is an independent writer of books and blog posts covering topics related to technology, programming, electric vehicles, and clean energy technologies. The blog posts appear on TechSparx, Medium, GreenTransportation, and LongTailPipe. Using Node.js, he is the creator of AkashaCMS, a static website generator, and AkashaEPUB, a tool for generating eBooks.

About the reviewers

Esref Durna has worked as a full stack engineer since 2004. He has worked as first engineer at several Silicon Valley start-ups, and currently works at American Express as a full stack engineer focused on micro-frontends.

Migsar Navarro is a full stack developer who loves programming in JavaScript because he finds it one of the most flexible and expressive programming languages. He enjoys building web applications that are as useful for the users as they are for the programmers that write the code. He is passionate about sharing knowledge, demystifying engineering, software architecture, and geographical information systems. He lives in Porto with his girl and his daughter.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
<hr/>	
Section 1: Introduction to Node.js	
<hr/>	
Chapter 1: About Node.js	9
Overview of Node.js	10
The capabilities of Node.js	11
What are folks doing with Node.js?	12
Server-side JavaScript	13
Why should you use Node.js?	14
Popularity	14
JavaScript everywhere	15
Leveraging Google's investment in V8	16
Leaner, asynchronous, event-driven model	16
Microservice architecture	16
Node.js is stronger after a major schism and hostile fork	17
The Node.js event-driven architecture	17
The Node.js answer to complexity	19
Asynchronous requests in Node.js	20
Performance and utilization	22
Is Node.js a cancerous scalability disaster?	24
Server utilization, overhead costs, and environmental impact	27
Embracing advances in the JavaScript language	28
Deploying ES2015/2016/2017/2018 JavaScript code	30
TypeScript and Node.js	31
Developing microservices or maxiservices with Node.js	31
Summary	32
Chapter 2: Setting Up Node.js	34
System requirements	34
Installing Node.js using package managers	35
Installing Node.js on macOS with MacPorts	35
Installing Node.js on macOS with Homebrew	36
Installing Node.js on Linux, *BSD, or Windows from package management systems	37
Installing Node.js in WSL	38
Opening an administrator-privileged PowerShell on Windows	39
Installing the Node.js distribution from nodejs.org	39
Installing from the source on POSIX-like systems	40
Installing prerequisites	41

Installing developer tools on macOS	42
Installing from the source for all POSIX-like systems	43
Installing from the source on Windows	45
Installing multiple Node.js instances with nvm	45
Installing nvm on Windows	47
Requirements for installing native code modules	48
Choosing Node.js versions to use and the version policy	50
Choosing editors and debuggers for Node.js	51
Running and testing commands	52
Using Node.js's command-line tools	52
Running a simple script with Node.js	54
Writing inline async arrow functions	56
Converting to async functions and the Promise paradigm	58
Launching a server with Node.js	59
Using npm, the Node.js package manager	60
Using npx to execute Node.js packaged binaries	62
Advancing Node.js with ECMAScript 2015, 2016, 2017, and beyond	62
Using Babel to use experimental JavaScript features	66
Summary	70
Chapter 3: Exploring Node.js Modules	72
Defining a Node.js module	73
Examining the traditional Node.js module format	73
Examining the ES6/ES2015 module format	77
Injected objects in ES6 modules	81
Computing the missing __dirname variable in ES6 modules	82
Using CommonJS and ES6 modules together	83
Using ES6 modules from CommonJS using import()	85
Hiding implementation details with encapsulation in CommonJS and ES6 modules	87
Using JSON modules	88
Supporting ES6 modules on older Node.js versions	90
Finding and loading modules using require and import	92
Understanding File modules	92
The ES6 import statement takes a URL	94
Understanding the Node.js core modules	94
Using a directory as a module	95
Comparing installed packages and modules	96
Finding the installed package in the file system	97
Handling multiple versions of the same installed package	99
Searching for globally installed packages	100
Reviewing module identifiers and pathnames	101
Using deep import module specifiers	102
Overriding a deep import module identifier	103
Studying an example project directory structure	103

Loading modules using require, import, and import()	105
Using npm – the Node.js package management system	108
The npm package format	108
Accessing npm helpful documentation	109
Initializing a Node.js package or project with npm init	109
Finding npm packages	111
The package.json fields that help finding packages	112
Installing an npm package	113
Installing a package by version number	114
Installing packages from outside the npm repository	115
Global package installs	116
Avoiding global module installation	116
Maintaining package dependencies with npm	117
Automatically updating package.json dependencies	119
Fixing bugs by updating package dependencies	119
Explicitly specifying package dependency version numbers	120
Packages that install commands	121
Configuring the PATH variable to handle locally installed commands	122
Configuring the PATH variable on Windows	123
Avoiding modifications to the PATH variable	124
Updating packages you've installed when they're outdated	125
Automating tasks with scripts in package.json	126
Declaring Node.js version compatibility	127
Publishing an npm package	128
The Yarn package management system	128
Summary	130
Chapter 4: HTTP Servers and Clients	131
Sending and receiving events with EventEmitter	132
JavaScript classes and class inheritance	133
The EventEmitter class	135
The EventEmitter theory	137
Understanding HTTP server applications	138
ES2015 multiline and template strings	142
HTTP Sniffer – listening to the HTTP conversation	144
Web application frameworks	146
Getting started with Express	147
Setting environment variables in the Windows cmd.exe command line	151
Walking through the default Express application	153
Understanding Express middleware	156
Contrasting middleware and request handlers	159
Error handling	161
Creating an Express application to compute Fibonacci numbers	162
Computationally intensive code and the Node.js event loop	168
Algorithmic refactoring	170

Making HTTPClient requests	174
Calling a REST backend service from an Express application	176
Implementing a simple REST server with Express	177
Refactoring the Fibonacci application to call the REST service	181
Some RESTful modules and frameworks	184
Summary	185
Section 2: Developing the Express Application	
Chapter 5: Your First Express Application	187
Exploring Promises and async functions in Express router functions	188
Promises and error handling in Express router functions	191
Integrating async functions with Express router functions	193
Architecting an Express application in the MVC paradigm	196
Creating the Notes application	197
Rewriting the generated router module as an ES6 module	198
Creating the Notes application wiring – app.mjs	199
Implementing the Notes data storage model	203
Data hiding in ES-2015 class definitions	205
Implementing an in-memory Notes datastore	208
The Notes home page	209
Adding a new note – create	213
Viewing notes – read	217
Editing an existing note – update	219
Deleting notes – destroy	220
Theming your Express application	222
Scaling up – running multiple Notes instances	224
Summary	226
Chapter 6: Implementing the Mobile-First Paradigm	227
Understanding the problem – the Notes app isn't mobile-friendly	228
Learning the mobile-first paradigm theory	230
Using Twitter Bootstrap on the Notes application	232
Setting up Bootstrap	232
Adding Bootstrap to the Notes application	234
Alternative layout frameworks	237
Flexbox and CSS Grids	238
Mobile-first design for the Notes application	238
Laying the Bootstrap grid foundation	239
Responsive page structure for the Notes application	241
Using icon libraries and improving visual appeal	242
Responsive page header navigation bar	243
Improving the Notes list on the front page	245

Cleaning up the note viewing experience	248
Cleaning up the add/edit note form	249
Cleaning up the delete-note window	251
Customizing a Bootstrap build	253
Using third-party custom Bootstrap themes	258
Summary	260
Chapter 7: Data Storage and Retrieval	261
Remembering that data storage requires asynchronous code	262
Logging and capturing uncaught errors	262
Request logging with morgan	264
Debugging messages	267
Capturing stdout and stderr	269
Capturing uncaught exceptions and unhandled rejected Promises	270
Storing notes in a filesystem	271
Dynamically importing ES6 modules	275
Running the Notes application with filesystem storage	278
Storing notes with the LevelDB datastore	279
Closing database connections when closing the process	283
Storing notes in SQL with SQLite3	285
The SQLite3 database schema	285
The SQLite3 model code	287
Running Notes with SQLite3	291
Storing notes the ORM way with Sequelize	293
Configuring Sequelize and connecting to a database	294
Creating a Sequelize model for the Notes application	297
Running the Notes application with Sequelize	301
Storing notes in MongoDB	303
A MongoDB model for the Notes application	304
Running the Notes application with MongoDB	309
Summary	310
Chapter 8: Authenticating Users with a Microservice	312
Creating a user information microservice	313
Developing the user information model	315
Creating a REST server for user information	319
Creating a command-line tool to test and administer the user authentication server	323
Creating a user in the user information database	327
Reading user data from the user information service	331
Updating user information in the user information service	334
Deleting a user record from the user information service	335
Checking the user's password in the user information service	337
Providing login support for the Notes application	339
Accessing the user authentication REST API	340

Incorporating login and logout routing functions in the Notes application	344
Login/logout changes to app.mjs	349
Login/logout changes in routes/index.mjs	351
Login/logout changes required in routes/notes.mjs	352
Viewing template changes supporting login/logout	354
Running the Notes application with user authentication	358
Providing Twitter login support for the Notes application	360
Registering an application with Twitter	361
Storing authentication tokens	363
Implementing TwitterStrategy	365
Keeping secrets and passwords secure	374
Adding password encryption to the user information service	375
Implementing encrypted password support in the Notes application	379
Running the Notes application stack	380
Summary	381
Chapter 9: Dynamic Client/Server Interaction with Socket.IO	383
Introducing Socket.IO	384
Initializing Socket.IO with Express	386
Real-time updates on the Notes homepage	389
Refactoring the NotesStore classes to emit events	389
Real-time changes in the Notes home page	392
Changing the home page and layout templates	394
Adding a Socket.IO client to the Notes home page	396
Running Notes with real-time home page updates	398
A word on enabling debug tracing in Socket.IO code	399
Real-time action while viewing notes	400
Changing the note view template for real-time action	402
Running Notes with pseudo-real-time updates while viewing a note	404
Inter-user chat and commenting for Notes	404
Data model for storing messages	405
Adding support for messages to the Notes router	410
Changing the note view template for messages	413
Composing messages on the Note view page	413
Showing any existing messages on the Note view page	417
Deleting messages on the Notes view page	420
Running Notes and passing messages	421
Summary	422
Section 3: Deployment	
Chapter 10: Deploying Node.js Applications to Linux Servers	425
Notes application architecture and deployment considerations	426
Traditional Linux deployment for Node.js services	427
Installing Multipass	429
Handling a failure to launch Multipass instances on Windows	431

Provisioning a server for the user authentication service	433
Testing the deployed user authentication service	438
Script execution in PowerShell on Windows	440
Provisioning a server for the Notes service	441
Adjusting Twitter authentication to work on the server	446
Setting up PM2 to manage Node.js processes	448
Familiarizing ourselves with PM2	448
Scripting the PM2 setup on Multipass	450
Integrating the PM2 setup as persistent background processes	454
Summary	456
Chapter 11: Deploying Node.js Microservices with Docker	458
Setting up Docker on your laptop or computer	460
Installing and starting Docker with Docker for Windows or macOS	461
Familiarizing ourselves with Docker	462
Setting up the user authentication service in Docker	464
Launching a MySQL container in Docker	465
The ephemeral nature of Docker containers	468
Defining the Docker architecture for the authentication service	469
Creating the MySQL container for the authentication service	472
Security in the database container	476
Dockerizing the authentication service	479
Creating the authentication service Dockerfile	480
Building and running the authentication service Docker container	482
Exploring AuthNet	485
Creating FrontNet for the Notes application	487
MySQL container for the Notes application	488
Dockerizing the Notes application	489
Managing multiple containers with Docker Compose	497
Docker Compose file for the Notes stack	498
Building and running the Notes application with Docker Compose	502
Using Redis for scaling the Notes application stack	507
Testing session management with multiple Notes service instances	508
Storing Express/Passport session data in a Redis server	509
Distributing Socket.IO messages using Redis	512
Summary	514
Chapter 12: Deploying a Docker Swarm to AWS EC2 with Terraform	516
Signed up with AWS and configuring the AWS CLI	518
Finding your way around the AWS account	519
Setting up the AWS CLI using AWS authentication credentials	520
Creating an IAM user account, groups, and roles	523
Creating an EC2 key pair	527
An overview of the AWS infrastructure to be deployed	528
Using Terraform to create an AWS infrastructure	531

Configuring an AWS VPC with Terraform	534
Configuring the AWS gateway and subnet resources	537
Deploying the infrastructure to AWS using Terraform	540
Setting up a Docker Swarm cluster on AWS EC2	545
Deploying a single-node Docker Swarm on a single EC2 instance	547
Adding an EC2 instance and configuring Docker	547
Launching the EC2 instance on AWS	551
Handling the AWS EC2 key-pair file	554
Testing the initial Docker Swarm	554
Setting up remote control access to a Docker Swarm hosted on EC2	558
Setting up ECR repositories for Notes Docker images	561
Using environment variables for AWS CLI commands	563
Defining a process to build Docker images and push them to the AWS ECR	564
Creating a Docker stack file for deployment to Docker Swarm	569
Creating a Docker stack file from the Notes Docker compose file	571
Placing containers across the swarm	574
Configuring secrets in Docker Swarm	576
Persisting data in a Docker swarm	580
Provisioning EC2 instances for a full Docker swarm	582
Configuring EC2 instances and connecting to the swarm	583
Implementing semi-automatic initialization of the Docker Swarm	587
Preparing the Docker Swarm before deploying the Notes stack	590
Deploying the Notes stack file to the swarm	594
Preparing to deploy the Notes stack to the swarm	595
Deploying the Notes stack to the swarm	596
Verifying the correct launch of the Notes application stack	597
Diagnosing a failure to launch the database services	599
Testing the deployed Notes application	600
Logging in with a regular account on Notes	601
Diagnosing an inability to log in with Twitter credentials	603
Scaling the Notes instances	604
Summary	607
Chapter 13: Unit Testing and Functional Testing	609
Assert – the basis of testing methodologies	610
Testing a Notes model	612
Mocha and Chai – the chosen test tools	612
Notes model test suite	613
Creating the initial Notes model test case	614
Running the first test case	616
Adding some tests	618
More tests for the Notes model	620
Diagnosing test failures	623
Testing against databases that require server setup – MySQL and MongoDB	626
Using Docker Swarm to manage test infrastructure	627

Using Docker Swarm to deploy test infrastructure	628
Executing tests under Docker Swarm	632
MongoDB setup under Docker and testing Notes against MongoDB	635
Testing REST backend services	638
Automating test results reporting	644
Frontend headless browser testing with Puppeteer	646
Setting up a Puppeteer-based testing project directory	647
Creating an initial Puppeteer test for the Notes application stack	648
Executing the initial Puppeteer test	651
Testing login/logout functionality in Notes	652
Testing the ability to add Notes	655
Implementing negative tests with Puppeteer	659
Testing login with a bad user ID	660
Testing a response to a bad URL	661
Improving testability in the Notes UI	662
Summary	663
Chapter 14: Security in Node.js Applications	665
Implementing HTTPS in Docker for deployed Node.js applications	666
Assigning a domain name for an application deployed on AWS EC2	668
Updating the Twitter application	669
Planning how to use Let's Encrypt	670
Using NGINX and Let's Encrypt in Docker to implement HTTPS for Notes	672
Adding the Cronginx container to support HTTPS on Notes	672
Creating an NGINX configuration to support registering domains with Let's Encrypt	675
Adding the required directories on the EC2 host	676
Deploying the EC2 cluster and Docker swarm	677
Registering a domain with Let's Encrypt	678
Implementing an NGINX HTTPS configuration using Let's Encrypt certificates	680
Testing HTTPS support for the Notes application	685
Using Helmet for across-the-board security in Express applications	686
Using Helmet to set the Content-Security-Policy header	687
Making the ContentSecurityPolicy configurable	689
Using Helmet to set the X-DNS-Prefetch-Control header	691
Using Helmet to control enabled browser features using the Feature-Policy header	692
Using Helmet to set the X-Frame-Options header	693
Using Helmet to remove the X-Powered-By header	693
Improving HTTPS with Strict Transport Security	694
Mitigating XSS attacks with Helmet	695
Addressing Cross-Site Request Forgery (CSRF) attacks	696

Table of Contents

Denying SQL injection attacks	698
Scanning for known vulnerabilities in Node.js packages	699
Using good cookie practices	702
Hardening the AWS EC2 deployment	703
AWS EC2 security best practices	708
Summary	709
Other Books You May Enjoy	711
Index	714

Preface

Node.js is a server-side JavaScript platform that allows developers to build fast and scalable applications using JavaScript outside of web browsers. It is playing an ever-wider role in the software development world, having started as a platform for server applications but now seeing wide use in command-line developer tools and even in GUI applications, thanks to toolkits such as Electron. Node.js has liberated JavaScript from being stuck in the browser.

It runs on top of the ultra-fast JavaScript engine at the heart of Google's Chrome browser, V8. The Node.js runtime follows an ingenious event-driven model that's widely used for concurrent processing capacity despite using a single-thread model.

The primary focus of Node.js is high-performance, highly scalable web applications, but it is seeing adoption in other areas. For example, Electron, the Node.js-based wrapper around the Chrome engine, lets Node.js developers create desktop GUI applications and is the foundation on which many popular applications have been built, including the Atom and Visual Studio Code editors, GitKraken, Postman, Etcher, and the desktop Slack client. Node.js is popular on Internet of Things devices. Its architecture is especially well suited to microservice development and often helps form the server side of full-stack applications.

The key to providing high throughput on a single-threaded system is Node.js's model for asynchronous execution. It's very different from platforms that rely on threads for concurrent programming, as those systems often have high overheads and complexity. By contrast, Node.js uses a simple event dispatch model that originally relied on callback functions but today relies on the JavaScript Promise object and `async` functions.

Because Node.js is on top of Chrome's V8 engine, the platform is able to quickly adopt the latest advances in the JavaScript language. The Node.js core team works closely with the V8 team, letting it quickly adopt new JavaScript language features as they are implemented in V8. Node.js 14.x is the current release and this book is written for that release.

Who this book is for

Server-side engineers may find JavaScript to be an excellent alternative programming language. Thanks to advances in the language, JavaScript long ago stopped being a simplistic toy language suitable only for animating buttons in browsers. We can now build large systems with the language, and Node.js has many built-in features, such as a top-notch module system, that help in larger projects.

Developers experienced with browser-side JavaScript may find it attractive to broaden their horizons to include server-side development using this book.

What this book covers

Chapter 1, *About Node.js*, introduces you to the Node.js platform. It covers its uses, the technological architecture choices in Node.js, its history, the history of server-side JavaScript, why JavaScript should be liberated from the browser, and important recent advances in the JavaScript scene.

Chapter 2, *Setting Up Node.js*, goes over setting up a Node.js developer environment. This includes installing Node.js on Windows, macOS, and Linux. Important tools are covered, including the `npm` and `yarn` package management systems and Babel, which is used to transpile modern JavaScript into a form that's runnable on older JavaScript implementations.

Chapter 3, *Exploring Node.js Modules*, delves into the module as the unit of modularity in Node.js applications. We will dive deep into understanding and developing Node.js modules and using `npm` to maintain dependencies. We will learn about the new module format, ES6 modules, and how to use it in Node.js now that it is natively supported.

Chapter 4, *HTTP Servers and Clients*, starts exploring web development with Node.js. We will develop several small webserver and client applications in Node.js. We will use the Fibonacci algorithm to explore the effects of heavy-weight, long-running computations on a Node.js application. We will also learn several mitigation strategies and get our first experience with developing REST services.

Chapter 5, *Your First Express Application*, begins the main journey of this book, which is developing an application for creating and editing notes. In this chapter, we get a basic notes application running and get started with the Express framework.

Chapter 6, *Implementing the Mobile-First Paradigm*, uses the Bootstrap V4 framework to implement responsive web design in the notes application. This includes integrating a popular icon set and the steps required to customize Bootstrap.

Chapter 7, *Data Storage and Retrieval*, explores several database engines and a method to easily switch between databases at will. The goal is to robustly persist data to disk.

Chapter 8, *Authenticating Users with a Microservice*, adds user authentication to the notes application. We will learn about handling login and logout using PassportJS. Authentication is supported both for locally stored user credentials and for using OAuth with Twitter.

Chapter 9, *Dynamic Client/Server Interaction with Socket.IO*, looks at letting our users talk with each other in real time. We will use a popular framework for dynamic interaction between client and server, Socket.IO, to support dynamic updates of content and a simple commenting system. Everything is dynamically updated by users in pseudo-real time, giving us the opportunity to learn about real-time dynamic updating.

Chapter 10, *Deploying Node.js Applications to Linux Servers*, is where we begin the deployment journey. In this chapter, we will use the traditional methods of deploying background services on Ubuntu using Systemd.

Chapter 11, *Deploying Node.js Microservices with Docker*, sees us start to explore cloud-based deployment using Docker to treat the notes application as a cluster of microservices.

Chapter 12, *Deploying a Docker Swarm to AWS EC2 with Terraform*, literally takes us to the cloud by looking at building a cloud hosting system using AWS EC2 systems. We will use a popular tool, Terraform, to create and manage an EC2 cluster, and we will learn how to almost completely automate the deployment of a Docker Swarm cluster using Terraform features.

Chapter 13, *Unit Testing and Functional Testing*, has us explore three testing modes: unit testing, REST testing, and functional testing. We will use popular test frameworks, Mocha and Chai, to drive test cases in all three modes. For function testing, we will use Puppeteer, a popular framework for automating test execution in a Chrome instance.

Chapter 14, *Security in Node.js Applications*, is where we integrate security techniques and tools to mitigate security intrusions. We will start by implementing HTTPS on the AWS EC2 deployment using Let's Encrypt. We will then discuss several tools in Node.js to implement security settings and discuss the best security practices for both Docker and AWS environments.

To get the most out of this book

The basic requirement is installing Node.js and having a programmer-oriented text editor. The editor need not be anything fancy; even vi/vim will do in a pinch. We will show you how to install everything that's needed, and it's all open source, so there's no barrier to entry.

The most important tool is the one between your ears, and we aren't referring to ear wax.

Software/hardware covered in the book	OS requirements
Node.js and related frameworks such as Express, Sequelize, and Socket.IO	Any
The npm/yarn package management tools	Any
Python and C/C++ compilers	Any
MySQL, SQLite3, and MongoDB databases	Any
Docker	Any
Multipass	Any
Terraform	Any
Mocha and Chai	Any

Every piece of software concerned is readily available. For C/C++ compilers on Windows and macOS, you will need to get either Visual Studio (Windows) or Xcode (macOS), but both are freely available.

It will be helpful to have some experience with JavaScript programming. It is a fairly easy language to learn if you are already experienced with other programming languages.

Download the example code files

While we aim to have identical code snippets in the book and in the repository, there are going to be minor differences in some places. The repository may contain comments, debugging statements, or alternate implementations (commented-out) that are not shown in the book.

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 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 onscreen 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 Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Node.js-Web-Development-Fifth-Edition>. 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!

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Start by changing `package.json` to have the following `scripts` section."

A block of code is set as follows:

```
function readFile(filename) {  
    return new Promise((resolve, reject) => {  
        fs.readFile(filename, (err, data) => {  
            if (err) reject(err);  
            else resolve(data);  
        });  
    });  
}
```

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

```
function asyncFunction(arg1, arg2) {  
    return new Promise((resolve, reject) => {  
        // perform some task or computation that's asynchronous  
        // for any error detected:  
        if (errorDetected) return reject(dataAboutError);  
        // When the task is finished  
        resolve(theResult);  
    });  
};
```

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

```
$ mkdir notes  
$ cd notes
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click on the **Submit** button."

Warnings or important notes appear like this.



Tips and tricks appear like this.



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 customercare@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 would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering 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 would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit 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

Section 1: Introduction to Node.js

This is a high-level overview of the Node.js landscape. The reader will have taken the first steps in using Node.js.

This section comprises the following chapters:

- Chapter 1, *About Node.js*
- Chapter 2, *Setting Up Node.js*
- Chapter 3, *Exploring Node.js Modules*
- Chapter 4, *HTTP Servers and Clients*

1

About Node.js

JavaScript is at the fingertips of every frontend web developer, making it a very popular programming language, so much so that it is stereotyped as being for client-side code in web pages. The chances are that, having picked up this book, you've heard of Node.js, a programming platform for coding in JavaScript outside web browsers. Now about ten years old, Node.js is becoming a mature programming platform that's widely used in projects both big and small.

This book will give you an introduction to Node.js. By the end of this book, you will have learned about the complete lifecycle of developing server-side web applications using Node.js, from concept to deployment and security. In writing this book, we have presumed the following:

- You already know how to write software.
- You are familiar with JavaScript.
- You know something about developing web applications in other languages.

When we evaluate a new programming tool, do we latch on because it's the popular new tool? Maybe some of us do that, but the mature approach is to weigh one tool against another. That's what this chapter is about, presenting the technical rationale for using Node.js. Before getting to the code, we must consider what Node.js is and how it fits in the overall marketplace of software development tools. Then we will dive right into developing working applications and recognize that often the best way to learn is by rummaging around in working code.

We will cover the following topics in this chapter:

- An introduction to Node.js
- What you can do with Node.js
- Why you should use Node.js
- The architecture of Node.js

- Performance, utilization, and scalability with Node.js
- Node.js, microservice architecture, and testing
- Implementing the twelve-factor app model with Node.js

Overview of Node.js

Node.js is an exciting new platform for developing web applications, application servers, any sort of network server or client, and general-purpose programming. It is designed for extreme scalability in networked applications through an ingenious combination of server-side JavaScript, asynchronous I/O, and asynchronous programming.

While only ten years old, Node.js has quickly grown in prominence and is now playing a significant role. Companies, both large and small, are using it for large-scale and small-scale projects. PayPal, for example, has converted many services from Java to Node.js.

The Node.js architecture departs from a typical choice made by other application platforms. Where threads are widely used to scale an application to fill the CPU, Node.js eschews threads because of their inherent complexity. It's claimed that with single-thread event-driven architectures, the memory footprint is low, throughput is high, the latency profile under load is better, and the programming model is simpler. The Node.js platform is in a phase of rapid growth, and many see it as a compelling alternative to the traditional web application architectures using Java, PHP, Python, or Ruby on Rails.

At its heart, it is a standalone JavaScript engine with extensions that is suitable for general-purpose programming and that has a clear focus on application server development. Even though we're comparing Node.js to application-server platforms, it is not an application server. Instead, Node.js is a programming runtime akin to Python, Go, or Java SE. While there are web application frameworks and application servers written in Node.js, it is simply a system to execute JavaScript programs.

The key architectural choice is that Node.js is event-driven, rather than multithreaded. The Node.js architecture rests on dispatching blocking operations to a single-threaded event loop, with results arriving back to the caller as an event that invokes an event handler function. In most cases, the event is converted into a promise that is handled by an `async` function. Because Node.js is based on Chrome's V8 JavaScript engine, the performance and feature improvements implemented in Chrome quickly flow through to the Node.js platform.

The Node.js core modules are general enough to implement any sort of server that is executing any TCP or UDP protocol, whether it's a **Domain Name System (DNS)**, **HTTP**, **internet relay chat (IRC)**, or **FTP**. While it supports the development of internet servers or clients, its biggest use case is regular website development, in place of technology such as an Apache/PHP or Rails stack, or to complement existing websites—for example, adding real-time chat or monitoring existing websites can easily be done with the **Socket.IO** library for Node.js. Its lightweight, high-performance nature often sees Node.js used as a **glue** service.

A particularly intriguing combination is the deployment of small services on modern cloud infrastructure using tools such as Docker and Kubernetes, or function-as-a-service platforms, such as AWS Lambda. Node.js works well when dividing a large application into easily deployable microservices at scale.

With a high-level understanding of Node.js under our belt, let's dig a little deeper.

The capabilities of Node.js

Node.js is a platform for writing JavaScript applications outside web browsers. This is not the JavaScript environment we are familiar with in web browsers! While Node.js executes the same JavaScript language that we use in browsers, it doesn't have some of the features associated with the browser. For example, there is no HTML DOM built into Node.js.

Beyond its native ability to execute JavaScript, the built-in modules provide capabilities of the following sort:

- Command-line tools (in shell script style)
- An interactive-terminal style of program—that is, a **read-eval-print loop (REPL)**
- Excellent process control functions to oversee child processes
- A buffer object to deal with binary data
- TCP or UDP sockets with comprehensive, event-driven callbacks
- DNS lookup
- An HTTP, HTTPS, and HTTP/2-client server layered on top of the TCP library filesystem access
- Built-in rudimentary unit testing support through assertions

The network layer of Node.js is low level while being simple to use—for example, the HTTP modules allow you to write an HTTP server (or client) using a few lines of code. This is powerful, but it puts you, the programmer, very close to the protocol requests and makes you implement precisely those HTTP headers that you should return in request responses.

Typical web-application developers don't need to work at a low level of the HTTP or other protocols; instead, we tend to be more productive working with higher-level interfaces—for example, PHP coders assume that Apache/Nginx/and so on are already there providing the HTTP, and that they don't have to implement the HTTP server portion of the stack. By contrast, a Node.js programmer does implement an HTTP server, to which their application code is attached.

To simplify the situation, the Node.js community has several web application frameworks, such as Express, providing the higher-level interfaces required by typical programmers. You can quickly configure an HTTP server with baked-in capabilities, such as sessions, cookies, serving static files, and logging, letting developers focus on their business logic. Other frameworks provide OAuth 2 support or focus on REST APIs, and so on.

The community of folks using Node.js has built an amazing variety of things on this foundation.

What are folks doing with Node.js?

Node.js is not limited to web service application development; the community around Node.js has taken it in many other directions:

- **Build tools:** Node.js has become a popular choice for developing command-line tools that are used in software development or communicating with service infrastructure. Grunt, Gulp, and Webpack are widely used by frontend developers to build assets for websites. Babel is widely used for transpiling modern ES-2016 code to run on older browsers. Popular CSS optimizers and processors, such as PostCSS, are written in Node.js. static website generation systems, such as Metalsmith, Punch, and AkashaCMS, run at the command line, and generate website content that you upload to a web server.
- **Web UI testing:** Puppeteer gives you control over a headless Chrome web-browser instance. With it, you can develop Node.js scripts by controlling a modern, full-featured web browser. Some typical use cases are web scraping and web application testing.

- **Desktop applications:** Both Electron and **node-webkit (NW.js)** are frameworks for developing desktop applications for Windows, macOS, and Linux. These frameworks utilize a large chunk of Chrome, wrapped by Node.js libraries, to develop desktop applications using web UI technologies. Applications are written with modern HTML5, CSS3, and JavaScript, and can utilize leading-edge web frameworks, such as Bootstrap, React, VueJS, and AngularJS. Many popular applications have been built using Electron, including the Slack desktop client application, the Atom, Microsoft Visual Code programming editors, the Postman REST client, the GitKraken GIT client, and Etcher, which makes it incredibly easy to burn OS images to flash drives to run on single-board computers.
- **Mobile applications:** The Node.js for Mobile Systems project lets you develop smartphone or tablet computer applications using Node.js for both iOS and Android. Apple's App Store rules preclude incorporating a JavaScript engine with JIT capabilities, meaning that normal Node.js cannot be used in an iOS application. For iOS application development, the project uses Node.js-on-ChakraCore to skirt around the App Store rules. For Android application development, the project uses regular Node.js on Android. At the time of writing, the project is in an early stage of development, but it looks promising.
- **Internet of things (IoT):** Node.js is a very popular language for Internet-of-Things projects, and Node.js runs on most ARM-based, single-board computers. The clearest example is the NodeRED project. It offers a graphical programming environment, letting you draw programs by connecting blocks together. It features hardware-oriented input and output mechanisms—for example, to interact with **General Purpose I/O (GPIO)** pins on Raspberry Pi or Beaglebone single-board computers.

You may already be using Node.js applications without realizing it! JavaScript has a place outside the web browser, and it's not just thanks to Node.js.

Server-side JavaScript

Quit scratching your head, already! Of course, you're doing it, scratching your head and mumbling to yourself, "What's a browser language doing on the server?" In truth, JavaScript has a long and largely unknown history outside the browser. JavaScript is a programming language, just like any other language, and the better question to ask is "Why should JavaScript remain trapped inside web browsers?"

Back in the dawn of the web age, the tools for writing web applications were at a fledgling stage. Some developers were experimenting with Perl or TCL to write CGI scripts, and the PHP and Java languages had just been developed. Even then, JavaScript saw use on the server side. One early web application server was Netscape's LiveWire server, which used JavaScript. Some versions of Microsoft's ASP used JScript, their version of JavaScript. A more recent server-side JavaScript project is the RingoJS application framework in the Java universe. Java 6 and Java 7 were both shipped with the Rhino JavaScript engine. In Java 8, Rhino was dropped in favor of the newer Nashorn JavaScript engine.

In other words, JavaScript outside the browser is not a new thing, even if it is uncommon.

You have learned that Node.js is a platform for writing JavaScript applications outside of web browsers. The Node.js community uses this platform for a huge array of application types, far more than was originally conceived for the platform. This proves that Node.js is popular, but we must still consider the technical rationale for using it.

Why should you use Node.js?

Of the many available web-application development platforms, why should you choose Node.js? There are many stacks to choose from; what is it about Node.js that makes it rise above the others? We will learn the answer to this in the following sections.

Popularity

Node.js is quickly becoming a popular development platform, and is being adopted by plenty of big and small players. One of these players is PayPal, who are replacing their incumbent Java-based system with one written in Node.js. Other large Node.js adopters include Walmart's online e-commerce platform, LinkedIn, and eBay.



For PayPal's blog post about this, visit
<https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/>.

According to NodeSource, Node.js usage is growing rapidly (for more information, visit <https://nodesource.com/node-by-numbers>). The evidence for this growth includes increasing bandwidth for downloading Node.js releases, increasing activity in Node.js-related GitHub projects, and more.

Interest in JavaScript itself remains very strong but has been at a plateau for years, measured in search volume (Google Insights) and its use as a programming skill (Dice Skills Center). Node.js interest has been growing rapidly, but is showing signs of plateauing.



For more on this, see <https://itnext.io/choosing-typescript-vs-javascript-technology-popularity-ea978af6b5f> or <http://bit.ly/2q5cu0w>.

It's best to not just follow the crowd because there are different crowds, and each one claims that their software platform does cool things. Node.js does some cool things, but what is more important is its technical merit.

JavaScript everywhere

Having the same programming language on the server and client has been a long-time dream on the web. This dream dates back to the early days of Java, where Java applets in the browser were to be the frontend to server applications written in Java, and JavaScript was originally envisioned as a lightweight scripting language for those applets. Java never fulfilled its hype as a client-side programming language, and even the phrase "Java Applets" is fading into a dim memory of the abandoned client-side application model. We ended up with JavaScript as the principle in-browser, client-side language, rather than Java. Typically, the frontend JavaScript developers were in a different language universe than the server-side team, which was likely to be coding in PHP, Java, Ruby, or Python.

Over time, in-browser JavaScript engines became incredibly powerful, letting us write ever-more-complex browser-side applications. With Node.js, we are finally able to implement applications with the same programming language on the client and server by having JavaScript at both ends of the web, in the browser and server.

A common language for frontend and backend offers several potential benefits:

- The same programming staff can work on both ends of the wire.
- Code can be migrated between the server and client more easily.
- Common data formats (JSON) between the server and client.

- Common software tools exist for the server and client.
- Common testing or quality-reporting tools for the server and client.
- When writing web applications, view templates can be used on both sides.

The JavaScript language is very popular because of its ubiquity in web browsers. It compares favorably with other languages while having many modern, advanced language concepts. Thanks to its popularity, there is a deep talent pool of experienced JavaScript programmers out there.

Leveraging Google's investment in V8

To make Chrome a popular and excellent web browser, Google invested in making V8 a super-fast JavaScript engine. Google, therefore, has a huge motivation to keep on improving V8. V8 is the JavaScript engine for Chrome, and it can also be executed in a standalone manner.

Node.js is built on top of the V8 JavaScript engine, letting it take advantage of all that work on V8. As a result, Node.js was able to quickly adopt new JavaScript language features as they were implemented by V8 and reap performance wins for the same reason.

Leaner, asynchronous, event-driven model

The Node.js architecture, built on a single execution thread, with an ingenious event-oriented, asynchronous-programming model, and a fast JavaScript engine, is claimed to have less overhead than thread-based architectures. Other systems using threads for concurrency tend to have memory overhead and complexity, which Node.js does not have. We'll get into this more later in the chapter.

Microservice architecture

A new sensation in software development is the idea of the microservice. Microservices are focused on splitting a large web application into small, tightly-focused services that can be easily developed by small teams. While they aren't exactly a new idea—they're more of a reframing of old client-server computing models—the microservice pattern fits well with agile project-management techniques, and gives us a more granular application deployment.

Node.js is an excellent platform for implementing microservices. We'll get into this later.

Node.js is stronger after a major schism and hostile fork

During 2014 and 2015, the Node.js community faced a major split over policy, direction, and control. The [io.js](#) project was a hostile fork driven by a group that wanted to incorporate several features and change who was in the decision-making process. The end result was a merge of the Node.js and io.js repositories, an independent Node.js foundation to run the show, and the community working together to move forward in a common direction.

A concrete result of healing that rift is the rapid adoption of new ECMAScript language features. The V8 engine is adopting these new features quickly to advance the state of web development. The Node.js team, in turn, is adopting these features as quickly as they show up in V8, meaning that promises and `async` functions are quickly becoming a reality for Node.js programmers.

The bottom line is that the Node.js community not only survived the io.js fork and the later ayo.js fork, but the community and the platform it nurtured grew stronger as a result.

In this section, you have learned several reasons to use Node.js. Not only is it a popular platform, with a strong community behind it, but there are also serious technical reasons to use it. Its architecture has some key technical benefits, so let's take a deeper look at these.

The Node.js event-driven architecture

Node.js's blistering performance is said to be because of its asynchronous event-driven architecture and its use of the V8 JavaScript engine. This enables it to handle multiple tasks concurrently, such as juggling between requests from multiple web browsers. The original creator of Node.js, Ryan Dahl, followed these key points:

- A single-thread, event-driven programming model is simpler to code and has less complexity and less overhead than application servers that rely on threads to handle multiple concurrent tasks.

- By converting blocking function calls into asynchronous code execution, you can configure the systems so that it issues an event when the blocking request is satisfied.
- You can leverage the V8 JavaScript engine from the Chrome browser, and all the work goes into improving V8; all the performance enhancements going into V8, therefore, benefits Node.js.

In most application servers, concurrency, or the ability to handle multiple concurrent requests, is implemented with a multithreaded architecture. In such a system, any request for data, or any other blocking function call, causes the current execution thread to suspend and wait for the result. Handling concurrent requests requires there to be multiple execution threads. When one thread is suspended, another thread can execute. This causes churn as the application server starts and stops the threads to handle requests. Each suspended thread (typically waiting on an input/output operation to finish) consumes a full call stack of memory, adding to overhead. Threads add complexity to the application server as well as server overhead.

To help us wrap our heads around why this would be, Ryan Dahl, the creator of Node.js, offered the following example. In his Cinco de NodeJS presentation in May 2010 (<https://www.youtube.com/watch?v=M-sc73Y-zQA>) Dahl asked us what happens when we execute a line of code such as this:

```
result = query('SELECT * from db.table');
// operate on the result
```

Of course, the program pauses at this point while the database layer sends the query to the database and waits for the result or the error. This is an example of a blocking function call. Depending on the query, this pause can be quite long (well, a few milliseconds, which is ages in computer time). This pause is bad because the execution thread can do nothing while it waits for the result to arrive. If your software is running on a single-threaded platform, the entire server would be blocked and unresponsive. If instead your application is running on a thread-based server platform, a thread-context switch is required to satisfy any other requests that arrive. The greater the number of outstanding connections to the server, the greater the number of thread-context switches. Context switching is not free because more threads require more memory per thread state and more time for the CPU to spend on thread management overheads.

The key inspiration guiding the original development of Node.js was the simplicity of a single-threaded system. A single execution thread means that the server doesn't have the complexity of multithreaded systems. This choice meant that Node.js required an event-driven model for handling concurrent tasks. Instead of the code waiting for results from a blocking request, such as retrieving data from a database, an event is instead dispatched to an event handler.

Using threads to implement concurrency often comes with admonitions, such as *expensive and error-prone, the error-prone synchronization primitives of Java, or designing concurrent software can be complex and error-prone*. The complexity comes from access to shared variables and various strategies to avoid deadlock and competition between threads. The *synchronization primitives of Java* are an example of such a strategy, and obviously many programmers find them difficult to use. There's a tendency to create frameworks such as `java.util.concurrent` to tame the complexity of threaded concurrency, but some argue that papering over complexity only makes things more complex.

A typical Java programmer might object at this point. Perhaps their application code is written against a framework such as Spring, or maybe they're directly using Java EE. In either case, their application code does not use concurrency features or deal with threads, and therefore where is the complexity that we just described? Just because that complexity is hidden within Spring and Java EE does not mean that there is no complexity and overhead.

Okay, we get it: while multithreaded systems can do amazing things, there is inherent complexity. What does Node.js offer?

The Node.js answer to complexity

Node.js asks us to think differently about concurrency. Callbacks fired asynchronously from an event loop are a much simpler concurrency model—simpler to understand, simpler to implement, simpler to reason about, and simpler to debug and maintain.

Node.js has a single execution thread with no waiting on I/O or context switching. Instead, there is an event loop that dispatches events to handler functions as things happen. A request that would have blocked the execution thread instead executes asynchronously, with the results or errors triggering an event. Any operation that would block or otherwise take time to complete must use the asynchronous model.

The original Node.js paradigm delivered the dispatched event to an anonymous function. Now that JavaScript has `async` functions, the Node.js paradigm is shifting to deliver results and errors via a promise that is handled by the `await` keyword. When an asynchronous function is called, control quickly passes to the event loop rather than causing Node.js to block. The event loop continues handling the variety of events while recording where to send each result or error.

By using an asynchronous event-driven I/O, Node.js removes most of this overhead while introducing very little of its own.

One of the points Ryan Dahl made in the Cinco de Node presentation is a hierarchy of execution time for different requests. Objects in memory are more quickly accessed (in the order of nanoseconds) than objects on disk or objects retrieved over the network (milliseconds or seconds). The longer access time for external objects is measured in millions of clock cycles, which can be an eternity when your customer is sitting at their web browser ready to move on if it takes longer than two seconds to load the page.

Therefore, concurrent request handling means using a strategy to handle the requests that take longer to satisfy. If the goal is to avoid the complexity of a multithreaded system, then the system must use asynchronous operations as Node.js does.

What do these asynchronous function calls look like?

Asynchronous requests in Node.js

In Node.js, the query that we looked at previously will read as follows:

```
query('SELECT * from db.table', function (err, result) {  
    if (err) throw err; // handle errors  
    // operate on result  
});
```

The programmer supplies a function that is called (hence the name *callback function*) when the result (or error) is available. The `query` function still takes the same amount of time. Instead of blocking the execution thread, it returns to the event loop, which is then free to handle other requests. The Node.js will eventually fire an event that causes this callback function to be called with the result or error indication.

A similar paradigm is used in client-side JavaScript, where we write event handler functions all the time.

Advances in the JavaScript language have given us new options. When used with ES2015 promises, the equivalent code would look like this:

```
query('SELECT * from db.table')
  .then(result => {
    // operate on result
  })
  .catch(err => {
    // handle errors
  });

```

This is a little better, especially in instances of deeply nested event handling.

The big advance came with the ES-2017 `async` function:

```
try {
  const result = await query('SELECT * from db.table');
  // operate on result
} catch (err) {
  // handle errors
}
```

Other than the `async` and `await` keywords, this looks like code we'd write in other languages, and is much easier to read. Because of what `await` does, it is still asynchronous code execution.

All three of these code snippets perform the same query that we wrote earlier. Instead of `query` being a blocking function call, it is asynchronous and does not block the execution thread.

With both the callback functions and the promise's asynchronous coding, Node.js had its own complexity issue. Oftentimes, we call one asynchronous function after another. With callback functions, that meant deeply nested callback functions, and with promises, that meant a long chain of `.then` handler functions. In addition to the complexity of the coding, we have errors and results landing in unnatural places. Instead of landing on the next line of code, the asynchronously executed callback function is invoked. The order of execution is not one line after another, as it is in synchronous programming languages; instead, the order of execution is determined by the order of the callback function execution.

The `async` function approach solves that coding complexity. The coding style is more natural since the results and errors land in the natural place, at the next line of code. The `await` keyword integrates asynchronous result handling without blocking the execution thread. A lot is buried under the covers of the `async/await` feature, and we'll be covering this model extensively throughout this book.

But does the asynchronous architecture of Node.js actually improve performance?

Performance and utilization

Some of the excitement over Node.js is due to its throughput (the requests per second that it can serve). Comparative benchmarks of similar applications—for example, Apache—show that Node.js has tremendous performance gains.

One benchmark going around is the following simple HTTP server (borrowed from <https://nodejs.org/en/>), which simply returns a Hello World message directly from memory:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

This is one of the simpler web servers that you can build with Node.js. The `http` object encapsulates the HTTP protocol, and its `http.createServer` method creates a whole web server, listening on the port specified in the `listen` method. Every request (whether a `GET` or `POST` on any URL) on that web server calls the provided function. It is very simple and lightweight. In this case, regardless of the URL, it returns a simple `text/plain` that is the Hello World response.

Ryan Dahl showed a simple benchmark in a video titled *Ryan Dahl: Introduction to Node.js* (on the YUI Library channel on YouTube, <https://www.youtube.com/watch?v=M-sc73Y-zQA>). It used a similar HTTP server to this, but that returned a one-megabyte binary buffer; Node.js gave 822 req/sec, while Nginx gave 708 req/sec, for a 15% improvement over Nginx. He also noted that Nginx peaked at four megabytes of memory, while Node.js peaked at 64 megabytes.

The key observation was that Node.js, running an interpreted, JIT-compiled, high-level language, was about as fast as Nginx, built of highly optimized C code, while running similar tasks. That presentation was in May 2010, and Node.js has improved hugely since then, as shown in Chris Bailey's talk that we referenced earlier.

Yahoo! search engineer Fabian Frank published a performance case study of a real-world search query suggestion widget implemented with Apache/PHP and two variants of Node.js stacks

(<http://www.slideshare.net/FabianFrankDe/nodejs-performance-case-study>).

The application is a pop-up panel showing search suggestions as the user types in phrases using a JSON-based HTTP query. The Node.js version could handle eight times the number of requests per second with the same request latency. Fabian Frank said both Node.js stacks scaled linearly until CPU usage hit 100%.

LinkedIn did a massive overhaul of their mobile app using Node.js for the server-side to replace an old Ruby on Rails app. The switch lets them move from 30 servers down to 3, and allowed them to merge the frontend and backend team because everything was written in JavaScript. Before choosing Node.js, they'd evaluated Rails with Event Machine, Python with Twisted, and Node.js, chose Node.js for the reasons that we just discussed. For a look at what LinkedIn did, see

<http://arstechnica.com/information-technology/2012/10/a-behind-the-scenes-look-at-linkedin-s-mobile-engineering/>.

Most existing Node.js performance tips tend to have been written for older V8 versions that used the CrankShaft optimizer. The V8 team has completely dumped CrankShaft, and it has a new optimizer called TurboFan—for example, under CrankShaft, it was slower to use `try/catch`, `let/const`, generator functions, and so on. Therefore, common wisdom said to not use those features, which is depressing because we want to use the new JavaScript features because of how much it has improved the JavaScript language. Peter Marshall, an engineer on the V8 team at Google, gave a talk at Node.js Interactive 2017 claiming that, using TurboFan, you should just write natural JavaScript. With TurboFan, the goal is for across-the-board performance improvements in V8. To view the presentation, see the video titled *High Performance JS in V8* at <https://www.youtube.com/watch?v=YqOhBezMx1o>.

A truism about JavaScript is that it's no good for heavy computation work because of the nature of JavaScript. We'll go over some ideas that are related to this in the next section. A talk by Mikola Lysenko at Node.js Interactive 2016 went over some issues with numerical computing in JavaScript, and some possible solutions. Common numerical computing involves large numerical arrays processed by numerical algorithms that you might have learned in calculus or linear algebra classes. What JavaScript lacks is multidimensional arrays and access to certain CPU instructions. The solution that he presented is a library to implement multidimensional arrays in JavaScript, along with another library full of numerical computing algorithms. To view the presentation, see the video titled *Numerical Computing in JavaScript* by Mikola Lysenko at <https://www.youtube.com/watch?v=10RaKEzlnys>.

At the Node.js Interactive conference in 2017, IBM's Chris Bailey made a case for Node.js being an excellent choice for highly scalable microservices. Key performance characteristics are I/O performance (measured in transactions per second), startup time (because that limits how quickly your service can scale up to meet demand), and memory footprint (because that determines how many application instances can be deployed per server). Node.js excels on all those measures; with every subsequent release, it either improves on each measure or remains fairly steady. Bailey presented figures comparing Node.js to a similar benchmark written in Spring Boot showing Node.js to perform much better. To view his talk, see the video titled *Node.js Performance and Highly Scalable Micro-Services - Chris Bailey, IBM* at <https://www.youtube.com/watch?v=Fbhfc4jtGW4>.

The bottom line is that Node.js excels at event-driven I/O throughput. Whether a Node.js program can excel at computational programs depends on your ingenuity in working around some limitations in the JavaScript language.

A big problem with computational programming is that it prevents the event loop from executing. As we will see in the next section, that can make Node.js look like a poor candidate for anything.

Is Node.js a cancerous scalability disaster?

In October 2011, a blog post (since pulled from the blog where it was published) titled *Node.js is a cancer* called Node.js a scalability disaster. The example shown for proof was a CPU-bound implementation of the Fibonacci sequence algorithm. While the argument was flawed—since nobody implements Fibonacci that way—it made the valid point that Node.js application developers have to consider the following: where do you put the heavy computational tasks?

A key to maintaining high throughput of Node.js applications is by ensuring that events are handled quickly. Because it uses a single execution thread, if that thread is bogged down with a big calculation, Node.js cannot handle events, and event throughput will suffer.

The Fibonacci sequence, serving as a stand-in for heavy computational tasks, quickly becomes computationally expensive to calculate for a naïve implementation such as this:

```
const fibonacci = exports.fibonacci = function(n) {  
    if (n === 1 || n === 2) {  
        return 1;  
    } else {  
        return fibonacci(n-1) + fibonacci(n-2);  
    }  
};
```

```
    }  
}
```

This is a particularly simplistic approach to calculating Fibonacci numbers. Yes, there are many ways to calculate Fibonacci numbers more quickly. We are showing this as a general example of what happens to Node.js when event handlers are slow and not to debate the best ways to calculate mathematical functions. Consider the following server:

```
const http = require('http');  
const url  = require('url');  
  
http.createServer(function (req, res) {  
  const urlP = url.parse(req.url, true);  
  let fibo;  
  res.writeHead(200, {'Content-Type': 'text/plain'});  
  if (urlP.query['n']) {  
    fibo = fibonacci(urlP.query['n']); // Blocking  
    res.end('Fibonacci ' + urlP.query['n'] + '=' + fibo);  
  } else {  
    res.end('USAGE: http://127.0.0.1:8124?n=# where ##  
           is the Fibonacci number desired');  
  }  
}).listen(8124, '127.0.0.1');  
console.log('Server running at http://127.0.0.1:8124');
```

This is an extension of the simple web server shown earlier. It looks in the request URL for an argument, n, for which to calculate the Fibonacci number. When it's calculated, the result is returned to the caller.

For sufficiently large values of n (for example, 40), the server becomes completely unresponsive because the event loop is not running. Instead, this function has blocked event processing because the event loop cannot dispatch events while the function is grinding through the calculation.

In other words, the Fibonacci function is a stand-in for any blocking operation.

Does this mean that Node.js is a flawed platform? No, it just means that the programmer must take care to identify code with long-running computations and develop solutions. These include rewriting the algorithm to work with the event loop, rewriting the algorithm for efficiency, integrating a native code library, or foisting computationally expensive calculations to a backend server.

A simple rewrite dispatches the computations through the event loop, letting the server continue to handle requests on the event loop. Using callbacks and closures (anonymous functions), we're able to maintain asynchronous I/O and concurrency promises, as shown in the following code:

```
const fibonacciAsync = function(n, done) {
    if (n === 0) {
        return 0;
    } else if (n === 1 || n === 2) {
        done(1);
    } else if (n === 3) {
        return 2;
    } else {
        process.nextTick(function() {
            fibonacciAsync(n-1, function(val1) {
                process.nextTick(function() {
                    fibonacciAsync(n-2, function(val2) {
                        done(val1+val2); });
                });
            });
        });
    }
}
```

This is an equally silly way to calculate Fibonacci numbers, but by using `process.nextTick`, the event loop has an opportunity to execute.

Because this is an asynchronous function that takes a callback function, it necessitates a small refactoring of the server:

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

http.createServer(function (req, res) {
    let urlP = url.parse(req.url, true);
    res.writeHead(200, {'Content-Type': 'text/plain'});
    if (urlP.query['n']) {
        fibonacciAsync(urlP.query['n'], fibo => { // Asynchronous
            res.end('Fibonacci ' + urlP.query['n'] + '=' + fibo);
        });
    } else {
        res.end('USAGE: http://127.0.0.1:8124?n=## where ## is the
               Fibonacci number desired');
    }
}).listen(8124, '127.0.0.1'); console.log('Server running at
http://127.0.0.1:8124');
```

We've added a callback function to receive the result. In this case, the server is able to handle multiple Fibonacci number requests. But there is still a performance issue because of the inefficient algorithm.

Later in this book, we'll explore this example a little more deeply to explore alternative approaches.

In the meantime, we can discuss why it's important to use efficient software stacks.

Server utilization, overhead costs, and environmental impact

The striving for optimal efficiency (handling more requests per second) is not just about the geeky satisfaction that comes from optimization. There are real business and environmental benefits. Handling more requests per second, as Node.js servers can do, means the difference between buying lots of servers and buying only a few servers. Node.js potentially lets your organization do more with less.

Roughly speaking, the more servers you buy, the greater the monetary cost and the greater the environmental cost. There's a whole field of expertise around reducing costs and the environmental impact of running web-server facilities to which that rough guideline doesn't do justice. The goal is fairly obvious—fewer servers, lower costs, and a lower environmental impact by using more efficient software.

Intel's paper, *Increasing Data Center Efficiency with Server Power Measurements* (<https://www.intel.com/content/dam/doc/white-paper/intel-it-data-center-efficiency-server-power-paper.pdf>), gives an objective framework for understanding efficiency and data center costs. There are many factors, such as buildings, cooling systems, and computer system designs. Efficient building design, efficient cooling systems, and efficient computer systems (data center efficiency, data center density, and storage density) can lower costs and environmental impact. But you can destroy these gains by deploying an inefficient software stack, compelling you to buy more servers than you would if you had an efficient software stack. Alternatively, you can amplify gains from data center efficiency with an efficient software stack that lets you decrease the number of servers required.

This talk about efficient software stacks isn't just for altruistic environmental purposes. This is one of those cases where being green can help your business bottom line.

In this section, we have learned a lot about how Node.js architecture differs from other programming platforms. The choice to eschew threads to implement concurrency simplifies away the complexity and overhead that comes from using threads. This seems to have fulfilled the promise of being more efficient. Efficiency has a number of benefits to many aspects of a business.

Embracing advances in the JavaScript language

The last couple of years have been an exciting time for JavaScript programmers. The TC-39 committee that oversees the ECMAScript standard has added many new features, some of which are syntactic sugar, but several of which have propelled us into a whole new era of JavaScript programming. By itself, the `async/await` feature promises us a way out of what's called callback hell, the situation that we find ourselves in when nesting callbacks within callbacks. It's such an important feature that it should necessitate a broad rethinking of the prevailing callback-oriented paradigm in Node.js and the rest of the JavaScript ecosystem.

A few pages ago, you saw this:

```
query('SELECT * from db.table', function (err, result) {  
    if (err) throw err; // handle errors  
    // operate on result  
});
```

This was an important insight on Ryan Dahl's part, and is what propelled Node.js's popularity. Certain actions take a long time to run, such as database queries, and should not be treated the same as operations that quickly retrieve data from memory. Because of the nature of the JavaScript language, Node.js had to express this asynchronous coding construct in an unnatural way. The results do not appear at the next line of code, but instead appear within this callback function. Furthermore, errors have to be handled in an unnatural way, inside that callback function.

The convention in Node.js is that the first parameter to a callback function is an error indicator and the subsequent parameters are the results. This is a useful convention that you'll find all across the Node.js landscape; however, it complicates working with results and errors because both land in an inconvenient location—that callback function. The natural place for errors and results to land is on the subsequent line(s) of code.

We descend further into callback hell with each layer of callback function nesting. The seventh layer of callback nesting is more complex than the sixth layer of callback nesting. Why? If nothing else, it's because the special considerations for error handling become ever more complex as callbacks are nested more deeply.

But as we saw earlier, this is the new preferred way to write asynchronous code in Node.js:

```
const results = await query('SELECT * from db.table');
```

Instead, ES2017 `async` functions return us to this very natural expression of programming intent. Results and errors land in the correct location while preserving the excellent event-driven asynchronous programming model that made Node.js great. We'll see how this works later in the book.

The TC-39 committee added many more new features to JavaScript, such as the following:

- An improved syntax for class declarations, making object inheritance and getter/setter functions very natural.
- A new module format that is standardized across browsers and Node.js.
- New methods for strings, such as the template string notation.
- New methods for collections and arrays—for example, operations for `map/reduce/filter`.
- The `const` keyword to define variables that cannot be changed and the `let` keyword to define variables whose scope is limited to the block in which they're declared, rather than hoisted to the front of the function.
- New looping constructs and an iteration protocol that works with those new loops.
- A new kind of function, the arrow function, which is lighter in weight, meaning less memory and execution time impact.
- The `Promise` object represents a result that is promised to be delivered in the future. By themselves, promises can mitigate the callback hell problem, and they form part of the basis for `async` functions.
- Generator functions are an intriguing way to represent asynchronous iteration over a set of values. More importantly, they form the other half of the basis for `async` functions.

You may see the new JavaScript described as ES6 or ES2017. What's the preferred name to describe the version of JavaScript that is being used?

ES1 through ES5 marked various phases of JavaScript's development. ES5 was released in 2009 and is widely implemented in modern browsers. Starting with ES6, the TC-39 committee decided to change the naming convention because of their intention to add new language features every year. Therefore, the language version name now includes the year—for example, ES2015 was released in 2015, ES2016 was released in 2016, and ES2017 was released in 2017.

Deploying ES2015/2016/2017/2018 JavaScript code

The elephant in the room is that often JavaScript developers are unable to use the latest features. Frontend JavaScript developers are limited by the deployed web browsers and the large number of old browsers in use on machines whose OS hasn't been updated for years. Internet Explorer version 6 has fortunately been almost completely retired, but there are still plenty of old browsers installed on older computers that are still serving a valid role for their owners. Old browsers mean old JavaScript implementations, and if we want our code to work, we need it to be compatible with old browsers.

One of the uses for Babel and other code-rewriting tools is to deal with this issue. Many products must be usable by folks using an old browser. Developers can still write their code with the latest JavaScript or TypeScript features, then use Babel to rewrite their code so that it runs on the old browser. This way, frontend JavaScript programmers can adopt (some of) the new features at the cost of a more complex build toolchain and the risk of bugs being introduced by the code-rewriting process.

The Node.js world doesn't have this problem. Node.js has rapidly adopted ES2015/2016/2017 features as quickly as they were implemented in the V8 engine. Starting with Node.js 8, we were able to freely use `async` functions as a native feature. The new module format was first supported in Node.js version 10.

In other words, while frontend JavaScript programmers can argue that they must wait a couple of years before adopting ES2015/2016/2017 features, Node.js programmers have no need to wait. We can simply use the new features without needing any code-rewriting tools, unless our managers insist on supporting older Node.js releases that predate the adoption of these features. In that case, it is recommended that you use Babel.

Some advances in the JavaScript world are happening outside the TC-39 community.

TypeScript and Node.js

The TypeScript language is an interesting offshoot of the JavaScript environment. Because JavaScript is increasingly able to be used for complex applications, it is increasingly useful for the compiler to help catch programming errors. Enterprise programmers in other languages, such as Java, are accustomed to strong type checking as a way of preventing certain classes of bugs.

Strong type checking is somewhat anathema to JavaScript programmers, but is demonstrably useful. The TypeScript project aims to bring enough rigor from languages such as Java and C# while leaving enough of the looseness that makes JavaScript so popular. The result is compile-time type checking without the heavy baggage carried by programmers in some other languages.

While we won't use TypeScript in this book, its toolchain is very easy to adopt in Node.js applications.

In this section, we've learned that as the JavaScript language changes, the Node.js platform has kept up with those changes.

Developing microservices or maxiservices with Node.js

New capabilities, such as cloud deployment systems and Docker, make it possible to implement a new kind of service architecture. Docker makes it possible to define server process configuration in a repeatable container that's easy to deploy by the millions into a cloud-hosting system. It lends itself best to small, single-purpose service instances that can be connected together to make a complete system. Docker isn't the only tool to help simplify cloud deployments; however, its features are well attuned to modern application deployment needs.

Some have popularized the microservice concept as a way to describe this kind of system. According to the microservices.io website, a microservice consists of a set of narrowly focused, independently deployable services. They contrast this with the monolithic application-deployment pattern where every aspect of the system is integrated into one bundle (such as a single WAR file for a Java EE app server). The microservice model gives developers much-needed flexibility.

Some advantages of microservices are as follows:

- Each microservice can be managed by a small team.
- Each team can work on its own schedule, so long as the service API compatibility is maintained.
- Microservices can be deployed independently should this be required, such as for easier testing.
- It's easier to switch technology stack choices.

Where does Node.js fit in with this? Its design fits the microservice model like a glove:

- Node.js encourages small, tightly focused, single-purpose modules.
- These modules are composed into an application by the excellent npm package management system.
- Publishing modules is incredibly simple, whether via the NPM repository or a Git URL.
- While an app framework such as Express can be used with large services, it works very well for small lightweight services and supports easy, simple deployment.

In short, it's easy to use Node.js in a lean and agile fashion, building large or small services depending on your architecture preferences.

Summary

You learned a lot in this chapter. Specifically, you saw that JavaScript has a life outside web browsers and that Node.js is an excellent programming platform with many interesting attributes. While it is a relatively young project, Node.js has become very popular and is widely used not just for web applications but for command-line developer tools and much more. Because the Node.js platform is based on Chrome's V8 JavaScript engine, the project has been able to keep up with the rapid improvements to the JavaScript language.

The Node.js architecture consists of asynchronous functions managed by an event loop triggering callback functions, rather than using threads and blocking I/O. This architecture has claimed performance benefits that seem to offer many benefits, including the ability to do more work with less hardware. But we also learned that inefficient algorithms can erase any performance benefits.

Our focus in this book is the real-world considerations of developing and deploying Node.js applications. We'll cover as many aspects of developing, refining, testing, and deploying Node.js applications as we can.

Now that we've had this introduction to Node.js, we're ready to dive in and start using it. In *Chapter 2, Setting up Node.js*, we'll go over how to set up a Node.js development environment on Mac, Linux, or Windows, and even write some code. So let's get started.

2

Setting Up Node.js

Before getting started with using Node.js, you must set up your development environment. While it's very easy to set up, there are a number of considerations to think about, including whether to install Node.js using the package management system, satisfying the requirements for installing native code Node.js packages, and deciding what the best editor is to use with Node.js. In the following chapters, we'll use this environment for development and non-production deployment.

In this chapter, we will cover the following topics:

- How to install Node.js from source and prepackaged binaries on Linux, macOS, or Windows
- How to install **node package manager (npm)** and some other popular tools
- The Node.js module system
- Node.js and JavaScript language improvements from the ECMAScript committee

System requirements

Node.js runs on POSIX-like OSes, various UNIX derivatives (Solaris, for example), and UNIX-workalike OSes (such as Linux, macOS, and so on), as well as on Microsoft Windows. It can run on machines both large and small, including tiny ARM devices, such as Raspberry Pi—a microscale embeddable computer for DIY software/hardware projects.

Node.js is now available via package management systems, limiting the need to compile and install from the source.

Because many Node.js packages are written in C or C++, you must have a C compiler (such as GCC), Python 2.7 (or later), and the `node-gyp` package. Since Python 2 will be end-of-lifed by the end of 2019, the Node.js community is rewriting its tools for Python 3 compatibility. If you plan on using encryption in your networking code, you will also need the OpenSSL cryptographic library. Modern UNIX derivatives almost certainly come with this and Node.js's `configure` script—used when installing from the source—will detect their presence. If you need to install it, Python is available at <http://python.org> and OpenSSL is available at <http://openssl.org>.

Now that we have covered the requirements for running Node.js, let's learn how to install it.

Installing Node.js using package managers

The preferred method for installing Node.js is to use the versions available in package managers, such as `apt-get`, or MacPorts. Package managers make your life easier by helping to maintain the current version of the software on your computer, ensuring to update dependent packages as necessary, all by typing a simple command, such as `apt-get update`. Let's go over installation from a package management system first.



For the official instructions on installing from package managers, go to <https://nodejs.org/en/download/package-manager/>.

Installing Node.js on macOS with MacPorts

The MacPorts project (<http://www.macports.org/>) has been packaging a long list of open-source software packages for macOS for years and they have packaged Node.js. The commands it manages are, by default, installed on `/opt/local/bin`. After you have installed MacPorts using the installer on their website, installing Node.js is very simple, making the Node.js binaries available in the directory where MacPorts installs commands:

```
$ port search nodejs npm
...
nodejs8 @8.16.2 (devel, net)
```

```
Evented I/O for V8 JavaScript

nodejs10 @10.16.3 (devel, net)
    Evented I/O for V8 JavaScript

nodejs12 @12.13.0 (devel, net)
    Evented I/O for V8 JavaScript

nodejs14 @14.0.0 (devel, net)
    Evented I/O for V8 JavaScript
...
npm6 @6.14.4 (devel)
    node package manager

$ sudo port install nodejs14 npm6
... long log of downloading and installing prerequisites and Node
$ which node
/opt/local/bin/node
$ node --version
v14.0.0
```

If you have followed the directions for setting up MacPorts, the MacPorts directory is already in your PATH environment variable. Running the `node`, `npm`, or `npx` commands is then simple. This proves Node.js has been installed and the installed version matched what you asked for.

MacPorts isn't the only tool for managing open source software packages on macOS.

Installing Node.js on macOS with Homebrew

Homebrew is another open source software package manager for macOS, which some say is the perfect replacement for MacPorts. It is available through their home page at <http://brew.sh/>. After installing Homebrew using the instructions on their website and ensuring that it is correctly set up, use the following code:

```
$ brew update
... long wait and lots of output
$ brew search node
==> Searching local taps...
node libbitcoin-node node-build node@8 nodeenv
leafnode llnode node node@10 node@12 nodebrew nodenv
==> Searching taps on GitHub...
caskroom/cask/node-profiler
==> Searching blacklisted, migrated and deleted formulae...
```

Then, install like this:

```
$ brew install node
...
==> Installing node
==> Downloading
https://homebrew.bintray.com/bottles/node-14.0.0_1.high_sierra.bottle.
tar.gz
#####
... 100.0%
==> Pouring node-14.0.0_1.high_sierra.bottle.tar.gz
==> Caveats
Bash completion has been installed to:
  /usr/local/etc/bash_completion.d
==> Summary
  /usr/local/Cellar/node/14.0.0_1: 4,660 files, 60MB
```

Like MacPorts, Homebrew installs commands on a public directory, which defaults to `/usr/local/bin`. If you have followed the Homebrew instructions to add that directory to your `PATH` variable, run the `Node.js` command as follows:

```
$ node --version
v14.0.0
```

This proves `Node.js` has been installed and the installed version matched what you asked for.

Of course, macOS is only one of many operating systems we might use.

Installing Node.js on Linux, *BSD, or Windows from package management systems

`Node.js` is now available through most package management systems. Instructions on the `Node.js` website currently list packaged versions of `Node.js` for a long list of Linux, as well as FreeBSD, OpenBSD, NetBSD, macOS, and even Windows. Visit <https://nodejs.org/en/download/package-manager/> for more information.

For example, on Debian and other Debian-based Linux distributions (such as Ubuntu), use the following commands:

```
$ curl -sL https://deb.nodesource.com/setup_14.x | sudo -E bash -
[sudo] password for david:
## Installing the NodeSource Node.js 14.x repo...
```

```
## Populating apt-get cache...

... much apt-get output
## Run `sudo apt-get install -y nodejs` to install Node.js 13.x and
npm
## You may also need development tools to build native addons:
    sudo apt-get install gcc g++ make
$ sudo apt-get install -y nodejs
... Much output
$ sudo apt-get install -y gcc g++ make build-essential
... Much output
```

This adds the NodeSource APT repository to the system, updates the package data, and prepares the system so that you can install Node.js packages. It also instructs us on how to install Node.js and the required compiler and developer tools.

To download other Node.js versions (this example shows version 14.x), modify the URL to suit you:

```
$ node --version
v14.0.0
```

The commands will be installed in `/usr/bin` and we can test whether the version downloaded is what we asked for.

Windows is starting to become a place where Unix/Linux geeks can work, thanks to a new tool called the **Windows subsystem for Linux (WSL)**.

Installing Node.js in WSL

WSL lets you install Ubuntu, openSUSE, or SUSE Linux Enterprise on Windows. All three are available via the store built into Windows 10. You may need to update your Windows device for the installation to work. For the best experience, install WSL2, which is a major overhaul of WSL, offering an improved integration between Windows and Linux.

Once installed, the Linux-specific instructions will install Node.js in the Linux subsystem.



To install WSL, see <https://msdn.microsoft.com/en-us/commandline/wsl/install-win10>.

To learn about and install WSL2, see <https://docs.microsoft.com/en-us/windows/wsl/wsl2-index>.

The process may require elevated privileges on Windows.

Opening an administrator-privileged PowerShell on Windows

Some of the commands that you'll run while installing tools on Windows are to be executed in a PowerShell window with elevated privileges. We are mentioning this because during the process of enabling WSL, a command will need to be run in a PowerShell window.

The process is simple:

1. In the Start menu, enter `PowerShell` in the application's search box. The resulting menu will list **PowerShell**.
2. Right-click the **PowerShell** entry.
3. The context menu that comes up will have an entry called **Run as Administrator**. Click on that.

The resulting command window will have administrator privileges and the title bar will say **Administrator: Windows PowerShell**.

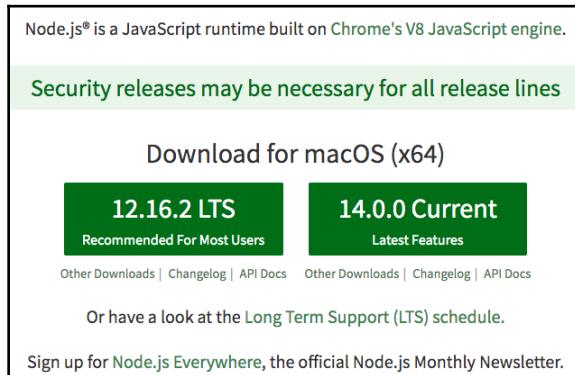
In some cases, you will be unable to use Node.js from package management systems.

Installing the Node.js distribution from nodejs.org

The <https://nodejs.org/en/> website offers built-in binaries for Windows, macOS, Linux, and Solaris. We can simply go to the website, click on the **Install** button, and run the installer. For systems with package managers, such as the ones we've just discussed, it's better to use the package management system. That's because you'll find it easier to stay up to date with the latest version. However, that doesn't serve all people because of the following reasons:

- Some will prefer to install a binary rather than deal with the package manager.
- Their chosen system doesn't have a package management system.
- The Node.js implementation in their package management system is out of date.

Simply go to the Node.js website and you'll see something as in the following screenshot. The page does its best to determine your OS and supply the appropriate download. If you need something different, click on the **DOWNLOADS** link in the header for all possible downloads:



For macOS, the installer is a `PKG` file that gives the typical installation process. For Windows, the installer simply takes you through the typical install wizard process.

Once you are finished with the installer, you have command-line tools, such as `node` and `npm`, which you can run Node.js programs with. On Windows, you're supplied with a version of the Windows command shell preconfigured to work nicely with Node.js.

As you have just learned, most of us will be perfectly satisfied with installing prebuilt packages. However, there are times when we must install Node.js from a source.

Installing from the source on POSIX-like systems

Installing the prepackaged Node.js distributions is the preferred installation method. However, installing Node.js from a source is desirable in a few situations:

- It can let you optimize the compiler settings as desired.
- It can let you cross-compile, say, for an embedded ARM system.
- You might need to keep multiple Node.js builds for testing.
- You might be working on Node.js itself.

Now that you have a high-level view, let's get our hands dirty by mucking around in some build scripts. The general process follows the usual `configure`, `make`, and `make install` routine that you may have already performed with other open source software packages. If not, don't worry, we'll guide you through the process.



The official installation instructions are in `README.md`, contained in the source distribution at <https://github.com/nodejs/node/blob/master/README.md>.

Installing prerequisites

There are three prerequisites: a C compiler, Python, and the OpenSSL libraries. The Node.js compilation process checks for their presence and will fail if the C compiler or Python is not present. These sorts of commands will check for their presence:

```
$ cc --version
Apple LLVM version 10.0.0 (clang-1000.11.45.5)
Target: x86_64-apple-darwin17.7.0
Thread model: posix
InstalledDir:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin
$ python
Python 2.7.16 (default, Oct 16 2019, 00:35:27)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.31)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



Go to <https://github.com/nodejs/node/blob/master/BUILDING.md> for details on the requirements.

The specific method for installing these depends on your OS.

The Node.js build tools are in the process of being updated to support Python 3.x. Python 2.x is in an end-of-life process, slated for the end of 2019, so it is therefore recommended that you update to Python 3.x.

Before we can compile the Node.js source, we must have the correct tools installed and on macOS, there are a couple of special considerations.

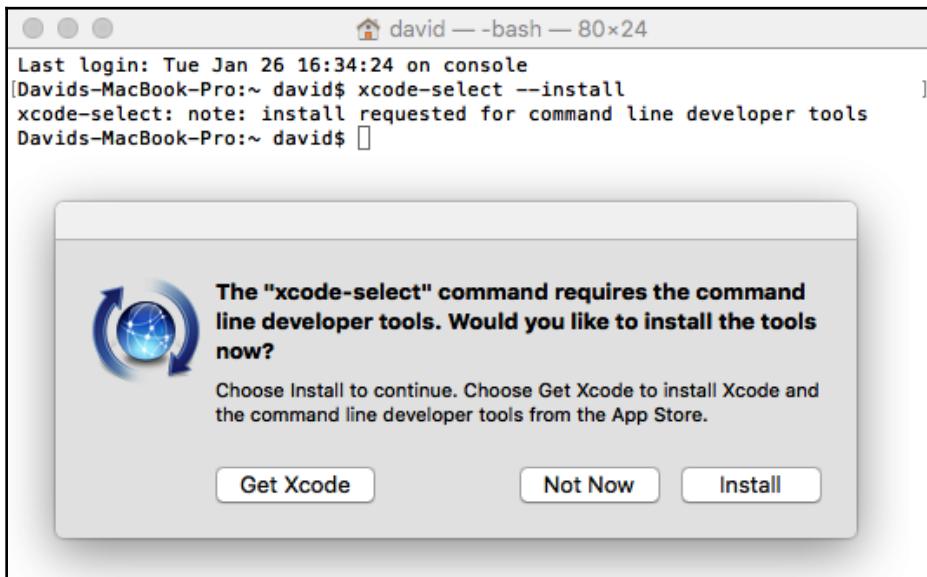
Installing developer tools on macOS

Developer tools (such as GCC) are an optional installation on macOS. Fortunately, they're easy to acquire.

You start with Xcode, which is available for free through the Macintosh app store. Simply search for Xcode and click on the Get button. Once you have Xcode installed, open a Terminal window and type the following:

```
$ xcode-select --install
```

This installs the Xcode command-line tools:



For additional information, visit

<http://osxdaily.com/2014/02/12/install-command-line-tools-mac-os-x/>.

Now that we have the required tools installed, we can proceed with compiling the Node.js source.

Installing from the source for all POSIX-like systems

Compiling Node.js from the source follows this familiar process:

1. Download the source from <http://nodejs.org/download>.
2. Configure the source for building using `./configure`.
3. Run `make`, then `make install`.

The source bundle can be downloaded through your browser or as follows, substituting your preferred version:

```
$ mkdir src  
$ cd src  
$ wget https://nodejs.org/download/release/v14.0.0/node-v14.0.0.tar.gz  
$ tar xvfz node-v14.0.0.tar.gz  
$ cd node-v14.0.0
```

Now, we configure the source so that it can be built. This is just like with many other open source packages and there is a long list of options to customize the build:

```
$ ./configure --help
```

To cause the installation to land in your `home` directory, run it this way:

```
$ ./configure --prefix=$HOME/node/14.0.0  
..output from configure
```

If you're going to install multiple Node.js versions side by side, it's useful to put the version number in the path like this. That way, each version will sit in a separate directory. It will then be a simple matter of switching between Node.js versions by changing the `PATH` variable appropriately:

```
# On bash shell:  
$ export PATH=${HOME}/node/VERSION-NUMBER/bin:$PATH  
# On csh  
$ setenv PATH ${HOME}/node/VERSION-NUMBER/bin:$PATH
```

A simpler way to install multiple Node.js versions is by using the `nvm` script, which will be described later.

If you want to install Node.js in a system-wide directory, simply leave off the `--prefix` option and it will default to installing in `/usr/local`.

After a moment, it'll stop and will likely have successfully configured the source tree for installation in your chosen directory. If this doesn't succeed, the error messages that are printed will describe what needs to be fixed. Once the configure script is satisfied, you can move on to the next step.

With the configure script satisfied, you compile the software:

```
$ make  
.. a long log of compiler output is printed  
$ make install
```

If you are installing on a system-wide directory, perform the last step this way instead:

```
$ make  
$ sudo make install
```

Once installed, you should make sure that you add the installation directory to your PATH variable, as follows:

```
$ echo 'export PATH=$HOME/node/14.0.0/bin:${PATH}' >>~/.bashrc  
$ . ~/.bashrc
```

Alternatively, for csh users, use this syntax to make an exported environment variable:

```
$ echo 'setenv PATH $HOME/node/14.0.0/bin:${PATH}' >>~/.cshrc  
$ source ~/.cshrc
```

When the build is installed, it creates a directory structure, as follows:

```
$ ls ~/node/14.0.0/  
bin include lib share  
$ ls ~/node/14.0.0/bin  
node npm npx
```

Now that we've learned how to install Node.js from the source on UNIX-like systems, we get to do the same on Windows.

Installing from the source on Windows

The BUILDING.md document referenced previously has instructions. You can use the build tools from Visual Studio or the full Visual Studio 2017 or 2019 product:

- Visual Studio 2019: <https://www.visualstudio.com/downloads/>
- The build tools: <https://visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2019>

Three additional tools are required:

- Git for Windows: <http://git-scm.com/download/win>
- Python: <https://www.python.org/>
- OpenSSL: <https://www.openssl.org/source/> and <https://wiki.openssl.org/index.php/Binaries>
- The **Netwide Assembler (NASM)** for OpenSSL: <https://www.nasm.us/>

Then, run the included `.\vcbuild` script to perform the build.

We've learned how to install one Node.js instance, so let's now take it to the next level by installing multiple instances.

Installing multiple Node.js instances with nvm

Normally, you wouldn't install multiple versions of Node.js—doing so adds complexity to your system. But if you are hacking on Node.js itself or testing your software against different Node.js releases, you may want to have multiple Node.js installations. The method to do so is a simple variation on what we've already discussed.

Earlier, while discussing building Node.js from the source, we noted that you can install multiple Node.js instances in separate directories. It's only necessary to build from the source if you need a customized Node.js build but most folks would be satisfied with pre-built Node.js binaries. They, too, can be installed on separate directories.

Switching between Node.js versions is simply a matter of changing the `PATH` variable (on POSIX systems), as in the following code, using the directory where you installed Node.js:

```
$ export PATH=/usr/local/node/VERSION-NUMBER/bin:$PATH
```

It starts to get a little tedious maintaining this after a while. For each release, you have to set up Node.js, npm, and any third-party modules you desire in your Node.js installation. Also, the command shown to change `PATH` is not quite optimal. Inventive programmers have created several version managers to simplify managing multiple Node.js/npm releases and provide commands to change `PATH` the smart way:

- Node version manager: <https://github.com/tj/n>
- Node version manager: <https://github.com/creationix/nvm>

Both maintain multiple, simultaneous versions of Node.js and let you easily switch between versions. Installation instructions are available on their respective websites.

For example, with `nvm`, you can run commands such as these:

```
$ nvm ls
...
v6.4.0
...
v6.11.2
v8.9.3
v10.15.2
...
v12.13.1
...
v14.0.0
-> system
default -> 12.9.1 (-> v12.9.1)
node -> stable (-> v12.13.1) (default)
stable -> 12.13 (-> v12.13.1) (default)
$ nvm use 10
Now using node v10.15.2 (npm v6.4.1)
$ node --version
v10.15.2
$ nvm use 4.9
Now using node v4.9.1 (npm v2.15.11)
$ node --version
v4.9.1
$ nvm install 14
Downloading and installing node v14.0.0...
Downloading
```

```
https://nodejs.org/dist/v14.0.0/node-v14.0.0-darwin-x64.tar.xz...
#####
... 100.0%
Computing checksum with shasum -a 256
Checksums matched!
Now using node v14.0.0 (npm v6.14.4)
$ node --version
v14.0.0
$ which node
/Users/david/.nvm/versions/node/v14.0.0/bin/node
$ /usr/local/bin/node --version
v13.13.0
$ /opt/local/bin/node --version
v13.13.0
```

In this example, we first listed the available versions. Then, we demonstrated how to switch between Node.js versions, verifying the version changed each time. We also installed and used a new version using `nvm`. Finally, we showed the directory where `nvm` installs Node.js packages versus Node.js versions that are installed using MacPorts or Homebrew.

This demonstrates that you can have Node.js installed system-wide, keep multiple private Node.js versions managed by `nvm`, and switch between them as needed. When new Node.js versions are released, they are simple to install with `nvm`, even if the official package manager for your OS hasn't yet updated its packages.

Installing `nvm` on Windows

Unfortunately, `nvm` doesn't support Windows. Fortunately, a couple of Windows-specific clones of the `nvm` concept exist:

- Node.js version management utility for Windows: <https://github.com/coreybutler/nvm-windows>
- Natural Node.js and npm version manager for Windows: <https://github.com/marcelklehr/nodist>

Another route is to use WSL. Because in WSL you're interacting with a Linux command line, you can use `nvm` itself. But let's stay focused on what you can do in Windows.

Many of the examples in this book were tested using the `nvm-windows` application. There are slight behavior differences but it acts largely the same as `nvm` for Linux and macOS. The biggest change is the version number specifier in the `nvm use` and `nvm install` commands.

With `nvm` for Linux and macOS, you can type a simple version number, such as `nvm use 8`, and it will automatically substitute the latest release of the named Node.js version. With `nvm-windows`, the same command acts as if you typed `nvm use 8.0.0`. In other words, with `nvm-windows`, you must use the exact version number. Fortunately, the list of supported versions is easily available using the `nvm list` available command.

Using a tool such as `nvm` simplifies the process of testing a Node.js application against multiple Node.js versions.

Now that we can install Node.js, we need to make sure we are installing any Node.js module that we want to use. This requires having build tools installed on our computer.

Requirements for installing native code modules

While we won't discuss native code module development in this book, we do need to make sure that they can be built. Some modules in the npm repository are native code and they must be compiled with a C or C++ compiler to build the corresponding `.node` files (the `.node` extension is used for binary native code modules).

The module will often describe itself as a wrapper for some other library. For example, the `libxmljs` and `libxslt` modules are wrappers around the C/C++ libraries of the same name. The module includes the C/C++ source code and when installed, a script is automatically run to do the compilation with `node-gyp`.

The `node-gyp` tool is a cross-platform command-line tool written in Node.js for compiling native add-on modules for Node.js. We've mentioned native code modules several times and it is this tool that compiles them for use with Node.js.

You can easily see this in action by running these commands:

```
$ mkdir temp  
$ cd temp  
$ npm install libxmljs libxslt
```

This is done in a temporary directory, so you can delete it afterward. If your system does not have the tools installed to compile native code modules, you'll see error messages. Otherwise, you'll see a `node-gyp` execution in the output, followed by many lines of text obviously related to compiling C/C++ files.

The `node-gyp` tool has prerequisites similar to those for compiling Node.js from the source—namely, a C/C++ compiler, a Python environment, and other build tools, such as Git. For Unix, macOS, and Linux systems, those are easy to come by. For Windows, you should install the following:

- Visual Studio build tools: <https://www.visualstudio.com/downloads/#build-tools-for-visual-studio-2017>
- Git for Windows: <http://git-scm.com/download/win>
- Python for Windows: <https://www.python.org/>

Normally, you don't need to worry about installing `node-gyp`. That's because it is installed behind the scenes as part of npm. That's done so that npm can automatically build native code modules.

Its GitHub repository contains documentation; go to <https://github.com/nodejs/node-gyp>.

Reading the `node-gyp` documentation in its repository will give you a clearer understanding of the compilation prerequisites discussed previously and of developing native code modules.

This is an example of a non-explicit dependency. It is best to explicitly declare all the things that a software package depends on. In Node.js, dependencies are declared in `package.json` so that the package manager (npm or yarn) can download and set up everything. But these compiler tools are set up by the OS package management system, which is outside the control of npm or yarn. Therefore, we cannot explicitly declare those dependencies.

We've just learned that Node.js supports modules written not just in JavaScript, but also in other programming languages. We've also learned how to support the installation of such modules. Next, we will learn about Node.js version numbers.

Choosing Node.js versions to use and the version policy

We just threw around so many different Node.js version numbers in the previous section that you may have become confused about which version to use. This book is targeted at Node.js version 14.x and it's expected that everything we'll cover is compatible with Node.js 10.x and any subsequent release.

Starting with Node.js 4.x, the Node.js team has followed a dual-track approach. The even-numbered releases (4.x, 6.x, 8.x, and so on) are what they're calling **long term support (LTS)**, while the odd-numbered releases (5.x, 7.x, 9.x, and so on) are where current new feature development occurs. While the development branch is kept stable, the LTS releases are positioned as being for production use and will receive updates for several years.

At the time of writing, Node.js 12.x is the current LTS release; Node.js 14.x has been released and will eventually become the LTS release.

A major impact of each new Node.js release, beyond the usual performance improvements and bug fixes, is the bringing in of the latest V8 JavaScript engine release. In turn, this means bringing in more of the ES2015/2016/2017 features as the V8 team implements them. In Node.js 8.x, the `async/await` functions arrived and in Node.js 10.x, support for the standard ES6 module format has arrived. In Node.js 14.x that module format will be completely supported.

A practical consideration is whether a new Node.js release will break your code. New language features are always being added as V8 catches up with ECMAScript and the Node.js team sometimes makes groundbreaking changes to the Node.js API. If you've tested on one Node.js version, will it work on an earlier version? Will a Node.js change break some assumptions we made?

What npm does is ensure that our packages execute on the correct Node.js version. This means that we can specify the compatible Node.js versions for a package in the `package.json` file (which we'll explore in [Chapter 3, Exploring Node.js Modules](#)).

We can add an entry to `package.json` as follows:

```
engines: {  
  "node": ">=8.x"  
}
```

This means exactly what it implies—that the given package is compatible with Node.js version 8.x or later.

Of course, your development environment(s) could have several Node.js versions installed. You'll need the version your software is declared to support, plus any later versions you wish to evaluate.

We have just learned how the Node.js community manages releases and version numbers. Our next step is to discuss which editor to use.

Choosing editors and debuggers for Node.js

Since Node.js code is JavaScript, any JavaScript-aware editor will be useful. Unlike some other languages that are so complex that an IDE with code completion is a necessity, a simple programming editor is perfectly sufficient for Node.js development.

Two editors are worth shouting out because they are written in Node.js: Atom and Microsoft Visual Studio Code.

Atom (<https://atom.io/>) describes itself as a hackable editor for the 21st century. It is extendable by writing Node.js modules using the Atom API and the configuration files are easily editable. In other words, it's hackable in the same way plenty of other editors have been—going back to Emacs, meaning you write a software module to add capabilities to the editor. The Electron framework was invented in order to build Atom and it is a super-easy way of building desktop applications using Node.js.

Microsoft Visual Studio Code (<https://code.visualstudio.com/>) is a hackable editor (well, the home page says extensible and customizable, which means the same thing) that is also open source and implemented in Electron. However, it's not a hollow me-too editor, copying Atom while adding nothing of its own. Instead, Visual Studio Code is a solid programmer's editor in its own right, bringing interesting functionality to the table.

As for debuggers, there are several interesting choices. Starting with Node.js 6.3, the `inspector` protocol has made it possible to use the Google Chrome debugger. Visual Studio Code has a built-in debugger that also uses the `inspector` protocol.



For a full list of debugging options and tools, see <https://nodejs.org/en/docs/guides/debugging-getting-started/>.

Another task related to the editor is adding extensions to help with the editing experience. Most programmer-oriented editors allow you to extend the behavior and assist with writing the code. A trivial example is syntax coloring for JavaScript, CSS, HTML, and so on. Code completion extensions are where the editor helps you write the code. Some extensions scan code for common errors; often these extensions use the word *lint*. Some extensions help to run unit test frameworks. Since there are so many editors available, we cannot provide specific suggestions.

For some, the choice of programming editor is a serious matter defended with fervor, so we carefully recommend that you use whatever editor you prefer, as long as it helps you edit JavaScript code. Next, we will learn about the Node.js commands and a little about running Node.js scripts.

Running and testing commands

Now that you've installed Node.js, we want to do two things—verify that the installation was successful and familiarize ourselves with the Node.js command-line tools and running simple scripts with Node.js. We'll also touch again on `async` functions and look at a simple example HTTP server. We'll finish off with the `npm` and `npx` command-line tools.

Using Node.js's command-line tools

The basic installation of Node.js includes two commands: `node` and `npm`. We've already seen the `node` command in action. It's used either for running command-line scripts or server processes. The other, `npm`, is a package manager for Node.js.

The easiest way to verify that your Node.js installation works is also the best way to get help with Node.js. Type the following command:

```
$ node --help
Usage: node [options] [ -e script | script.js | - ] [arguments]
      node inspect script.js [arguments]

Options:
  -v, --version print Node.js version
```

```
-e, --eval script evaluate script
-p, --print evaluate script and print result
-c, --check syntax check script without executing
-i, --interactive always enter the REPL even if stdin
                                does not appear to be a terminal
-r, --require module to preload (option can be repeated)
- script read from stdin (default; interactive mode if a tty)

... many more options

Environment variables:
NODE_DEBUG ', '-separated list of core modules that should print debug
information
NODE_DEBUG_NATIVE ', '-separated list of C++ core debug categories that
should print debug output
NODE_DISABLE_COLORS set to 1 to disable colors in the REPL
NODE_EXTRA_CA_CERTS path to additional CA certificates file
NODE_NO_WARNINGS set to 1 to silence process warnings
NODE_OPTIONS set CLI options in the environment via a space-separated
list
NODE_PATH ':'-separated list of directories prefixed to the module
search path
... many more environment variables
```

That was a lot of output but don't study it too closely. The key takeaway is that `node --help` provides a lot of useful information.

Note that there are options for both Node.js and V8 (not shown in the previous command line). Remember that Node.js is built on top of V8; it has its own universe of options that largely focus on details of bytecode compilation or garbage collection and heap algorithms. Enter `node --v8-options` to see the full list of these options.

On the command line, you can specify options, a single script file, and a list of arguments to that script. We'll discuss script arguments further in the following section, *Running a simple script with Node.js*.

Running Node.js with no arguments drops you in an interactive JavaScript shell:

```
$ node
> console.log('Hello, world!');
Hello, world!
undefined
```

Any code you can write in a Node.js script can be written here. The command interpreter gives a good terminal-oriented user experience and is useful for interactively playing with your code. You do play with your code, don't you? Good!

Running a simple script with Node.js

Now, let's look at how to run scripts with Node.js. It's quite simple; let's start by referring to the help message shown previously. The command-line pattern is just a script filename and some script arguments, which should be familiar to anyone who has written scripts in other languages.



Creating and editing Node.js scripts can be done with any text editor that deals with plain text files, such as VI/VIM, Emacs, Notepad++, Atom, Visual Studio Code, Jedit, BB Edit, TextMate, or Komodo. It's helpful if it's a programmer-oriented editor, if only for the syntax coloring.

For this and other examples in this book, it doesn't truly matter where you put the files. However, for the sake of neatness, you can start by making a directory named `node-web-dev` in the `home` directory of your computer and inside that, creating one directory per chapter (for example, `chap02` and `chap03`).

First, create a text file named `ls.js` with the following content:

```
const fs = require('fs').promises;

async function listFiles() {
  try {
    const files = await fs.readdir('.');
    for (const file of files) {
      console.log(file);
    }
  } catch (err) {
    console.error(err);
  }
}

listFiles();
```

Next, run it by typing the following command:

```
$ node ls.js  
ls.js
```

This is a pale and cheap imitation of the Unix `ls` command (as if you couldn't figure that out from the name!). The `readdir` function is a close analog to the Unix `readdir` system call used to list the files in a directory. On Unix/Linux systems, we can run the following command to learn more:

```
$ man 3 readdir
```

The `man` command, of course, lets you read manual pages and section 3 covers the C library.

Inside the function body, we read the directory and print its contents. Using `require('fs').promises` gives us a version of the `fs` module (filesystem functions) that returns Promises; it, therefore, works well in an `async` function. Likewise, the ES2015 `for..of` loop construct lets us loop over entries in an array in a way that works well in `async` functions.



By default, the `fs` module functions use the callback paradigm originally created for Node.js. As a result, most Node.js modules use the callback paradigm. Within `async` functions, it is more convenient if functions return Promises instead so that the `await` keyword can be used. The `util` module provides a function, `util.promisify`, which generates a wrapper function for old-style callback-oriented functions so it instead returns a Promise.

This script is hardcoded to list files in the current directory. The real `ls` command takes a directory name, so let's modify the script a little.

Command-line arguments land in a global array named `process.argv`. Therefore, we can modify `ls.js`, copying it as `ls2.js` (as follows) to see how this array works:

```
const fs = require('fs').promises;  
  
async function listFiles() {  
  try {  
    var dir = '.';  
    if (process.argv[2]) dir = process.argv[2];  
    const files = await fs.readdir(dir);  
    for (let fn of files) {  
      console.log(fn);  
    }  
  }  
}
```

```
        } catch (err) {
            console.error(err);
        }
    }

listFiles();
```

You can run it as follows:

```
$ pwd
/Users/David/chap02
$ node ls2 ..
chap01
chap02
$ node ls2
app.js
ls.js
ls2.js
```

We simply checked whether a command-line argument was present, if (`process.argv[2]`). If it was, we override the value of the `dir` variable, `dir = process.argv[2]`, and we then use that as the `readdir` argument:

```
$ node ls2.js /nonexistent
{ Error: ENOENT: no such file or directory, scandir '/nonexistent'
  errno: -2,
  code: 'ENOENT',
  syscall: 'scandir',
  path: '/nonexistent' }
```

If you give it a non-existent directory pathname, an error will be thrown and printed using the `catch` clause.

Writing inline async arrow functions

There is a different way to write these examples that some feel is more concise. These examples were written as a regular function—with the `function` keyword—but with the `async` keyword in front. One of the features that came with ES2015 is the arrow function, which lets us streamline the code a little bit.

Combined with the `async` keyword, an `async` arrow function looks like this:

```
async () => {
    // function body
}
```

You can use this anywhere; for example, the function can be assigned to a variable or it can be passed as a callback to another function. When used with the `async` keyword, the body of the arrow function has all of the `async` function's behavior.

For the purpose of these examples, an `async` arrow function can be wrapped for immediate execution:

```
(async () => {
  // function body
})()
```

The final parenthesis causes the inline function to immediately be invoked.

Then, because `async` functions return a `Promise`, it is necessary to add a `.catch` block to catch errors. With all that, the example looks as follows:

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

(async () => {
  var dir = '.';
  if (process.argv[2]) dir = process.argv[2];
  const files = await fs.readdir(dir);
  for (let fn of files) {
    console.log(fn);
  }
})().catch(err => { console.error(err); });
```

Whether this or the previous style is preferable is perhaps a matter of taste. However, you will find both styles in use and it is necessary to understand how both work.

When invoking an `async` function at the top level of a script, it is necessary to capture any errors and report them. Failure to catch and report errors can lead to mysterious problems that are hard to pin down. For the original version of this example, the errors were explicitly caught with a `try/catch` block. In this version, we catch errors using a `.catch` block.

Before we had `async` functions, we had the `Promise` object and before that, we had the callback paradigm. All three paradigms are still used in Node.js, meaning you'll need to understand each.

Converting to async functions and the Promise paradigm

In the previous section, we discussed `util.promisify` and its ability to convert a callback-oriented function into one that returns a Promise. The latter plays well with `async` functions and therefore, it is preferable for functions to return a Promise.

To be more precise, `util.promisify` is to be given a function that uses the error-first-callback paradigm. The last argument of such functions is a callback function, whose first argument is interpreted as an error indicator, hence the phrase error-first-callback. What `util.promisify` returns is another function that returns a Promise.

The Promise serves the same purpose as error-first-callback. If an error is indicated, the Promise resolves to the rejected status, while if success is indicated, the Promise resolves to a success status. As we see in these examples, the Promise is handled very nicely within an `async` function.

The Node.js ecosystem has a large body of functions that use error-first-callback. The community has began a conversion process where functions will return a Promise and possibly also take an error-first-callback for API compatibility.

One of the new features in Node.js 10 is an example of such a conversion. Within the `fs` module is a submodule, named `fs.promises`, with the same API but producing Promise objects. We wrote the previous examples using that API.

Another choice is a third-party module, `fs-extra`. This module has an extended API beyond the standard `fs` module. On one hand, its functions return a Promise if no callback function is provided or else invokes the callback. In addition, it includes several useful functions.



In the rest of this book, we will often use `fs-extra` because of those additional functions. For documentation on the module, go to <https://www.npmjs.com/package/fs-extra>.

The `util` module has another function, `util.callbackify`, which does as the name implies—it converts a function that returns a Promise into one that uses a callback function.

Now that we've seen how to run a simple script, let's look at a simple HTTP server.

Launching a server with Node.js

Many scripts that you'll run are server processes; we'll be running lots of these scripts later on. Since we're still trying to verify the installation and get you familiar with using Node.js, we want to run a simple HTTP server. Let's borrow the simple server script on the Node.js home page (<http://nodejs.org>).

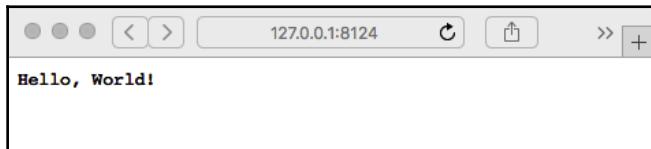
Create a file named `app.js`, containing the following:

```
const http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, World!\n');
}).listen(8124, '127.0.0.1');
console.log('Server running at http://127.0.0.1:8124');
```

Run it as follows:

```
$ node app.js
Server running at http://127.0.0.1:8124
```

This is the simplest of web servers you can build with Node.js. If you're interested in how it works, flip forward to Chapter 4, *HTTP Servers and Clients*, Chapter 5, *Your First Express Application*, and Chapter 6, *Implementing the Mobile-First Paradigm*. But for now, just type `http://127.0.0.1:8124` in your browser to see the **Hello, World!** message:



A question to ponder is why this script didn't exit when `ls.js` did. In both cases, execution of the script reaches the end of the file; the Node.js process does not exit in `app.js`, while it does in `ls.js`.

The reason for this is the presence of active event listeners. Node.js always starts up an event loop and in `app.js`, the `listen` function creates an event, `listener`, that implements the HTTP protocol. This `listener` event keeps `app.js` running until you do something, such as press `Ctrl + C` in the terminal window. In `ls.js`, there is nothing there to create a long-running `listener` event, so when `ls.js` reaches the end of its script, the `node` process will exit.

To carry out more complex tasks with Node.js, we must use third-party modules. The npm repository is the place to go.

Using npm, the Node.js package manager

Node.js, being a JavaScript interpreter with a few interesting asynchronous I/O libraries, is by itself a pretty basic system. One of the things that makes Node.js interesting is the rapidly growing ecosystem of third-party modules for Node.js.

At the center of that ecosystem is the npm module repository. While Node.js modules can be downloaded as source and assembled manually for use with Node.js programs, that's tedious to do and it's difficult to implement a repeatable build process. npm gives us a simpler method; npm is the de facto standard package manager for Node.js and it greatly simplifies downloading and using these modules. We will talk about npm at length in the next chapter.

The sharp-eyed among you will have noticed that npm is already installed via all the installation methods discussed previously. In the past, npm was installed separately, but today it is bundled with Node.js.

Now that we have npm installed, let's take it for a quick spin. The **hexy** program is a utility used for printing hex dumps of files. That's a very 1970s thing to do, but it is still extremely useful. It serves our purpose right now as it gives us something to quickly install and try out:

```
$ npm install -g hexy
/opt/local/bin/hexy ->
/opt/local/lib/node_modules/hexy/bin/hexy_cmd.js
+ hexy@0.2.10
added 1 package in 1.107s
```

Adding the `-g` flag makes the module available globally, irrespective of the present working directory of your command shell. A global install is most useful when the module provides a command-line interface. When a package provides a command-line script, npm sets that up. For a global install, the command is installed correctly for use by all users of the computer.

Depending on how Node.js is installed for you, it may need to be run with `sudo`:

```
$ sudo npm install -g hexy
```

Once it is installed, you'll be able to run the newly-installed program this way:

```
$ hexy --width 12 ls.js
00000000: 636f 6e73 7420 6673 203d 2072 const.fs.=.r
0000000c: 6571 7569 7265 2827 6673 2729 equire('fs')
00000018: 3b0a 636f 6e73 7420 7574 696c ;.const.util
00000024: 203d 2072 6571 7569 7265 2827 .=.require('
00000030: 7574 696c 2729 3b0a 636f 6e73 util');.cons
0000003c: 7420 6673 5f72 6561 6464 6972 t.fs.readdir
00000048: 203d 2075 7469 6c2e 7072 6f6d .=.util.prom
00000054: 6973 6966 7928 6673 2e72 6561 isify(fs.rea
00000060: 6464 6972 293b 0a0a 2861 7379 ddir);..(asy
0000006c: 6e63 2028 2920 3d3e 207b 0a20 nc.().=>{..
00000078: 2063 6f6e 7374 2066 696c 6573 .const.files
00000084: 203d 2061 7761 6974 2066 735f .=.await.fs_
00000090: 7265 6164 6469 7228 272e 2729 readdir('.');
0000009c: 3b0a 2020 666f 7220 2866 6e20 ;...for.(fn.
000000a8: 6f66 2066 696c 6573 2920 7b0a of.files).{.
000000b4: 2020 2020 636f 6e73 6f6c 652e ....console.
000000c0: 6c6f 6728 666e 293b 0a20 207d log(fn);...
000000cc: 0a7d 2928 292e 6361 7463 6828 .})().catch(
000000d8: 6572 7220 3d3e 207b 2063 6f6e err.>=.con
000000e4: 736f 6c65 2e65 7272 6f72 2865 sole.error(e
000000f0: 7272 293b 207d 293b rr);.});
```

The `hexy` command was installed as a global command, making it easy to run.

Again, we'll be doing a deep dive into npm in the next chapter. The `hexy` utility is both a Node.js library and a script for printing out these old-style hex dumps.

In the open source world, a perceived need often leads to creating an open source project. The folks who launched the Yarn project saw needs that weren't being addressed by npm and created an alternative package manager tool. They claim a number of advantages over npm, primarily in the area of performance. To learn more about Yarn, go to <https://yarnpkg.com/>.



For every example in this book that uses npm, there is a close equivalent command that uses Yarn.

For npm-packaged command-line tools, there is another, simpler way to use the tool.

Using npx to execute Node.js packaged binaries

Some packages in the npm repository are command-line tools, such as the `hexy` program we looked at earlier. Having to first install such a program before using it is a small hurdle. The sharp-eyed among you will have noticed that `npx` is installed alongside the `node` and `npm` commands when installing Node.js. This tool is meant to simplify running command-line tools from the npm repository by removing the need to first install the package.

The previous example could have been run this way:

```
$ npx hexy --width 12 ls.js
```

Under the covers, `npx` uses `npm` to download the package to a cache directory, unless the package is already installed in the current project directory. Because the package is then in a cache directory, it is only downloaded once.

There are a number of interesting options to this tool; to learn more, go to <https://www.npmjs.com/package/npx>.

We have learned a lot in this section about the command-line tools delivered with Node.js, as well as ran a simple script and HTTP server. Next, we will learn how advances in the JavaScript language affect the Node.js platform.

Advancing Node.js with ECMAScript 2015, 2016, 2017, and beyond

In 2015, the ECMAScript committee released a long-awaited major update of the JavaScript language. The update brought in many new features to JavaScript, such as Promises, arrow functions, and class objects. The language update sets the stage for improvement since it should dramatically improve our ability to write clean, understandable JavaScript code.

The browser makers are adding those much-needed features, meaning the V8 engine is adding those features as well. These features are making their way to Node.js, starting with version 4.x.



To learn about the current status of ES2015/2016/2017/and so on in Node.js, visit <https://nodejs.org/en/docs/es6/>.

By default, only the ES2015, 2016, and 2017 features that V8 considers stable are enabled by Node.js. Further features can be enabled with command-line options. The almost-complete features are enabled with the `--es_staging` option. The website documentation gives more information.

The Node green website (<http://node.green/>) has a table that lists the status of a long list of features in Node.js versions.



The ES2019 language spec is published at <https://www.ecma-international.org/publications/standards/Ecma-262.htm>.

The TC-39 committee does its work on GitHub at <https://github.com/tc39>.

The ES2015 (and later) features make a big improvement to the JavaScript language. One feature, the `Promise` class, should mean a fundamental rethinking of common idioms in Node.js programming. In ES2017, a pair of new keywords, `async` and `await`, simplifies writing asynchronous code in Node.js, which should encourage the Node.js community to further rethink the common idioms of the platform.

There's a long list of new JavaScript features but let's quickly go over the two of them that we'll use extensively.

The first is a lighter-weight function syntax called the arrow function:

```
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) ...; // do something with the error
  else ...; // do something with the data
});
```

This is more than the syntactic sugar of replacing the `function` keyword with the fat arrow. Arrow functions are lighter weight as well as being easier to read. The lighter weight comes at the cost of changing the value of `this` inside the arrow function. In regular functions, `this` has a unique value inside the function. In an arrow function, `this` has the same value as the scope containing the arrow function. This means that, when using an arrow function, we don't have to jump through hoops to bring `this` into the callback function because `this` is the same at both levels of the code.

The next feature is the `Promise` class, which is used for deferred and asynchronous computations. Deferred code execution to implement asynchronous behavior is a key paradigm for Node.js and it requires two idiomatic conventions:

- The last argument to an asynchronous function is a callback function, which is called when an asynchronous execution is to be performed.
- The first argument to the callback function is an error indicator.

While convenient, these conventions have resulted in multilayer code pyramids that can be difficult to understand and maintain:

```
doThis(arg1, arg2, (err, result1, result2) => {
  if (err) ...;
  else {
    // do some work
    doThat(arg2, arg3, (err2, results) => {
      if (err2) ...;
      else {
        doSomethingElse(arg5, err => {
          if (err) ... ;
          else ...;
        });
      }
    });
  });
});
```

You don't need to understand the code; it's just an outline of what happens in practice as we use callbacks. Depending on how many steps are required for a specific task, a code pyramid can get quite deep. Promises will let us unravel the code pyramid and improve reliability because error handling is more straightforward and easily captures all errors.

A `Promise` class is created as follows:

```
function doThis(arg1, arg2) {
  return new Promise((resolve, reject) => {
    // execute some asynchronous code
    if (errorIsDetected) return reject(errorObject);
    // When the process is finished call this:
    resolve(result1, result2);
  });
}
```

Rather than passing in a callback function, the caller receives a `Promise` object. When properly utilized, the preceding pyramid can be coded as follows:

```
doThis(arg1, arg2)
  .then(result => {
    // This can receive only one value, hence to
    // receive multiple values requires an object or array
    return doThat(arg2, arg3);
  })
  .then((results) => {
    return doSomethingElse(arg5);
  })
  .then(() => {
    // do a final something
  })
  .catch(err => {
    // errors land here
  });
}
```

This works because the `Promise` class supports chaining if a `then` function returns a `Promise` object.

The `async/await` feature implements the promise of the `Promise` class to simplify asynchronous coding. This feature becomes active within an `async` function:

```
async function mumble() {
  // async magic happens here
}
```

An `async` arrow function is as follows:

```
const mumble = async () => {
  // async magic happens here
};
```

To see how much of an improvement the `async` function paradigm gives us, let's recode the earlier example as follows:

```
async function doSomething(arg1, arg2, arg3, arg4, arg5) {
  const { result1, result2 } = await doThis(arg1, arg2);
  const results = await doThat(arg2, arg3);
  await doSomethingElse(arg5);
  // do a final something
  return finalResult;
}
```

Again, we don't need to understand the code but just look at its shape. Isn't this a breath of fresh air compared to the nested structure we started with?

The `await` keyword is used with a Promise. It automatically waits for the Promise to resolve. If the Promise resolves successfully, then the value is returned and if it resolves with an error, then that error is thrown. Both handling results and throwing errors are handled in the usual manner.

This example also shows another ES2015 feature: destructuring. The fields of an object can be extracted using the following code:

```
const { value1, value2 } = {  
    value1: "Value 1", value2: "Value 2", value3: "Value3"  
};
```

This demonstrates having an object with three fields but only extracting two of the fields.

To continue our exploration of advances in JavaScript, let's take a look at Babel.

Using Babel to use experimental JavaScript features

The Babel transpiler is the leading tool for using cutting-edge JavaScript features or experimenting with new JavaScript features. Since you've probably never seen the word **transpiler**, it means to rewrite source code from one language to another. It is like a **compiler** in that Babel converts computer source code into another form, but instead of directly executable code, Babel produces JavaScript. That is, it converts JavaScript code into JavaScript code, which may not seem useful until you realize that Babel's output can target older JavaScript releases.

Put more simply, Babel can be configured to rewrite code with ES2015, ES2016, ES2017 (and so on) features into code conforming to the ES5 version of JavaScript. Since ES5 JavaScript is compatible with practically every web browser on older computers, a developer can write their frontend code in modern JavaScript then convert it to execute on older browsers using Babel.



To learn more about Babel, go to <https://babeljs.io>.

The Node Green website makes it clear that Node.js supports pretty much all of the ES2015, 2016, and 2017 features. Therefore, as a practical matter, we no longer need to use Babel for Node.js projects. You may possibly be required to support an older Node.js release and you can use Babel to do so.

For web browsers, there is a much longer time lag between a set of ECMAScript features and when we can reliably use those features in browser-side code. It's not that the web browser makers are slow in adopting new features as the Google, Mozilla, and Microsoft teams are proactive about adopting the latest features. Apple's Safari team seems slow to adopt new features, unfortunately. What's slower, however, is the penetration of new browsers into the fleet of computers in the field.

Therefore, modern JavaScript programmers need to familiarize themselves with Babel.



We're not ready to show example code for these features yet, but we can go ahead and document the setting up of the Babel tool. For further information on setup documentation, visit <http://babeljs.io/docs/setup/> and click on the **CLI** button.

To get a brief introduction to Babel, we'll use it to transpile the scripts we saw earlier to run on Node.js 6.x. In those scripts, we used `async` functions, a feature that is not supported on Node.js 6.x.

In the directory containing `ls.js` and `ls2.js`, type these commands:

```
$ npm install babel-cli \
  babel-plugin-transform-es2015-modules-commonjs \
  babel-plugin-transform-async-to-generator
```

This installs the Babel software, along with a couple of transformation plugins. Babel has a plugin system so that you can enable the transformations required by your project. Our primary goal in this example is converting the `async` functions shown earlier into Generator functions. Generators are a new sort of function introduced with ES2015 that form the foundation for the implementation of `async` functions.

Because Node.js 6.x does not have either the `fs.promises` function or `util.promisify`, we need to make some substitutions to create a file named `ls-old-school.js`:

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

const fs_readdir = dir => {
  return new Promise((resolve, reject) => {
```

```
        fs.readdir(dir, (err, fileList) => {
            if (err) reject(err);
            else resolve(fileList);
        });
    });
};

async function listFiles() {
    try {
        let dir = '.';
        if (process.argv[2]) dir = process.argv[2];
        const files = await fs.readdir(dir);
        for (let fn of files) {
            console.log(fn);
        }
    } catch(err) { console.error(err); }
}
listFiles();
```

We have the same example we looked at earlier, but with a couple of changes. The `fs.readdir` function creates a Promise object then calls `fs.readdir`, making sure to either `reject` or `resolve` the Promise based on the result we get. This is more or less what the `util.promisify` function does.

Because `fs.readdir` returns a Promise, the `await` keyword can do the right thing and wait for the request to either succeed or fail. This code should run as is on Node.js releases, which support `async` functions. But what we're interested in—and the reason why we added the `fs.readdir` function—is how it works on older Node.js releases.

The pattern used in `fs.readdir` is what is required to use a callback-oriented function in an `async` function context.

Next, create a file named `.babelrc`, containing the following:

```
{
  "plugins": [
    "transform-es2015-modules-commonjs",
    "transform-async-to-generator"
  ]
}
```

This file instructs Babel to use the named transformation plugins that we installed earlier. As the name implies, it will transform the `async` functions to generator functions.

Because we installed `babel-cli`, a `babel` command is installed, such that we can type the following:

```
$ ./node_modules/.bin/babel -help
```

To transpile your code, run the following command:

```
$ ./node_modules/.bin/babel ls2-old-school.js -o ls2-babel.js
```

This command transpiles the named file, producing a new file. The new file is as follows:

```
'use strict';

function _asyncToGenerator(fn) { return function () {
  var gen = fn.apply(this, arguments);
  return new Promise(function (resolve, reject)
    { function step(key, arg) { try { var info =
      gen[key](arg); var value = info.value; } catch (error)
        { reject(error); return; } if (info.done) { resolve(value);
        } else { return Promise.resolve(value).then(function (value)
          { step("next", value); }, function (err) { step("throw",
            err); }); } } return step("next"); }; }

const fs = require('fs');

const fs_readdir = dir => {
  return new Promise((resolve, reject) => {
    fs.readdir(dir, (err, fileList) => {
      if (err) reject(err);
      else resolve(fileList);
    });
  });
};

_asyncToGenerator(function* () {
  var dir = '.';
  if (process.argv[2]) dir = process.argv[2];
  const files = yield fs_readdir(dir);
  for (let fn of files) {
    console.log(fn);
  }
})().catch(err => {
  console.error(err);
});
```

This code isn't meant to be easy to read for humans. Instead, it means that you edit the original source file and then convert it for your target JavaScript engine. The main thing to notice is that the transpiled code uses a Generator function (the notation `function*` indicates a generator function) in place of the `async` function and the `yield` keyword in place of the `await` keyword. What a generator function is—and precisely what the `yield` keyword does—is not important; the only thing to note is that `yield` is roughly equivalent to `await` and that the `_asyncToGenerator` function implements functionality similar to `async` functions. Otherwise, the transpiled code is fairly clean and looks rather similar to the original code.

The transpiled script is run as follows:

```
$ nvm use 4
Now using node v4.9.1 (npm v2.15.11)
$ node --version
v4.9.1
$ node ls2-babel
.babelrc
app.js
ls.js
ls2-babel.js
ls2-old-school.js
ls2.js
node_modules
```

In other words, it runs the same as the `async` version but on an older Node.js release. Using a similar process, you can transpile code written with modern ES2015 (and so on) constructions so it can run in an older web browser.

In this section, we learned about advances in the JavaScript language, especially `async` functions, and then learned how to use Babel to use those features on older Node.js releases or in older web browsers.

Summary

You learned a lot in this chapter about installing Node.js using its command-line tools and running a Node.js server. We also breezed past a lot of details that will be covered later in this book, so be patient.

Specifically, we covered downloading and compiling the Node.js source code, installing Node.js—either for development use in your home directory or for deployment in system directories—and installing npm, the de facto standard package manager used with Node.js. We also saw how to run Node.js scripts or Node.js servers. We then took a look at the new features in ES2015, 2016, and 2017. Finally, we looked at how to use Babel to implement those features in your code.

Now that we've seen how to set up a development environment, we're ready to start working on implementing applications with Node.js. The first step is to learn the basic building blocks of Node.js applications and modules, meaning taking a more careful look at Node.js modules, how they are used, and how to use npm to manage application dependencies. We will cover all of that in the next chapter.

3

Exploring Node.js Modules

Modules and packages are the building blocks for breaking down your application into smaller pieces. A module encapsulates some functionality, primarily JavaScript functions, while hiding implementation details and exposing an API for the module. Modules can be distributed by third parties and installed for use by our modules. An installed module is called a package.

The npm package repository is a huge library of modules that's available for all Node.js developers to use. Within that library are hundreds of thousands of packages you can be used to accelerate the development of your application.

Since modules and packages are the building blocks of your application, understanding how they work is vital to your success with Node.js. By the end of this chapter, you will have a solid grounding in both CommonJS and ES6 modules, how to structure the modules in an application, how to manage dependencies on third-party packages, and how to publish your own packages.

In this chapter, we will cover the following topics:

- Definitions of all types of Node.js modules and how to structure both simple and complex modules
- Using CommonJS and ES2015/ES6 modules and when to use each
- Understanding how Node.js finds modules and installed packages, so you can better structure your application
- Using the npm package management system (and Yarn) to manage application dependencies, to publish packages, and to record administrative scripts for the project

So, let's get on with it.

Defining a Node.js module

Modules are the basic building blocks for constructing Node.js applications. A Node.js module encapsulates functions, hiding details inside a well-protected container, and exposing an explicitly declared API.

When Node.js was created, the ES6 module system, of course, did not yet exist. Ryan Dahl, therefore, based on the Node.js module system on the CommonJS standard. The examples we've seen so far are modules written to that format. With ES2015/ES2016, a new module format was created for use with all JavaScript implementations. This new module format is used by both front-end engineers in their in-browser JavaScript code and by Node.js engineers, and for any other JavaScript implementation.

Because ES6 modules are now the standard module format, the Node.js **Technical Steering Committee (TSC)** committed to first-class support for ES6 modules alongside the CommonJS format. Starting with Node.js 14.x, the Node.js TSC delivered on that promise.

Every source file used in an application on the Node.js platform is a *module*. Over the next few sections, we'll examine the different types of modules, starting with the CommonJS module format.



Throughout this book, we'll identify traditional Node.js modules as CommonJS modules, and the new module format as ES6 modules.

To start our exploration of Node.js modules, we must, of course, start at the beginning.

Examining the traditional Node.js module format

We already saw CommonJS modules in action in the previous chapter. It's now time to see what they are and how they work.

In the `ls.js` example in Chapter 2, *Setting Up Node.js*, we wrote the following code to pull in the `fs` module, giving us access to its functions:

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

The `require` function is given a *module identifier*, and it searches for the module named by that identifier. If found, it loads the module definition into the Node.js runtime and making its functions available. In this case, the `fs` object contains the code (and data) exported by the `fs` module. The `fs` module is part of the Node.js core and provides filesystem functions.

By declaring `fs` as `const`, we have a little bit of assurance against making coding mistakes. We could mistakenly assign a value to `fs`, and then the program would fail, but as a `const` we know the reference to the `fs` module will not be changed.

The file, `ls.js`, is itself a module because every source file we use on Node.js is a module. In this case, it does not export anything but is instead a script that consumes other modules.

What does it mean to say the `fs` object contains the code exported by the `fs` module? In a CommonJS module, there is an object, `module`, provided by Node.js, with which the module's author describes the module. Within this object is a field, `module.exports`, containing the functions and data exported by the module. The return value of the `require` function is the object. The object is the interface provided by the module to other modules. Anything added to the `module.exports` object is available to other pieces of code, and everything else is hidden. As a convenience, the `module.exports` object is also available as `exports`.



The `module` object contains several fields that you might find useful. Refer to the online Node.js documentation for details.

Because `exports` is an alias of `module.exports`, the following two lines of code are equivalent:

```
exports.funcName = function(arg, arg1) { ... };
module.exports.funcName = function(arg, arg2) { .. };
```

Whether you use `module.exports` or `exports` is up to you. However, do not ever do anything like the following:

```
exports = function(arg, arg1) { ... };
```

Any assignment to `exports` will break the alias, and it will no longer be equivalent to `module.exports`. Assignments to `exports.something` are okay, but assigning to `exports` will cause failure. If your intent is to assign a single object or function to be returned by `require`, do this instead:

```
module.exports = function(arg, arg1) { ... };
```

Some modules do export a single function because that's how the module author envisioned delivering the desired functionality.

When we said `ls.js` does not export anything, we meant that `ls.js` did not assign anything to `module.exports`.

To give us a brief example, let's create a simple module, named `simple.js`:

```
var count = 0;
exports.next = function() { return ++count; };
exports.hello = function() {
  return "Hello, world!";
};
```

We have one variable, `count`, which is not attached to the `exports` object, and a function, `next`, which is attached. Because `count` is not attached to `exports`, it is private to the module.

Any module can have private implementation details that are not exported and are therefore not available to any other code.

Now, let's use the module we just wrote:

```
$ node
> const s = require('./simple');
undefined
> s.hello();
'Hello, world!'
> s.next();
1
> s.next();
2
> s.next();
3
```

```
> console.log(s.count);
undefined
undefined
>
```

The `exports` object in the module is the object that is returned by `require('./simple')`. Therefore, each call to `s.next` calls the `next` function in `simple.js`. Each returns (and increments) the value of the local variable, `count`. An attempt to access the private field, `count`, shows it's unavailable from outside the module.

This is how Node.js solves the global object problem of browser-based JavaScript. The variables that look like they are global variables are only global to the module containing the variable. These variables are not visible to any other code.



The Node.js package format is derived from the CommonJS module system (<http://commonjs.org>). When developed, the CommonJS team aimed to fill a gap in the JavaScript ecosystem. At that time, there was no standard module system, making it trickier to package JavaScript applications. The `require` function, the `exports` object, and other aspects of Node.js modules come directly from the CommonJS Modules/1.0 spec.

The `module` object is a global-to-the-module object injected by Node.js. It also injects two other variables: `__dirname` and `__filename`. These are useful for helping code in a module know where it is located in the filesystem. Primarily, this is used for loading other files using a path relative to the module's location.

For example, one can store assets like CSS or image files in a directory relative to the module. An app framework can then make the files available via an HTTP server. In Express, we do so with this code snippet:

```
app.use('/assets/vendor/jquery', express.static(
  path.join(__dirname, 'node_modules', 'jquery')));
```

This says that HTTP requests on the `/assets/vendor/jquery` URL are to be handled by the static handler in Express, from the contents of a directory relative to the directory containing the module. Don't worry about the details because we'll discuss this more carefully in a later chapter. Just notice that `__dirname` is useful to calculate a filename relative to the location of the module source code.

To see it in action, create a file named `dirname.js` containing the following:

```
console.log(`dirname: ${__dirname}`);
console.log(`filename: ${__filename}`);
```

This lets us see the values we receive:

```
$ node dirname.js
dirname: /home/david/Chapter03
filename: /home/david/Chapter03 dirname.js
```

Simple enough, but as we'll see later these values are not directly available in ES6 modules.

Now that we've got a taste for CommonJS modules, let's take a look at ES2015 modules.

Examining the ES6/ES2015 module format

ES6 modules are a new module format designed for all JavaScript environments. While Node.js has always had a good module system, browser-side JavaScript has not. That meant the browser-side community had to use non-standardized solutions. The CommonJS module format was one of those non-standard solutions, which was borrowed for use in Node.js. Therefore, ES6 modules are a big improvement for the entire JavaScript world, by getting everyone on the same page with a common module format and mechanisms.

An issue we have to deal with is the file extension to use for ES6 modules. Node.js needs to know whether to parse using the CommonJS or ES6 module syntax. To distinguish between them, Node.js uses the file extension `.mjs` to denote ES6 modules, and `.js` to denote CommonJS modules. However, that's not the entire story since Node.js can be configured to recognize the `.js` files as ES6 modules. We'll give the exact particulars later in this chapter.

The ES6 and CommonJS modules are conceptually similar. Both support exporting data and functions from a module, and both support hiding implementation inside a module. But they are very different in many practical ways.

Let's start with defining an ES6 module. Create a file named `simple2.mjs` in the same directory as the `simple.js` example that we looked at earlier:

```
let count = 0;
export function next() { return ++count; }
function squared() { return Math.pow(count, 2); }
```

```
export function hello() {
  return "Hello, world!";
}
export default function() { return count; }
export const meaning = 42;
export let nocount = -1;
export { squared };
```

This is similar to `simple.js` but with a few additions to demonstrate further features. As before `count` is a private variable that isn't exported, and `next` is an exported function that increments `count`.

The `export` keyword declares what is being exported from an ES6 module. In this case, we have several exported functions and two exported variables. The `export` keyword can be put in front of any top-level declaration, such as variable, function, or class declarations:

```
export function next() { .. }
```

The effect of this is similar to the following:

```
module.exports.next = function() { .. }
```

The intent of both is essentially the same: to make a function or other object available to code outside the module. But instead of explicitly creating an object, `module.exports`, we're simply declaring what is to be exported. A statement such as `export function next()` is a named export, meaning the exported function (as here) or object has a name, and that code outside the module uses that name to access the object. As we see here, named exports can be functions or objects, and they may also be class definitions.

The *default export* from a module, defined with `export default`, can be done once per module. The default export is what code outside the module accesses when using the module object itself, rather than when using one of the exports from the module.

You can also declare something, such as the `squared` function, and then export it later.

Now let's see how to use the ES2015 module. Create a `simplerdem0.mjs` file with the following:

```
import * as simple2 from './simple2.mjs';

console.log(simple2.hello());
console.log(` ${simple2.next()} ${simple2.squared()}`);
console.log(` ${simple2.next()} ${simple2.squared()}`);
console.log(` ${simple2.default()} ${simple2.squared()}`);
console.log(` ${simple2.next()} ${simple2.squared()}`);
console.log(` ${simple2.next()} ${simple2.squared()}`);
console.log(` ${simple2.next()} ${simple2.squared()}`);
console.log(simple2.meaning);
```

The `import` statement does what it says: it imports objects exported from a module. Because it uses the `import * as foo` syntax, it imports everything from the module, attaching everything to an object, in this case named `simple2`. This version of the `import` statement is most similar to a traditional Node.js `require` statement because it creates an object with fields containing the objects exported from the module.

This is how the code executes:

```
$ node simplerdem0.mjs
Hello, world!
1 1
2 4
2 4
3 9
4 16
5 25
42
```

In the past, the ES6 module format was hidden behind an option flag, `--experimental-module`, but as of Node.js 13.2 that flag is no longer required. Accessing the `default` export is accomplished by accessing the field named `default`. Accessing an exported value, such as the `meaning` field, is done without parentheses because it is a value and not a function.

Now to see a different way to import objects from a module, create another file, named `simplerdem02.mjs`, containing the following:

```
import {
  default as simple, hello, next, meaning
} from './simple2.mjs';
console.log(hello());
```

```
console.log(next());
console.log(next());
console.log(simple());
console.log(next());
console.log(next());
console.log(next());
console.log(meaning);
```

In this case, the import is treated similarly to an ES2015 destructuring assignment. With this style of import, we specify exactly what is to be imported, rather than importing everything. Furthermore, instead of attaching the imported things to a common object, and therefore executing `simple2.next()`, the imported things are executed using their simple name, as in `next()`.

The import for `default as simple` is the way to declare an alias of an imported thing. In this case, it is necessary so that the default export has a name other than `default`.

Node.js modules can be used from the ES2015 `.mjs` code. Create a file named `ls.mjs` containing the following:

```
import { promises as fs } from 'fs';

async function listFiles() {
    const files = await fs.readdir('.');
    for (const file of files) {
        console.log(file);
    }
}

listFiles().catch(err => { console.error(err); });
```

This is a reimplementation of the `ls.js` example in Chapter 2, *Setting Up Node.js*. In both cases, we're using the `promises` submodule of the `fs` package. To do this with the `import` statement, we access the `promises` export from the `fs` module, and use the `as` clause to rename `fs.promises` to `fs`. This way we can use an `async` function rather than deal with callbacks.

Otherwise, we have an `async` function, `listFiles`, that performs filesystem operations to read filenames from a directory. Because `listFiles` is `async`, it returns a `Promise`, and we must catch any errors using a `.catch` clause.

Executing the script gives the following:

```
$ node ls.mjs
ls.mjs
module1.js
module2.js
simple.js
simple2.mjs
simpledemo.mjs
simplesdemo2.mjs
```

The last thing to note about ES2015 module code is that the `import` and `export` statements must be top-level code. Try putting an `export` inside a simple block like this:

```
{
  export const meaning = 42;
}
```

That innocent bit of code results in an error:

```
$ node badexport.mjs
file:///home/david/Chapter03/badexport.mjs:2
  export const meaning = 42;
  ^^^^^^

SyntaxError: Unexpected token 'export'
  at Loader.moduleStrategy
(internal/modules/esm/translators.js:83:18)
  at async link (internal/modules/esm/module_job.js:36:21)
```

While there are a few more details about the ES2015 modules, these are their most important attributes.

Remember that the objects injected into CommonJS modules are not available to ES6 modules. The `__dirname` and `__filename` objects are the most important, since there are many cases where we compute a filename relative to the currently executing module. Let us explore how to handle that issue.

Injected objects in ES6 modules

Just as for CommonJS modules, certain objects are injected into ES6 modules. Furthermore, ES6 modules do not receive the `__dirname`, and `__filename` objects or other objects that are injected into CommonJS modules.

The `import.meta` meta-property is the only value injected into ES6 modules. In Node.js it contains a single field, `url`. This is the URL from which the currently executing module was loaded.

Using `import.meta.url`, we can compute `__dirname` and `__filename`.

Computing the missing `__dirname` variable in ES6 modules

If we make a duplicate of `dirname.js` as `dirname.mjs`, so it will be interpreted as an ES6 module, we get the following:

```
$ cp dirname.js dirname.mjs
$ node dirname.mjs
console.log(`dirname: ${__dirname}`);
^
ReferenceError: __dirname is not defined
  at file:///home/david/Chapter03 dirname.mjs:1:25
  at ModuleJob.run (internal/modules/esm/module_job.js:109:37)
  at async Loader.import (internal/modules/esm/loader.js:132:24)
```

Since `__dirname` and `__filename` are not part of the JavaScript specification, they are not available within ES6 modules. Enter the `import.meta.url` object, from which we can compute `__dirname` and `__filename`. To see it in action, create a `dirname-fixed.mjs` file containing the following:

```
import { fileURLToPath } from 'url';
import { dirname } from 'path';

console.log(`import.meta.url: ${import.meta.url}`);

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

console.log(`dirname: ${__dirname}`);
console.log(`filename: ${__filename}`);
```

We are importing a couple of useful functions from the `url` and `path` core packages. While we could take the `import.meta.url` object and do our own computations, these functions already exist. The computation is to extract the pathname portion of the module URL, to compute `__filename`, and then use `dirname` to compute `__dirname`.

```
$ node dirname-fixed.mjs
import.meta.url: file:///home/david/Chapter03 dirname-fixed.mjs
dirname: /home/david/Chapter03
filename: /home/david/Chapter03 dirname-fixed.mjs
```

And we see the `file://` URL of the module, and the computed values for `__dirname` and `__filename` using the built-in core functions.

We've talked about both the CommonJS and ES6 module formats, and now it's time to talk about using them together in an application.

Using CommonJS and ES6 modules together

Node.js supports two module formats for JavaScript code: the CommonJS format originally developed for Node.js, and the new ES6 module format. The two are conceptually similar, but there are many practical differences. Because of this, we will face situations of using both in the same application and will need to know how to proceed.

First is the question of file extensions and recognizing which module format to use. The ES6 module format is used in the following situations:

- Files where the filename ends in `.mjs`.
- If the `package.json` has a field named `type` with the value `module`, then filenames ending with `.js`.
- If the `node` binary is executed with the `--input-type=module` flag, then any code passed through the `--eval` or `--print` argument, or piped in via STDIN (the standard input), is interpreted as ES6 module code.

That's fairly straight-forward. ES6 modules are in files named with the `.mjs` extension, unless you've declared in the `package.json` that the package defaults to ES6 modules, in which case files named with the `.js` extension are also interpreted as ES6 modules.

The CommonJS module format is used in the following situations:

- Files where the file name ends in `.cjs`.
- If the `package.json` does not contain a `type` field, or if it contains a `type` field with a value of `commonjs`, the filenames will end with `.js`.
- If the `node` binary is executed with the `--input-type` flag or with the `--type-type=commonjs` flag, then any code passed through the `--eval` or `--print` argument, or piped in via `STDIN` (the standard input), is interpreted as CommonJS module code.

Again this is straight-forward, with Node.js defaulting to CommonJS modules for the `.js` files. If the package is explicitly declared to default to CommonJS modules, then Node.js will interpret the `.js` files as CommonJS.

The Node.js team strongly recommends that package authors include a `type` field in `package.json`, even if the type is `commonjs`.

Consider a `package.json` with this declaration:

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

This, of course, informs Node.js that the package defaults to ES6 modules. Therefore, this command interprets the module as an ES6 module:

```
$ node my-module.js
```

This command will do the same, even without the `package.json` entry:

```
$ node --input-type=module my-module.js
```

If instead, the `type` field had the `commonjs`, or the `--input-type` flag specified as `commonjs`, or if both those were completely missing, then `my-module.js` would be interpreted as a CommonJS module.

These rules also apply to the `import` statement, the `import()` function, and the `require()` function. We will cover those commands in more depth in a later section. In the meantime, let's learn how the `import()` function partly resolves the inability to use ES6 modules in a CommonJS module.

Using ES6 modules from CommonJS using import()

The `import` statement in ES6 modules is a statement, and not a function like `require()`. This means that `import` can only be given a static string, and you cannot compute the module identifier to import. Another limitation is that `import` only works in ES6 modules, and therefore a CommonJS module cannot load an ES6 module. Or, can it?

Since the `import()` function is available in both CommonJS and ES6 modules, that means we should be able to use it to import ES6 modules in a CommonJS module.

To see how this works, create a file named `simple-dynamic-import.js` containing the following:

```
async function simpleFn() {
  const simple2 = await import('./simple2.mjs');
  console.log(simple2.hello());
  console.log(simple2.next());
  console.log(simple2.next());
  console.log(`count = ${simple2.default()}`);
  console.log(`Meaning: ${simple2.meaning}`);
}

simpleFn().catch(err => { console.error(err); });
```

This is a CommonJS module that's using an ES6 module we created earlier. It simply calls a few of the functions, nothing exciting except that it is using an ES6 module when we said earlier `import` only works in ES6 modules. Let's see this module in action:

```
$ node simple-dynamic-import.js
Hello, world!
1
2
count = 2
Meaning: 42
```

This is a CommonJS module successfully executing code contained in an ES6 module simply by using `import()`.

Notice that `import()` was called not in the global scope of the module, but inside an `async` function. As we saw earlier, the ES6 module keyword statements like `export` and `import` must be called in the global scope. However, `import()` is an asynchronous function, limiting our ability to use it in the global scope.

The `import` statement is itself an asynchronous process, and by extension the `import()` function is asynchronous, while the Node.js `require()` function is synchronous.

In this case, we executed `import()` inside an `async` function using the `await` keyword. Therefore, even if `import()` were used in the global scope, it would be tricky getting a global-scope variable to hold the reference to that module. To see, why let's rewrite that example as `simple-dynamic-import-fail.js`:

```
const simple2 = import('./simple2.mjs');
console.log(simple2);
console.log(simple2.hello());
console.log(simple2.next());
console.log(simple2.next());
console.log(`count = ${simple2.default()}`);
console.log(`Meaning: ${simple2.meaning}`);
```

It's the same code but running in the global scope. In the global scope, we cannot use the `await` keyword, so we should expect that `simple2` will contain a pending Promise. Running the script gives us this failure:

```
$ node simple-dynamic-import-fail.js
Promise { <pending> }
/home/david/Chapter03/simple-dynamic-import-fail.js:4
  console.log(simple2.hello());
  ^
TypeError: simple2.hello is not a function
  at Object.<anonymous> (/home/david/Chapter03/simple-dynamic-import-
fail.js:4:21)
  at Module._compile (internal/modules/cjs/loader.js:1139:30)
  at Object.Module._extensions..js
(internal/modules/cjs/loader.js:1159:10)
  at Module.load (internal/modules/cjs/loader.js:988:32)
  at Function.Module._load (internal/modules/cjs/loader.js:896:14)
  at Function.executeUserEntryPoint [as runMain]
(internal/modules/run_main.js:71:12)
  at internal/main/run_main_module.js:17:47
```

We see that `simple2` does indeed contain a pending Promise, meaning that `import()` has not yet finished. Since `simple2` does not contain a reference to the module, attempts to call the exported function fail.

The best we could do in the global scope is to attach the `.then` and `.catch` handlers to the `import()` function call. That would wait until the Promise transitions to either a success or failure state, but the loaded module would be inside the callback function. We'll see this example later in the chapter.

Let's now see how modules hide implementation details.

Hiding implementation details with encapsulation in CommonJS and ES6 modules

We've already seen a couple of examples of how modules hide implementation details with the `simple.js` example and the programs we examined in Chapter 2, *Setting up Node.js*. Let's take a closer look.

Node.js modules provide a simple encapsulation mechanism to hide implementation details while exposing an API. To review, in CommonJS modules the exposed API is assigned to the `module.exports` object, while in ES6 modules the exposed API is declared with the `export` keyword. Everything else inside a module is not available to code outside the module.

In practice, CommonJS modules are treated as if they were written as follows:

```
(function(exports, require, module, __filename, __dirname) {  
  // Module code actually lives in here  
});
```

Thus, everything within the module is contained within an anonymous private namespace context. This is how the global object problem is resolved: everything in a module that looks global is actually contained within a private context. This also explains how the injected variables are actually injected into the module. They are parameters to the function that creates the module.

The other advantage is code safety. Because the private code in a module is stashed in a private namespace, it is impossible for code outside the module to access the private code or data.

Let's take a look at a practical demonstration of the encapsulation. Create a file named `module1.js`, containing the following:

```
const A = "value A";  
const B = "value B";  
exports.values = function() {  
  return { A: A, B: B };  
}
```

Then, create a file named `module2.js`, containing the following:

```
const util = require('util');
const A = "a different value A";
const B = "a different value B";
const m1 = require('./module1');
console.log(`A=${A} B=${B} values=${util.inspect(m1.values())}`);
console.log(`${m1.A} ${m1.B}`);
const vals = m1.values();
vals.B = "something completely different";
console.log(util.inspect(vals));
console.log(util.inspect(m1.values()));
```

Using these two modules we can see how each module is its own protected bubble.

Then run it as follows:

```
$ node module2.js
A=a different value A B=a different value B values={ A: 'value A', B:
  'value B' }
undefined undefined
{ A: 'value A', B: 'something completely different' }
{ A: 'value A', B: 'value B' }
```

This artificial example demonstrates encapsulation of the values in `module1.js` from those in `module2.js`. The `A` and `B` values in `module1.js` don't overwrite `A` and `B` in `module2.js` because they're encapsulated within `module1.js`. The `values` function in `module1.js` does allow code in `module2.js` access to the values; however, `module2.js` cannot directly access those values. We can modify the object `module2.js` received from `module1.js`. But doing so does not change the values within `module1.js`.

In Node.js modules can also be data, not just code.

Using JSON modules

Node.js supports using `require('./path/to/file-name.json')` to import a JSON file in a CommonJS module. It is equivalent to the following code:

```
const fs = require('fs');
module.exports = JSON.parse(
  fs.readFileSync('/path/to/file-name.json', 'utf8'));
```

That is, the JSON file is read synchronously, and the text is parsed as JSON. The resultant object is available as the object exported from the module. Create a file named `data.json`, containing the following:

```
{  
  "hello": "Hello, world!",  
  "meaning": 42  
}
```

Now create a file named `showdata.js` containing the following:

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

It will execute as follows:

```
$ node showdata.js  
{ hello: 'Hello, world!', meaning: 42 }
```

The `console.log` function outputs information to the Terminal. When it receives an object, it prints out the object content like this. And this demonstrates that `require` correctly read the JSON file since the resulting object matched the JSON.

In an ES6 module, this is done with the `import` statement and requires a special flag. Create a file named `showdata-es6.mjs` containing the following:

```
import * as data from './data.json';  
console.log(data);
```

So far that is equivalent to the CommonJS version of this script, but using `import` rather than `require`.

```
$ node --experimental-modules --experimental-json-modules showdata-  
es6.mjs  
(node:12772) ExperimentalWarning: The ESM module loader is  
experimental.  
[Module] { default: { hello: 'Hello, world!', meaning: 42 } }
```

Currently using `import` to load a JSON file is an experimental feature. Enabling the feature requires these command-line arguments, causing this warning to be printed. We also see that instead of `data` being an anonymous object, it is an object with the type `Module`.

Now let's look at how to use ES6 modules on some older Node.js releases.

Supporting ES6 modules on older Node.js versions

Initially, ES6 module support was an experimental feature in Node.js 8.5 and became a fully supported feature in Node.js 14. With the right tools, we can use it on earlier Node.js implementations.



For an example of using Babel to transpile ES6 code for older Node.js versions, see <https://blog.revillweb.com/using-es2015-es6-modules-with-babel-6-3ffc0870095b>.

The better method of using ES6 modules on Node.js 6.x is the `esm` package. Simply do the following:

```
$ nvm install 6
Downloading and installing node v6.14.1...
Downloading
https://nodejs.org/dist/v6.14.1/node-v6.14.1-darwin-x64.tar.xz...
#####
## 100.0%
Computing checksum with shasum -a 256
Checksums matched!
Now using node v6.14.1 (npm v3.10.10)
$ nvm use 6
Now using node v6.14.1 (npm v3.10.10)
$ npm install esm
... npm output
$ node --require esm simpledemo.mjs
Hello, world!
1 1
2 4
2 4
3 9
4 16
5 25
42
```

There are two ways to use this module:

- In a CommonJS module, invoke `require('esm')`.
- On the command line, use `--require esm`, as shown here.

In both cases, the effect is the same, to load the `esm` module. This module only needs to be loaded once, and we do not have to call any of its methods. Instead `esm` retrofits ES6 module support into the Node.js runtime, and is compatible with version 6.x and later.

So, we can use this module to retrofit ES6 module support; it does not retrofit other features such as `async` functions. Successfully executing the `ls.mjs` example requires support for both the `async` functions and arrow functions. Since Node.js 6.x does not support either, the `ls.mjs` example will load correctly, but will still fail because it uses other unsupported features.

```
$ node --version
v6.14.1
$ node --require esm ls.mjs
/Users/David/chap03/ls.mjs:5
(async () => {
^

SyntaxError: Unexpected token (
  at exports.runInThisContext (vm.js:53:16)
  at Module._compile (module.js:373:25)
```

It is, of course, possible to use Babel in such cases to convert the full set of ES2015+ features to run on older Node.js releases.



For more information about `esm`, see:
<https://medium.com/web-on-the-edge/es-modules-in-node-today-32cff914e4b>. The article describes an older release of the `esm` module, at the time named `@std/esm`.

The current documentation for the `esm` package is available
at: <https://www.npmjs.com/package/esm>.

In this section, we've learned about how to define a Node.js module and various ways to use both CommonJS and ES6 modules. But we've left out some very important things: what is the module identifier and all the ways to locate and use modules. In the next section, we cover these topics.

Finding and loading modules using require and import

In the course of learning about modules for Node.js, we've used the `require` and `import` features without going into detail about how modules are found and all the options available. The algorithm for finding Node.js modules is very flexible. It supports finding modules that are siblings of the currently executing module, or have been installed local to the current project, or have been installed globally.

For both `require` and `import`, the command takes a *module identifier*. The algorithm Node.js uses is in charge of resolving the module identifier into a file containing the module, so that Node.js can load the module.



The official documentation for this is in the Node.js documentation, at <https://nodejs.org/api/modules.html>.

The official documentation for ES6 modules also discusses how the algorithm differs, at <https://nodejs.org/api/esm.html>.

Understanding the module resolution algorithm is one key to success with Node.js. This algorithm determines how best to structure the code in a Node.js application. While debugging problems with loading the correct version of a given package, we need to know how Node.js finds packages.

First, we must consider several types of modules, starting with the simple file modules we've already used.

Understanding File modules

The CommonJS and ES6 modules we've just looked at are what the Node.js documentation describes as a **file module**. Such modules are contained within a single file, whose filename ends with `.js`, `.cjs`, `.mjs`, `.json`, or `.node`. The latter are compiled from C or C++ source code, or even other languages such as Rust, while the former are, of course, written in JavaScript or JSON.

The *module identifier* of a file module must start with `./` or `../`. This signals Node.js that the module identifier refers to a local file. As should already be clear, this module identifier refers to a pathname relative to the currently executing module.

It is also possible to use an absolute pathname as the module identifier. In a CommonJS module, such an identifier might be `/path/to/some/directory/my-module.js`. In an ES6 module, since the module identifier is actually a URL, then we must use a `file://` URL like `file:///path/to/some/directory/my-module.mjs`. There are not many cases where we would use an absolute module identifier, but the capability does exist.

One difference between CommonJS and ES6 modules is the ability to use extensionless module identifiers. The CommonJS module loader allows us to do this, which you should save as `extensionless.js`:

```
const simple = require('./simple');

console.log(simple.hello());
console.log(`\$${simple.next()}`);
console.log(`\$${simple.next()}`);
```

This uses an extension-less module identifier to load a module we've already discussed, `simple.js`:

```
$ node ./extensionless
Hello, world!
1
2
```

And we can run it with the `node` command using an extension-less module identifier.

But if we specify an extension-less identifier for an ES6 module:

```
$ node ./simpledo2
internal/modules/cjs/loader.js:964
  throw err;
^

Error: Cannot find module '/home/david/Chapter03/simpledo2'
  at Function.Module._resolveFilename
(internal/modules/cjs/loader.js:961:17)
  at Function.Module._load (internal/modules/cjs/loader.js:854:27)
  at Function.executeUserEntryPoint [as runMain]
(internal/modules/run_main.js:71:12)
  at internal/main/run_main_module.js:17:47 {
  code: 'MODULE_NOT_FOUND',
  requireStack: []
}
```

We get the error message making it clear that Node.js could not resolve the file name. Similarly, in an ES6 module, the file name given to the `import` statement must have the file extension.

Next, let's discuss another side effect of ES6 module identifiers being a URL.

The ES6 import statement takes a URL

The module identifier in the ES6 `import` statement is a URL. There are several important considerations.

Since Node.js only supports the `file://` URLs, we're not allowed to retrieve a module over from a web server. There are obvious security implications, and the corporate security team would rightfully get anxious if modules could be loaded from `http://` URLs.

Referencing a file with an absolute pathname must use the `file:///path/to/file.ext` syntax, as mentioned earlier. This is different from `require`, where we would use `/path/to/file.ext` instead.

Since `?` and `#` have special significance in a URL, they also have special significance to the `import` statement, as in the following example:

```
import './module-name.mjs?query=1'
```

This loads the module named `module-name.mjs` with a query string containing `query=1`. By default, this is ignored by the Node.js module loader, but there is an experimental loader hook feature by which you can do something with the module identifier URL.

The next type of module to consider is those baked into Node.js, the core modules.

Understanding the Node.js core modules

Some modules are pre-compiled into the Node.js binary. These are the core Node.js modules documented on the Node.js website at <https://nodejs.org/api/index.html>.

They start out as source code within the Node.js build tree. The build process compiles them into the binary so that the modules are always available.

We've already seen how the core modules are used. In a CommonJS module, we might use the following:

```
const http = require('http');
const fs = require('fs').promises;
```

And the equivalent in an ES6 module would be as follows:

```
import http from 'http';
import { promises as fs } from 'fs';
```

In both cases, we're loading the `http` and `fs` core modules that would then be used by other code in the module.

Moving on, we will next talk about more complex module structures.

Using a directory as a module

We commonly organize stuff into a directory structure. The stuff here is a technical term referring to internal file modules, data files, template files, documentation, tests, assets, and more. Node.js allows us to create an entry-point module into such a directory structure.

For example, with a module identifier like `./some-library` that refers to a directory, then there must be a file named `index.js`, `index.cjs`, `index.mjs`, or `index.node` in the directory. In such a case, the module loader loads the appropriate `index` module even though the module identifier did not reference a full pathname. The pathname is computed by appending the file it finds in the directory.

One common use for this is that the `index` module provides an API for a library stored in the directory and that other modules in the directory contain what's meant to be private implement details.



This may be a little confusing because the word *module* is being overloaded with two meanings. In some cases, a module is a file, and in other cases, a module is a directory containing one or more file modules.

While overloading the word *module* this way might be a little confusing, it's going to get even more so as we consider the packages we install from other sources.

Comparing installed packages and modules

Every programming platform supports the distribution of libraries or packages that are meant to be used in a wide array of applications. For example, where the Perl community has CPAN, the Node.js community has the npm registry. A Node.js *installed package* is the same as we just described as a *folder as a module*, in that the package format is simply a directory containing a `package.json` file along with the code and other files comprising the package.



There is the same risk of confusion caused by overloading the word *module* since an installed package is typically the same as the *directories as modules* concept just described. Therefore, it's useful to refer to an installed package with the word *package*.

The `package.json` file describes the package. A minimal set of fields are defined by Node.js, specifically as follows:

```
{ "name" : "some-library",
  "main" : "./lib/some-library.js" }
```

The `name` field gives the name of the package. If the `main` field is present, it names the JavaScript file to use instead of `index.js` to load when the package is loaded. The package manager applications like npm and Yarn support many more fields in `package.json`, which they use to manage dependencies and versions and everything else.

If there is no `package.json`, then Node.js will look for either `index.js` or `index.node`. In such a case, `require('some-library')` will load the file module in `/path/to/some-library/index.js`.

Installed packages are kept in a directory named `node_modules`. When JavaScript source code has `require('some-library')` or `import 'some-library'`, Node.js searches through one or more `node_modules` directories to find the named package.

Notice that the module identifier, in this case, is just the package name. This is different from the file and directory module identifiers we studied earlier since both those are pathnames. In this case, the module identifier is somewhat abstract, and that's because Node.js has an algorithm for finding packages within the nested structure of the `node_modules` directories.

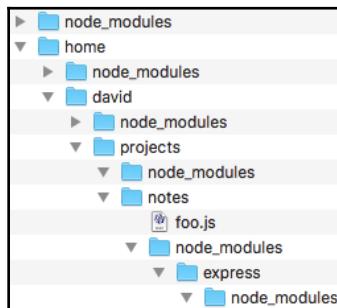
To understand how that works, we need a deeper dive into the algorithm.

Finding the installed package in the file system

One key to why the Node.js package system is so flexible is the algorithm used to search for packages.

For a given `require`, `import()`, or `import` statement, Node.js searches upward in the file system from the directory containing the statement. It is looking for a directory named `node_modules` containing a module satisfying the module identifier.

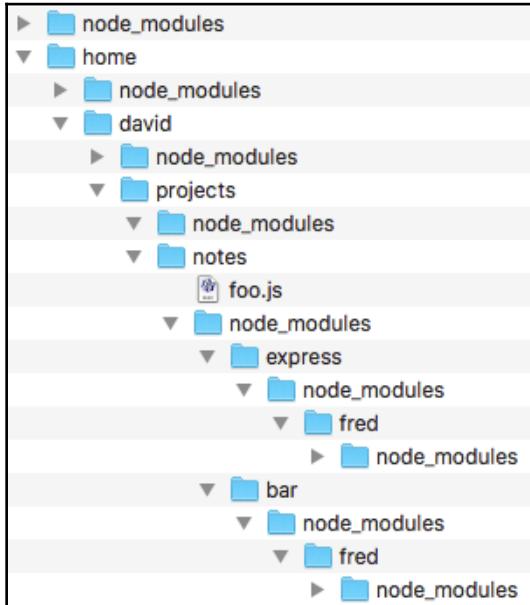
For example, with a source file named `/home/david/projects/notes/foo.js` and a `require` or `import` statement requesting the module identifier `bar.js`, Node.js tries the following options:



As just said, the search starts at the same level of the file system as `foo.js`. Node.js will look either for a file module named `bar.js` or else a directory named `bar.js` containing a module as described earlier in *Using a Directory as a module*. Node.js will check for this package in the `node_modules` directory next to `foo.js` and in every directory above that file. It will not, however, descend into any directory such as `express` or `express/node_modules`. The traversal only moves upward in the file system, not downward.

While some of the third-party packages have a name ending in `.js`, the vast majority do not. Therefore, we will typically use `require('bar')`. Also typically the 3rd party installed packages are delivered as a directory containing a `package.json` file and some JavaScript files. Therefore, in the typical case, the package module identifier would be `bar`, and Node.js will find a directory named `bar` in one of the `node_modules` directories and access the package from that directory.

This act of searching upward in the file system means Node.js supports the nested installation of packages. A Node.js package that in turn depends on other modules that will have its own `node_modules` directory; that is, the `bar` package might depend on the `fred` package. The package manager application might install `fred` as `/home/david/projects/notes/node_modules/bar/node_modules/fred`:



In such a case, when a JavaScript file in the `bar` package uses `require('fred')` its search for modules starts in `/home/david/projects/notes/node_modules/bar/node_modules`, where it will find the `fred` package. But if the package manager detects that other packages used by `notes` also use the `fred` package, the package manager will install it as `/home/david/projects/notes/node_modules/fred`.

Because the search algorithm traverses the file system upwards, it will find `fred` in either location.

The last thing to note is that this nesting of `node_modules` directories can be arbitrarily deep. While the package manager applications try to install packages in a flat hierarchy, it may be necessary to nest them deeply.

One reason for doing so is to enable using two or more versions of the same package.

Handling multiple versions of the same installed package

The Node.js package identifier resolution algorithm allows us to install two or more versions of the same package. Returning to the hypothetical *notes* project, notice that the `fred` package is installed not just for the `bar` package but also for the `express` package.

Looking at the algorithm, we know that `require('fred')` in the `bar` package, and in the `express` package, will be satisfied by the corresponding `fred` package installed locally to each.

Normally, the package manager applications will detect the two instances of the `fred` package and install only one. But, suppose the `bar` package required the `fred` version 1.2, while the `express` package required the `fred` version 2.1.

In such a case, the package manager application will detect the incompatibility and install two versions of the `fred` package as so:

- In `/home/david/projects/notes/node_modules/bar/node_modules`, it will install `fred` version 1.2.
- In `/home/david/projects/notes/node_modules/express/node_modules`, it will install `fred` version 2.1.

When the `express` package executes `require('fred')` or `import 'fred'`, it will be satisfied by the package

in `/home/david/projects/notes/node_modules/express/node_modules/fred`. Likewise, the `bar` package will be satisfied by the package

in `/home/david/projects/notes/node_modules/bar/node_modules/fred`.

In both cases, the `bar` and `express` packages have the correct version of the `fred` package available. Neither is aware there is another version of `fred` installed.

The `node_modules` directory is meant for packages required by an application. Node.js also supports installing packages in a global location so they can be used by multiple applications.

Searching for globally installed packages

We've already seen that with npm we can perform a *global install* of a package. For example, command-line tools like hexy or babel are convenient if installed globally. In such a case the package is installed in another folder outside of the project directory. Node.js has two strategies for finding globally installed packages.

Similar to the PATH variable, the NODE_PATH environment variable can be used to list additional directories in which to search for packages. On Unix-like operating systems, NODE_PATH is a colon-separated list of directories, and on Windows it is semicolon-separated. In both cases, it is similar to how the PATH variable is interpreted, meaning that NODE_PATH has a list of directory names in which to find installed modules.



The NODE_PATH approach is not recommended, because of surprising behavior that can happen if people are unaware that this variable must be set. If a specific module located in a specific directory referenced in NODE_PATH is required for a proper function and the variable is not set, the application will likely fail. The best practice is for all dependencies to be explicitly declared, and with Node.js that means listing all dependencies in the package.json file so that npm or yarn can manage the dependencies.

This variable was implemented before the module resolution algorithm just described was finalized. Because of that algorithm, NODE_PATH is largely unnecessary.

There are three additional locations that can hold modules:

- \$HOME/.node_modules
- \$HOME/.node_libraries
- \$PREFIX/lib/node

In this case, \$HOME is what you expect (the user's home directory), and \$PREFIX is the directory where Node.js is installed.

Some recommend against using global packages. The rationale is the desire for repeatability and deployability. If you've tested an app and all its code is conveniently located within a directory tree, you can copy that tree for deployment to other machines. But, what if the app depended on some other file that was magically installed elsewhere on the system? Will you remember to deploy such files? The application author might write documentation saying to *install this* then *install that* and *install something-else* before running *npm install*, but will the users of the application correctly follow all those steps?

The best installation instructions is to simply run *npm install* or *yarn install*. For that to work, all dependencies must be listed in `package.json`.

Before moving forward, let's review the different kinds of module identifiers.

Reviewing module identifiers and pathnames

That was a lot of details spread out over several sections. It's useful, therefore, to quickly review how the module identifiers are interpreted when using the `require`, `import()`, or `import` statements:

- **Relative module identifiers:** These begin with `./` or `../`, and absolute identifiers begin with `/`. The module name is identical to POSIX filesystem semantics. The resultant pathname is interpreted relative to the location of the file being executed. That is, a module identifier beginning with `./` is looked for in the current directory, whereas one starting with `..` is looked for in the parent directory.
- **Absolute module identifiers:** These begin with `/` (or `file://` for ES6 modules) and are, of course, looked for in the root of the filesystem. This is not a recommended practice.
- **Top-level module identifiers:** These do not begin with those strings and are just the module name. These must be stored in a `node_modules` directory, and the Node.js runtime has a nicely flexible algorithm for locating the correct `node_modules` directory.
- **Core modules:** These are the same as the *top-level module identifiers*, in that there is no prefix, but the core modules are prebaked into the Node.js binary.

In all cases, except for the core modules, the module identifier resolves to a file that contains the actual module, and which is loaded by Node.js. Therefore, what Node.js does is to compute the mapping between the module identifier and the actual file name to load.



Using a package manager application is not required. The Node.js module resolution algorithm does not depend on a package manager, like npm or Yarn, to set up the `node_modules` directories. There is nothing magical about those directories, and it is possible to use other means to construct a `node_modules` directory containing installed packages. But the simplest mechanism is to use a package manager application.

Some packages offer what we might call a sub-package included with the main package, let's see how to use them.

Using deep import module specifiers

In addition to a simple module identifier like `require('bar')`, Node.js lets us directly access modules contained within a package. A different module specifier is used that starts with the module name, adding what's called a *deep import* path. For a concrete example, let's look at the `mime` module (<https://www.npmjs.com/package/mime>), which handles mapping a file name to its corresponding MIME type.

In the normal case, you use `require('mime')` to use the package. However, the authors of this package developed a lite version of this package that leaves out a lot of vendor-specific MIME types. For that version, you use `require('mime/lite')` instead. And of course, in an ES6 module, you use `import 'mime'` and `import 'mime/lite'`, as appropriate.

The specifier `mime/lite` is an example of a deep import module specifier.

With such a module identifier, Node.js first locates the `node_modules` directory containing the main package. In this case, that is the `mime` package. By default, the deep import module is simply a path-name relative to the package directory, for example, `/path/to/node_modules/mime/lite`. Going by the rules we've already examined, it will be satisfied by a file named `lite.js` or a by a directory named `lite` containing a file named `index.js` or `index.mjs`.

But it is possible to override the default behavior and have the deep import specifier refer to a different file within the module.

Overriding a deep import module identifier

The deep import module identifier used by code using the package does not have to be the pathname used within the package source. We can put declarations in `package.json` describing the actual pathname for each deep import identifier. For example, a package with interior modules named `./src/cjs-module.js` and `./src/es6-module.mjs` can be remapped with this declaration in `package.json`:

```
{  
  "exports": {  
    "./cjsmodule": "./src/cjs-module.js",  
    "./es6module": "./src/es6-module.mjs"  
  }  
}
```

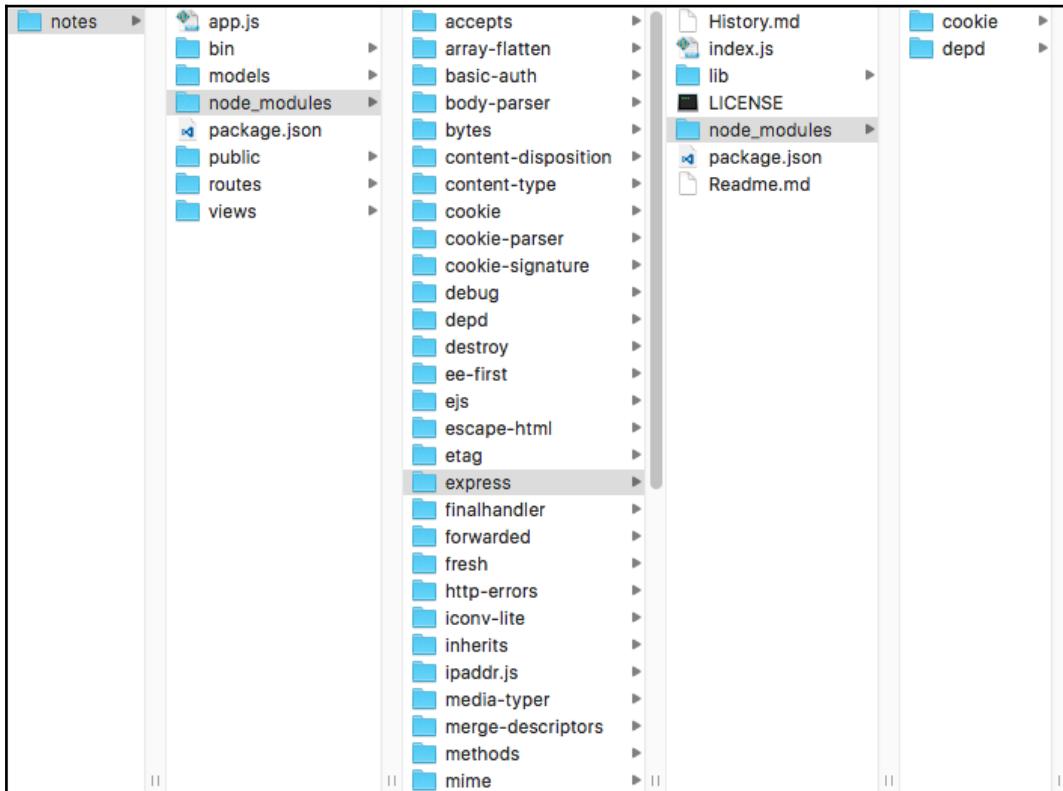
With this, code using such a package can load the inner module using `require('module-name/cjsmodule')` or `import 'module-name/es6module'`. Notice that the filenames do not have to match what's exported.

In a `package.json` file using this `exports` feature, a request for an inner module not listed in `exports` will fail. Supposing the package has a `./src/hidden-module.js` file, calling `require('module-name/src/hidden-module.js')` will fail.

All these modules and packages are meant to be used in the context of a Node.js project. Let's take a brief look at a typical project.

Studying an example project directory structure

A typical Node.js project is a directory containing a `package.json` file declaring the characteristics of the package, especially its dependencies. That, of course, describes a directory module, meaning that each module is its own project. At the end of the day, we create applications, for example, an Express application, and these applications depend on one or more (possibly thousands of) packages that are to be installed:



This is an Express application (we'll start using Express in Chapter 5, *Your First Express Application*) containing a few modules installed in the `node_modules` directory. A typical Express application uses `app.js` as the main module for the application, and has code and asset files distributed in the `public`, `routes`, and `views` directories. Of course, the project dependencies are installed in the `node_modules` directory.

But let's focus on the content of the `node_modules` directory versus the actual project files. In this screenshot, we've selected the `express` package. Notice it has a `package.json` file and there is an `index.js` file. Between those two files, Node.js will recognize the `express` directory as a module, and calling `require('express')` or `import 'express'` will be satisfied by this directory.

The `express` directory has its own `node_modules` directory, in which are installed two packages. The question is, why are those packages installed in `express/node_modules` rather than as a sibling of the `express` package?

Earlier we discussed what happens if two modules (modules A and B) list a dependency on different versions of the same module (C). In such a case, the package manager application will install two versions of C, one as A/node_modules/C and the other as B/node_modules/C. The two copies of C are thus located such that the module search algorithm will cause module A and module B to have the correct version of module C.

That's the situation we see with express/node_modules/cookie. To verify this, we can use an npm command to query for all references to the module:

```
$ npm ls cookie
notes@0.0.0 /Users/David/chap05/notes
  └── cookie-parser@1.3.5
    ├── cookie@0.1.3
    └── express@4.13.4
      └── cookie@0.1.5
```

This says the cookie-parser module depends on version 0.1.3 of cookie, while Express depends on version 0.1.5.

Now that we can recognize what a module is and how they're found in the file system, let's discuss when we can use each of the methods to load modules.

Loading modules using require, import, and import()

Obviously `require` is used in CommonJS modules, and `import` is used in ES6 modules, but there are some details to go over. We've already discussed the format and filename differences between CommonJS and ES6 modules, so let's focus here on loading the modules.

The `require` function is only available in CommonJS modules, and it is used for loading a CommonJS module. The module is loaded synchronously, meaning that when the `require` function returns, the module is completely loaded.

By default, a CommonJS module cannot load an ES6 module. But as we saw with the `simple-dynamic-import.js` example, a CommonJS module can load an ES6 module using `import()`. Since the `import()` function is an asynchronous operation, it returns a Promise, and we, therefore, cannot use the resulting module as a top-level object. But we can use it inside a function:

```
module.exports.usesES6module = async function() {
  const es6module = await import('./es6-module.mjs');
  return es6module.functionCall();
}
```

And at the top-level of a Node.js script, the best we can do is the following:

```
import './simple2.mjs'
.then(simple2 => {
  console.log(simple2.hello());
  console.log(simple2.next());
  console.log(simple2.next());
  console.log(`count = ${simple2.default()}`);
  console.log(`Meaning: ${simple2.meaning}`);
})
.catch(err => {
  console.error(err);
});
```

It's the same as the `simple-dynamic-import.js` example, but we are explicitly handling the Promise returned by `import()` rather than using an `async` function. While we could assign `simple2` to a global variable, other code using that variable would have to accommodate the possibility the assignment hasn't yet been made.

The module object provided by `import()` contains the fields and functions exported with the `export` statements in the ES6 module. As we see here, the default export has the `default` name.

In other words, using an ES6 module in a CommonJS module is possible, so long as we accommodate waiting for the module to finish loading before using it.

The `import` statement is used to load ES6 modules, and it only works inside an ES6 module. The module specifier you hand to the `import` statement is interpreted as a URL.

An ES6 module can have multiple named exports. In the `simple2.mjs` we used earlier, these are the functions `next`, `squared`, and `hello`, and the values `meaning` and `nocount`. ES6 modules can have a single default export, as we saw in `simple2.mjs`.

With `simpldemo2.mjs`, we saw that we can import only the required things from the module:

```
import { default as simple, hello, next } from './simple2.mjs';
```

In this case, we use the exports as just the name, without referring to the module: `simple()`, `hello()`, and `next()`.

It is possible to import just the default export:

```
import simple from './simple2.mjs';
```

In this case, we can invoke the function as `simple()`. We can also use what's called a namespace import; that is similar to how we import CommonJS modules:

```
import * as simple from './simple2.mjs';

console.log(simple.hello());
console.log(simple.next());
console.log(simple.next());
console.log(simple.default());
console.log(simple.meaning);
```

In this case, each property exported from the module is a property of the named object in the `import` statement.

An ES6 module can also use `import` to load a CommonJS module. Loading the `simple.js` module we used earlier is accomplished as follows:

```
import simple from './simple.js';
console.log(simple.next());
console.log(simple.next());
console.log(simple.hello());
```

This is similar to the `default export` method shown for ES6 modules, and we can think of the `module.exports` object inside the CommonJS module as the default export. Indeed, the `import` can be rewritten as follows:

```
import { default as simple } from './simple.js';
```

This demonstrates that the CommonJS `module.exports` object is surfaced as `default` when imported.

We've learned a lot about using modules in Node.js. This included the different types of modules, and how to find them in the file system. Our next step is to learn about package management applications and the npm package repository.

Using npm – the Node.js package management system

As described in Chapter 2, *Setting up Node.js*, npm is a package management and distribution system for Node.js. It has become the de facto standard for distributing modules (packages) for use with Node.js. Conceptually, it's similar to tools such as apt-get (Debian), rpm/yum (Red Hat/Fedora), MacPorts/Homebrew (macOS), CPAN (Perl), or PEAR (PHP). Its purpose is to publish and distribute Node.js packages over the internet using a simple command-line interface. In recent years, it has also become widely used for distributing front-end libraries like jQuery and Bootstrap that are not Node.js modules. With npm, you can quickly find packages to serve specific purposes, download them, install them, and manage packages you've already installed.

The npm application extends on the package format for Node.js, which in turn is largely based on the CommonJS package specification. It uses the same `package.json` file that's supported natively by Node.js, but with additional fields for additional functionality.

The npm package format

An npm package is a directory structure with a `package.json` file describing the package. This is exactly what was referred to earlier as a directory module, except that npm recognizes many more `package.json` tags than Node.js does. The starting point for npm's `package.json` file is the CommonJS Packages/1.0 specification. The documentation for the npm `package.json` implementation is accessed using the following command:

```
$ npm help package.json
```

A basic `package.json` file is as follows:

```
{ "name": "packageName",
  "version": "1.0",
  "main": "mainModuleName".
  "bin": "./path/to/program"
}
```

Npm recognizes many more fields than this, and we'll go over some of them in the coming sections. The file is in JSON format, which, as a JavaScript programmer, you should be familiar with.

There is a lot to cover concerning the npm package.json format, and we'll do so over the following sections.

Accessing npm helpful documentation

The main npm command has a long list of subcommands for specific package management operations. These cover every aspect of the life cycle of publishing packages (as a package author), and downloading, using, or removing packages (as an npm consumer).

You can view the list of these commands just by typing npm (with no arguments). If you see one you want to learn more about, view the help information:

```
$ npm help <command>
```

The help text will be shown on your screen.



Help information is also available on the npm website at: <https://docs.npmjs.com/cli-documentation/>.

Before we can look for and install Node.js packages, we must have a project directory initialized.

Initializing a Node.js package or project with npm init

The npm tool makes it easy to initialize a Node.js project directory. Such a directory contains at the minimum a package.json file and one or more Node.js JavaScript files.

All Node.js project directories are therefore modules, going by the definition we learned earlier. However, in many cases, a Node.js project is not meant to export any functionality but instead is an application. Such a project will likely require other Node.js packages, and those packages will be declared in the `package.json` file so that they're easy to install using npm. The other common use case of a Node.js project is a package of functionality meant to be used by other Node.js packages or applications. These also consist of a `package.json` file plus one or more Node.js JavaScript files, but in this case, they're Node.js modules that export functions and can be loaded using `require()`, or `import`.

What this means is the key to initializing a Node.js project directory is creating the `package.json` file.

While the `package.json` file can be created by hand – it's just a JSON file after all – the npm tool provides a convenient method:

```
$ mkdir example-package
$ cd example-package/
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible
defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (example-package)
version: (1.0.0)
description: This is an example of initializing a Node.js project
entry point: (index.js)
test command: mocha
git repository:
keywords: example, package
author: David Herron <david@davidherron.com>
license: (ISC)
About to write to /home/david/example-package/package.json:

{
  "name": "example-package",
  "version": "1.0.0",
  "description": "This is an example of initializing a Node.js
project",
```

```
"main": "index.js",
"scripts": {
  "test": "mocha"
},
"keywords": [
  "example",
  "package"
],
"author": "David Herron <david@davidherron.com>",
"license": "ISC"
}

Is this OK? (yes) yes
```

In a blank directory, run `npm init`, answer the questions, and as quick as that you have the starting point for a Node.js project.

This is, of course, a starting point, and as you write the code for your project it will often be necessary to use other packages.

Finding npm packages

By default, `npm` packages are retrieved over the internet from the public package registry maintained on <http://npmjs.com>. If you know the module name, it can be installed simply by typing the following:

```
$ npm install moduleName
```

But what if you don't know the module name? How do you discover the interesting modules? The website <http://npmjs.com> publishes a searchable index of the modules in the registry. The `npm` package also has a command-line search function to consult the same index:

NAME	DESCRIPTION	AUTHOR	DATE	VERSION	KEYWORDS
mp3	I An MP3 decoder for...	=devongovett	2014-06-17	0.1.0	audio av aurora.js aurora decode
file-type	I Detect the file...	=mifi-	2017-11-22	7.3.0	mime file type archive image img pic picture flash photo vid
mp3-duration	I Get the duration of...	=ddsolt	2017-10-16	1.1.0	mp3 duration length file audio
is-mp3	I Check if a...	=hemanth	2017-05-02	1.1.3	mp3 type detect check is binary buffer uint8array
browser-id3-writer	I Pure JS library for...	=egorooft	2017-07-06	4.0.0	browser nodejs writer id3 mp3 audio tag library
transloadit	I Node.js SDK for...	=kvz =tim-kos	2017-10-16	1.10.2	transloadit encoding transcoding video audio mp3
audio-decode	I Decode audio data...	=dfcreative...	2017-06-19	1.3.1	audiojs audio dsp decode codec mp3 wav web-audio
id3-parser	I A pure JavaScript...	=creepert	2017-10-26	1.5.1	id3 id3 parser id3 tag id3v2 id3v1 mp3 metadata mp3
audio-type	I Detect the audio...	=dfcreative...	2016-04-23	1.0.2	audio sound wav mp3 flac type detect check is binary buffer
handbrake-js	I Handbrake for...	=75lb	2017-05-26	2.2.2	handbrake encode transcode video mp4 m4v avi h.264 h.265 vp8
npmdoc-youtube-mp3	I ##### basic apl...	=npmdoc	2017-04-26	2017.4...	documentation youtube-mp3
npmtest-youtube-mp3	I ##### basic test...	=npmtest2	2017-04-25	2017.4...	coverage test youtube-mp3
jsmediatags	I Media Tags Reader...	=aadsim	2017-10-27	3.8.1	ID3 tags mp3 audio mp4
hypem-resolver	I Resolve a hypem...	=feedm3	2016-03-05	1.2.5	hypem soundcloud mp3 converter
soundcloud-mp3	I Guess the mp3...	=olitzilla	2015-06-17	1.0.0	
mp3-to-video	I Create video from...	=slorenzo	2016-10-05	1.0.3	mp3 convert video ffmpeg
music-metadata	I Streaming music...	=borewit	2017-10-25	0.8.7	tag tags MusicBrainz Discogs Picard ID3v1 ID3v2 m4a mp3
react-cassette-player	I Simple ReactJS...	=chadpaulson	2016-04-10	1.1.2	react-component svg html5 audio html5 audio mp3 ogg wav medi
media-library	I a media library...	=guillaume86	2016-01-05	1.2.4	media mp3 audio library id3 music
amrToMp3	I 微信amr音频转mp3模块	=traveller	2016-10-06	1.0.7	amr amr to mp3 wechat amr

Of course, upon finding a module, it's installed as follows:

```
$ npm install acoustid
```

The npm repository uses a few package.json fields to aid in finding packages.

The package.json fields that help finding packages

For a package to be easily found in the npm repository requires a good package name, package description, and keywords. The npm search function scans those package attributes and presents them in search results.

The relevant package.json fields are as follows:

```
{
  ...
  "description": "My wonderful package that walks dogs",
  "homepage": "http://npm.dogs.org/dogwalker/",
  "author": "dogwhisperer@dogs.org",
  "keywords": [ "dogs", "dog walking" ]
  ...
}
```

The npm view command shows us information from package.json file for a given package, and with the --json flag we're shown the raw JSON.

The name tag is of course the package name, and it is used in URLs and command names, so choose one that's safe for both. If you desire to publish a package in the public npm repository, it's helpful to check whether a particular name is already being used by searching on <https://npmjs.com> or by using the npm search command.

The description tag is a short description that's meant as a brief/terse description of the package.

It is the name and description tags that are shown in npm search results.

The `keywords` tag is where we list attributes of the package. The npm website contains pages listing all packages using a particular keyword. These keyword indexes are useful when searching for a package since it lists the related packages in one place, and therefore when publishing a package it's useful to land on the correct keyword pages.

Another source is the contents of the `README.md` file. This file should be added to the package to provide basic package documentation. This file is shown on the package page on `npmjs.com`, and therefore it is important for this file to convince potential users of your package to actually use it. As the file name implies, this is a Markdown file.

Once you have found a package to use, you must install it in order to use the package.

Installing an npm package

The `npm install` command makes it easy to install packages upon finding one of your dreams, as follows:

```
$ npm install express
/home/david/projects/notes/
- express@4.13.4
...
```

The named module is installed in `node_modules` in the current directory. During the installation process, the package is set up. This includes installing any packages it depends on and running the `preinstall` and `postinstall` scripts. Of course, installing the dependent packages also involves the same installation process of installing dependencies and executing pre-install and post-install scripts.

Some packages in the npm repository have a package *scope* prepended to the package name. The package name in such cases is presented as `@scope-name/package-name`, or, for example, `@akashacms/plugins-footnotes`. In such a package, the `name` field in `package.json` contains the full package name with its `@scope`.

We'll discuss dependencies and scripts later. In the meantime, we notice that a version number was printed in the output, so let's discuss package version numbers.

Installing a package by version number

Version number matching in npm is powerful and flexible. With it, we can target a specific release of a given package or any version number range. By default, npm installs the latest version of the named package, as we did in the previous section. Whether you take the default or specify a version number, npm will determine what to install.

The package version is declared in the `package.json` file, so let's look at the relevant fields:

```
{ ...  
  "version": "1.2.1",  
  "dist-tags": {  
    "latest": "1.2.1"  
  },  
  ... }
```

The `version` field obviously declares the current package version. The `dist-tags` field lists symbolic tags that the package maintainer can use to aid their users in selecting the correct version. This field is maintained by the `npm dist-tag` command.

The `npm install` command supports these variants:

```
$ npm install package-name@tag  
$ npm install package-name@version  
$ npm install package-name@version-range
```

The last two are what they sound like. You can specify `express@4.16.2` to target a precise version, or `express@">4.1.0 < 5.0"` to target a range of Express V4 versions. We might use that specific expression because Express 5.0 might include breaking changes.

The version match specifiers include the following choices:

- **Exact version match:** `1.2.3`
- **At least version N:** `>1.2.3`
- **Up to version N:** `<1.2.3`
- **Between two releases:** `>=1.2.3 <1.3.0`

The `@tag` attribute is a symbolic name such as `@latest`, `@stable`, or `@canary`. The package owner assigns these symbolic names to specific version numbers and can reassign them as desired. The exception is `@latest`, which is updated whenever a new release of the package is published.



For more documentation, run these commands: `npm help json` and `npm help npm-dist-tag`.

In selecting the correct package to use, sometimes we want to use packages that are not in the npm repository.

Installing packages from outside the npm repository

As awesome as the npm repository is, we don't want to push everything we do through their service. This is especially true for internal development teams who cannot publish their code for all the world to see. Fortunately, Node.js packages can be installed from other locations. Details about this are in `npm help package.json` in the `dependencies` section. Some examples are as follows:

- **URL:** You can specify any URL that downloads a tarball, that is, a `.tar.gz` file. For example, GitHub or GitLab repositories can easily export a tarball URL. Simply go to the **Releases** tab to find them.
- **Git URL:** Similarly, any Git repository can be accessed with the right URL, for example:

```
$ npm install git+ssh://user@hostname:project.git#git-tag
```

- **GitHub shortcut:** For GitHub repositories, you can list just the repository specifier, such as `expressjs/express`. A tag or a commit can be referenced using `expressjs/express#tag-name`.
- **GitLab, BitBucket, and GitHub URL shortcuts:** In addition to the GitHub shortcut, npm supports a special URL format for specific Git services with URLs like `github:user/repo`, `bitbucket:user/repo`, and `gitlab:user/repo`.
- **Local filesystem:** You can install from a local directory using a URL with the: `file:.../.../path/to/dir`.

Sometimes we need to install a package for use by several projects, without requiring that each project installs the package.

Global package installs

In some instances, you want to install a module globally, so that it can be used from any directory. For example, the Grunt or Babel build tools are widely useful, and conceivably you will find it useful if these tools are installed globally. Simply add the `-g` option:

```
$ npm install -g grunt-cli
```

If you get an error, and you're on a Unix-like system (Linux/Mac), you may need to run this with `sudo`:

```
$ sudo npm install -g grunt-cli
```

This variant, of course, runs `npm install` with elevated permissions.



The npm website offers a guideline with more information at <https://docs.npmjs.com/resolving-eacces-permissions-errors-when-installing-packages-globally>.

If a local package install lands in `node_modules`, where does a global package install land? On a Unix-like system, it lands in `PREFIX/lib/node_modules`, and on Windows, it lands in `PREFIX/node_modules`. In this case, `PREFIX` means the directory where Node.js is installed. You can inspect the location of the directory as follows:

```
$ npm config get prefix  
/opt/local
```

The algorithm used by Node.js for the `require` function automatically searches the directory for packages if the package is not found elsewhere.

ES6 modules do not support global packages.

Many believe it is not a good idea to install packages globally, which we will look at next.

Avoiding global module installation

Some in the Node.js community now frown on installing packages globally. One rationale is that a software project is more reliable if all its dependencies are explicitly declared. If a build tool such as Grunt is required but is not explicitly declared in `package.json`, the users of the application would have to receive instructions to install Grunt, and they would have to follow those instructions.

Users being users, they might skip over the instructions, fail to install the dependency, and then complain the application doesn't work. Surely, most of us have done that once or twice.

It's recommended to avoid this potential problem by installing everything locally via one mechanism—the `npm install` command.

There are two strategies we use to avoid using globally installed Node.js packages. For the packages that install commands, we can configure the `PATH` variable, or use `npx` to run the command. In some cases, a package is used only during development and can be declared as such in `package.json`.

Maintaining package dependencies with npm

The `npm install` command by itself, with no package name specified, installs the packages listed in the `dependencies` section of `package.json`. Likewise, the `npm update` command compares the installed packages against the dependencies and against what's available in the npm repository and updates any package that is out of date in regards to the repository.

These two commands make it easy and convenient to set up a project, and to keep it up to date as dependencies are updated. The package author simply lists all the dependencies, and npm installs or updates the dependencies required for using the package. What happens is npm looks in `package.json` for the `dependencies` or `devDependencies` fields, and it works out what to do from there.

You can manage the dependencies manually by editing `package.json`. Or you can use `npm` to assist you with editing the dependencies. You can add a new dependency like so:

```
$ npm install akasharender --save
```

With the `--save` flag, `npm` will add a `dependencies` tag to `package.json`:

```
"dependencies": {  
    "akasharender": "^0.7.8"  
}
```

With the added dependency, when your application is installed, `npm` will now install the package along with any other dependencies listed in `package.json` file.

The `devDependencies` lists modules used during development and testing. The field is initialized the same as the preceding one, but with the `--save-dev` flag. The `devDependencies` can be used to avoid some cases where one might instead perform a global package install.

By default, when `npm install` is run, modules listed in both `dependencies` and `devDependencies` are installed. Of course, the purpose of having two dependency lists is to control when each set of dependencies is installed.

```
$ npm install --production
```

This installs the "production" version, which means to install only the modules listed in `dependencies` and none of the `devDependencies` modules. For example, if we use a build tool like Babel in development, the tool should not be installed in production.

While we can manually maintain dependencies in `package.json`, `npm` can handle this for us.

Automatically updating package.json dependencies

With npm@5 (also known as npm version 5), one change was that it's no longer required to add `--save` to the `npm install` command. Instead, `npm` by default acts as if you ran the command with `--save`, and will automatically add the dependency to `package.json`. This is meant to simplify using `npm`, and it is arguably more convenient than `npm` now does this. At the same time, it can be very surprising and inconvenient for `npm` to go ahead and modify `package.json` for you. The behavior can be disabled by using the `--no-save` flag, or it can be permanently disabled using the following:

```
$ npm config set save false
```

The `npm config` command supports a long list of settable options for tuning the behavior of `npm`. See `npm help config` for the documentation and `npm help 7 config` for the list of options.

Now let's talk about the one big use for package dependencies: to fix or avoid bugs.

Fixing bugs by updating package dependencies

Bugs exist in every piece of software. An update to the Node.js platform may break an existing package, as might an upgrade to packages used by the application. Your application may trigger a bug in a package it uses. In these and other cases, fixing the problem might be as simple as updating a package dependency to a later (or earlier) version.

First, identify whether the problem exists in the package or in your code. After determining it's a problem in another package, investigate whether the package maintainers have already fixed the bug. Is the package hosted on GitHub or another service with a public issue queue? Look for an open issue on this problem. That investigation will tell you whether to update the package dependency to a later version. Sometimes, it will tell you to revert to an earlier version; for example, if the package maintainer introduced a bug that doesn't exist in an earlier version.

Sometimes, you will find that the package maintainers are unprepared to issue a new release. In such a case, you can fork their repository and create a patched version of their package. In such a case, your package might use a Github URL referencing your patched package.

One approach to fixing this problem is **pinning** the package version number to one that's known to work. You might know that version 6.1.2 was the last release against which your application functioned and that starting with version 6.2.0 your application breaks. Hence, in `package.json`:

```
"dependencies": {  
    "module1": "6.1.2"  
}
```

This freezes your dependency on the specific version number. You're free, then, to take your time updating your code to work against later releases of the module. Once your code is updated, or the upstream project is updated, change the dependency appropriately.

When listing dependencies in `package.json`, it's tempting to be lazy, but that leads to trouble.

Explicitly specifying package dependency version numbers

As we've said several times in this chapter, explicitly declaring your dependencies is A Good Thing. We've already touched on this, but it's worth reiterating and to see how npm makes this easy to accomplish.

The first step is ensuring that your application code is checked into a source code repository. You probably already know this, and even have the best of intentions to ensure that everything is checked in. With Node.js, each module should have its own repository rather than putting every single last piece of code in one repository.

Each module can then progress on its own timeline. A breakage in one module is easy to back out by changing the version dependency in `package.json`.

The next step is to explicitly declare all dependencies of every module. The goal is simplifying and automating the process of setting up every module. Ideally, on the Node.js platform, the module setup is as simple as running `npm install`.

Any additional required steps can be forgotten or executed incorrectly. An automated setup process eliminates several kinds of potential mistakes.

With the `dependencies` and `devDependencies` sections of `package.json`, we can explicitly declare not only the dependencies but the precise version numbers.

The lazy way of declaring dependencies is putting `*` in the version field. That uses the latest version in the npm repository. This will seem to work, until that one day the maintainers of that package introduce a bug. You'll type `npm update`, and all of a sudden your code doesn't work. You'll head over to the GitHub site for the package, look in the issue queue, and possibly see that others have already reported the problem you're seeing. Some of them will say that they've pinned on the previous release until this bug is fixed. What that means is their `package.json` file does not depend on `*` for the latest version, but on a specific version number before the bug was created.

Don't do the lazy thing, do the smart thing.

The other aspect of explicitly declaring dependencies is to not implicitly depend on global packages. Earlier, we said that some people in the Node.js community caution against installing modules in the global directories. This might seem like an easy shortcut to sharing code between applications. Just install it globally, and you don't have to install the code in each application.

But, doesn't that make deployment harder? Will the new team member be instructed on all the special files to install here and there to make the application run? Will you remember to install that global module on all destination machines?

For Node.js, that means listing all the module dependencies in `package.json`, and then the installation instructions are simply `npm install`, followed perhaps by editing a configuration file.

While most packages in the npm repository are libraries with an API, some are tools we can run from the command line.

Packages that install commands

Some packages install command-line programs. A side effect of installing such packages is a new command that you can type at the shell prompt or use in shell scripts. An example is the `hexy` program that we briefly used in [Chapter 2, Setting Up Node.js](#). Another example is the widely used Grunt or Babel build tools.

The recommendation to explicitly declare all dependencies in `package.json` applies to command-line tools as well as any other package. Therefore these packages will typically be installed locally. This requires special care in setting up the `PATH` environment variable correctly. As you should already be aware, the `PATH` variable is used on both Unix-like systems and Windows to list the directories in which the command-line shell searches for commands.

The command can be installed to one of two places:

- **Global install:** It is installed either to a directory, such as `/usr/local`, or to the `bin` directory where Node.js was installed. The `npm bin -g` command tells you the absolute pathname for this directory. In this case, it's unlikely you'll have to modify the `PATH` environment variable.
- **Local install:** Installs to `node_modules/.bin` in the package where the module is being installed, the `npm bin` command tells you the absolute pathname for the directory. Because the directory is inconveniently located to run commands, a change to the `PATH` variable is useful.

To run the command, simply type the command name at a shell prompt. This works correctly if the directory where the command is installed happens to be in the `PATH` variable. Let's look at how to configure the `PATH` variable to handle locally installed commands.

Configuring the PATH variable to handle locally installed commands

Assume we have installed the `hexy` command like so:

```
$ npm install hexy
```

As a local install, this creates a command as `node_modules/.bin/hexy`. We can attempt to use it as follows:

```
$ hexy package.json
-bash: hexy: command not found
```

But this breaks because the command is not in a directory listed in the `PATH`. The workaround is to use the full pathname or relative pathname:

```
$ ./node_modules/.bin/hexy package.json
... hexy output
```

But obviously typing the full or partial pathname is not a user-friendly way to execute the command. We want to use the commands installed by modules, and we want a simple process for doing so. This means, we must add an appropriate value in the `PATH` variable, but what is it?

For global package installations, the executable lands in a directory that is probably already in your `PATH` variable, like `/usr/bin` or `/usr/local/bin`. Local package installations require special handling. The full path for the `node_modules/.bin` directory varies for each project, and obviously it won't work to add the full path for every `node_modules/.bin` directory to your `PATH`.

Adding `./node_modules/.bin` to the `PATH` variable (or, on Windows, `.\node_modules\.bin`) works great. Any time your shell is in the root of a Node.js project, it will automatically find locally installed commands from Node.js packages.

How we do this depends on the command shell you use and your operating system.

On a Unix-like system, the command shells are `bash` and `csh`. Your `PATH` variable would be set up in one of these ways:

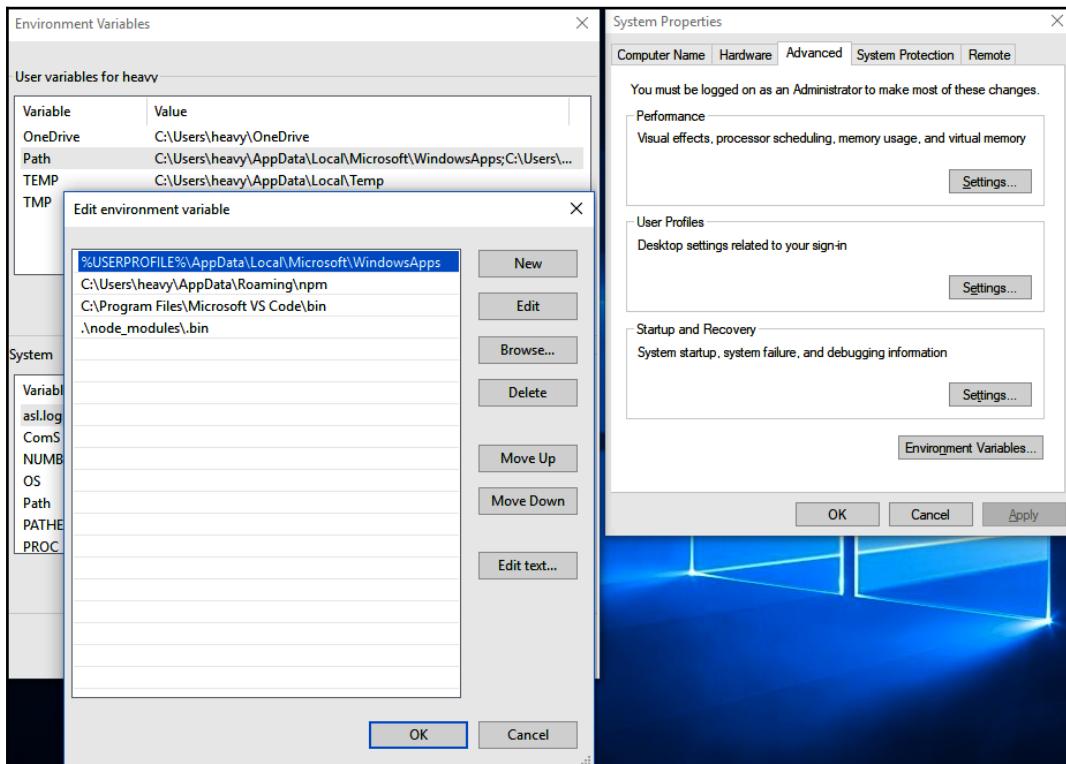
```
$ export PATH=./node_modules/.bin:${PATH}      # bash  
$ setenv PATH ./node_modules/.bin:${PATH}      # csh
```

The next step is adding the command to your login scripts so the variable is always set. On `bash`, add the corresponding line to `~/.bashrc`, and on `csh` add it to `~/.cshrc`.

Once this is accomplished the command-line tool executes correctly.

Configuring the PATH variable on Windows

On Windows, this task is handled through a system-wide settings panel:



This pane of the **System Properties** panel is found by searching for `PATH` in the **Windows Settings** screen. Click on the **Environment Variables** button, then select the **Path** variable, and finally click on the **Edit** button. On the screen here, click the **New** button to add an entry to this variable, and enter `.\node_modules\.\bin` as shown. You'll have to restart any open command shell windows. Once you do, the effect will be as shown previously.

As easy as it is to modify the `PATH` variable, we don't want to do this in all circumstances.

Avoiding modifications to the `PATH` variable

What if you don't want to add these variables to your `PATH` at all times? The `npm-path` module may be of interest. This is a small program that computes the correct `PATH` variable for your shell and operating system. See the package at <https://www.npmjs.com/package/npm-path>.

Another option is to use the `npx` command to execute such commands. This tool is automatically installed alongside the `npm` command. This command either executes commands from a locally installed package or it silently installs commands in a global cache:

```
$ npx hexy package.json
```

Using `npx` is this easy.

Of course, once you've installed some packages, they'll go out of date and need to be updated.

Updating packages you've installed when they're outdated

The coder codes, updating their package, leaving you in the dust unless you keep up.

To find out whether your installed packages are out of date, use the following command:

```
$ npm outdated
```

The report shows the current `npm` packages, the currently installed version, as well as the current version in the `npm` repository. Updating outdated packages is very simple:

```
$ npm update express  
$ npm update
```

Specifying a package name updates just the named package. Otherwise, it updates every package that would be printed by `npm outdated`.

Npm handles more than package management, it has a decent built-in task automation system.

Automating tasks with scripts in package.json

The npm command handles not just installing packages, it can also be used to automate running tasks related to the project. In `package.json`, we can add a field, `scripts`, containing one or more command strings. Originally scripts were meant to handle tasks related to installing an application, such as compiling native code, but they can be used for much more. For example, you might have a deployment task using `rsync` to copy files to a server. In `package.json`, you can add this:

```
{ ...  
  "scripts": {  
    "deploy": "rsync --archive --delete local-dir  
              user@host:/path/to/dest-dir  
  }  
... }
```

What's important here is that we can add any script we like, and the `scripts` entry records the command to run:

```
$ npm run deploy
```

Once it has been recorded in `scripts`, running the command is this easy.

There is a long list of "lifecycle events" for which npm has defined script names. These include the following:

- `install`, for when the package is installed
- `uninstall`, for when it is uninstalled
- `test`, for running a test suite
- `start` and `stop`, for controlling a server defined by the package

Package authors are free to define any other script they like.



For the full list of predefined script names, see the documentation: <https://docs.npmjs.com/misc/scripts>

Npm also defines a pattern for scripts that run before or after another script, namely to prepend `pre` or `post` to the script name. Therefore the `pretest` script runs before the `test` script, and the `posttest` script runs afterward.

A practical example is to run a test script in a `prepublish` script to ensure the package is tested before publishing it to the npm repository:

```
{  
  "scripts": {  
    "test": "cd test && mocha",  
    "prepublish": "npm run test"  
  }  
}
```

With this combination, if the test author types `npm publish`, the `prepublish` script will cause the `test` script to run, which in turn uses `mocha` to run the test suite.

It is a well-known best practice to automate all administrative tasks, if only so that you never forget how to run those tasks. Creating the `scripts` entries for every such task not only prevents you from forgetting how to do things, but it also documents the administrative tasks for the benefit of others.

Next, let's talk about how to ensure the Node.js platform on which a package is executed supports the required features.

Declaring Node.js version compatibility

It's important that your Node.js software runs on the correct version of Node.js. The primary reason being that the Node.js platform features required by your package are available every time your package is run. Therefore, the package author must know which Node.js releases are compatible with the package, and then describe in `package.json` that compatibility.

This dependency is declared in `package.json` using the `engines` tag:

```
"engines": {  
  "node": ">= 8.x <=10.x"  
}
```

The version string is similar to what we can use in `dependencies` and `devDependencies`. In this case, we've defined that the package is compatible with Node.js 8.x, 9.x, and 10.x.

Now that we know how to construct a package, let's talk about publishing packages.

Publishing an npm package

All those packages in the npm repository came from people like you with an idea of a better way of doing something. It is very easy to get started with publishing packages.



Online docs about publishing packages can be found at <https://docs.npmjs.com/getting-started/publishing-npm-packages>.

Also consider this: <https://xkcd.com/927/>.

You first use the `npm adduser` command to register yourself with the npm repository. You can also sign up with the website. Next, you log in using the `npm login` command.

Finally, while sitting in the package root directory, use the `npm publish` command. Then, stand back so that you don't get stampeded by the crush of thronging fans, or, maybe not. There are several zillion packages in the repository, with hundreds of packages added every day. To get yours to stand out, you will require some marketing skill, which is another topic beyond the scope of this book.

It is suggested that your first package be a scoped package, for example, `@my-user-name/my-great-package`.

We've learned a lot in this section about using npm to manage and publish packages. But npm is not the only game in town for managing Node.js packages.

The Yarn package management system

As powerful as npm is, it is not the only package management system for Node.js. Because the Node.js core team does not dictate a package management system, the Node.js community is free to roll up their sleeves and develop any system they feel best. That the vast majority of us use npm is a testament to its value and usefulness. But, there is a significant competitor.

Yarn (see <https://yarnpkg.com/en/>) is a collaboration between engineers at Facebook, Google, and several other companies. They proclaim that Yarn is ultrafast, ultra-secure (by using checksums of everything), and ultrareliable (by using a `yarn-lock.json` file to record precise dependencies).

Instead of running their own package repository, Yarn runs on top of the npm package repository at npmjs.com. This means that the Node.js community is not forked by Yarn, but enhanced by having an improved package management tool.

The npm team responded to Yarn in npm@5 (also known as npm version 5) by improving performance and by introducing a `package-lock.json` file to improve reliability. The npm team has implemented additional improvements in npm@6.

Yarn has become very popular and is widely recommended over npm. They perform extremely similar functions, and the performance is not that different from npm@5. The command-line options are worded differently. Everything we've discussed for npm is also supported by Yarn, albeit with slightly different command syntax. An important benefit Yarn brings to the Node.js community is that the competition between Yarn and npm seems to be breeding faster advances in Node.js package management overall.

To get you started, these are the most important commands:

- `yarn add`: Adds a package to use in your current package
- `yarn init`: Initializes the development of a package
- `yarn install`: Installs all the dependencies defined in a `package.json` file
- `yarn publish`: Publishes a package to a package manager
- `yarn remove`: Removes an unused package from your current package

Running `yarn` by itself does the `yarn install` behavior. There are several other commands in Yarn, and `yarn help` will list them all.

Summary

You learned a lot in this chapter about modules and packages for Node.js. Specifically, we covered implementing modules and packages for Node.js, the different module structures we can use, the difference between CommonJS and ES6 modules, managing installed modules and packages, how Node.js locates modules, the different types of modules and packages, how and why to declare dependencies on specific package versions, how to find third-party packages, and we gained a good grounding in using npm or Yarn to manage the packages we use and to publish our own packages.

Now that you've learned about modules and packages, we're ready to use them to build applications, which we'll look at in the next chapter.

4

HTTP Servers and Clients

Now that you've learned about Node.js modules, it's time to put this knowledge to use by building a simple Node.js web application. The goal of this book is to learn about web application development with Node.js. The next step in that journey is getting a basic understanding of the `HTTPServer` and `HTTPClient` objects. To do that, we'll create a simple application that will enable us to explore a popular application framework for Node.js—Express. In later chapters, we'll do more complex work on the application, but before we can walk, we must learn to crawl.

The goal of this chapter is to start to understand how to create applications on the Node.js platform. We'll create a handful of small applications, which means we'll be writing code and talking about what it does. Beyond learning about some specific technologies, we want to get comfortable with the process of initializing a work directory, creating the Node.js code for an application, installing dependencies required by the application, and running/testing the application.

The Node.js runtime includes objects such as `EventEmitter`, `HTTPServer`, and `HTTPClient`, which provide a foundation on which we can build applications. Even if we rarely use these objects directly, it is useful to understand how they work, and in this chapter, we will cover a couple of exercises using these specific objects.

We'll first build a simple application directly using the `HTTPServer` object. Then, we'll move on to using Express to create an application for computing Fibonacci numbers. Because this can be computationally expensive, we'll use this to explore why it's important to not block the event queue in Node.js and what happens to applications that do. This will give us an excuse to develop a simple background **Representational State Transfer (REST)** server, an HTTP client for making requests on that server, and the implementation of a multi-tier web application.

In today's world, the microservice application architecture implements background REST servers, which is what we'll do in this chapter.

We will cover the following topics in this chapter:

- Sending and receiving events using the `EventEmitter` pattern
- Understanding an HTTP server application by building a simple application
- Web application frameworks
- Using the Express framework to build a simple application
- Handling computationally intensive calculations in an Express application and the Node.js event loop
- Making HTTP Client requests
- Creating a simple REST service with Express

By going through these topics, you'll gain an understanding of several aspects of designing HTTP-based web services. The goal is for you to understand how to create or consume an HTTP service and to get an introduction to the Express framework. By the end of this chapter, you'll have a basic understanding of these two tools.

That's a lot to cover, and it will give us a good foundation for the rest of this book.

Sending and receiving events with `EventEmitter`

`EventEmitter` is one of the core idioms of Node.js. If Node.js's core idea is an event-driven architecture, emitting events from an object is one of the primary mechanisms of that architecture. `EventEmitter` is an object that gives notifications (events) at different points in its life cycle. For example, an `HTTPServer` object emits events concerning each stage of the startup/shutdown of the Server object and at each stage of processing HTTP requests from HTTP clients.

Many core Node.js modules are `EventEmitter` objects, and `EventEmitter` objects are an excellent skeleton on which to implement asynchronous programming. `EventEmitter` objects are so much a part of the Node.js woodwork that you may skip over their existence. However, because they're used everywhere, we need some understanding of what they are and how to use them when necessary.

In this chapter, we'll work with the `HTTPServer` and `HTTPClient` objects. Both are subclasses of the `EventEmitter` class and rely on it to send events for each step of the HTTP protocol. In this section, we'll first learn about using JavaScript classes, and then we will create an `EventEmitter` subclass so that we can learn about `EventEmitter`.

JavaScript classes and class inheritance

Before getting started on the `EventEmitter` class, we need to take a look at another one of the ES2015 features: classes. JavaScript has always had objects and the concept of a class hierarchy, but nothing as formal as in other languages. The ES2015 class object builds on the existing prototype-based inheritance model, but with a syntax that looks a lot like class definitions in other languages.

For example, consider the following class, which we'll be using later in this book:

```
class Note {  
    constructor(key, title, body) {  
        this._key = key;  
        this._title = title;  
        this._body = body;  
    }  
    get key() { return this._key; }  
    get title() { return this._title; }  
    set title(newTitle) { return this._title = newTitle; }  
    get body() { return this._body; }  
    set body(newBody) { return this._body = newBody; }  
}
```

This should look familiar to anyone who's implemented a class definition in other languages. The class has a name—`Note`. There is also a constructor method and attributes for each instance of the class.

Once you've defined the class, you can export the class definition to other modules:

```
module.exports.Note = class Note { .. } # in CommonJS modules  
export class Note { .. } # in ES6 modules
```

Functions marked with the `get` or `set` keywords are getters and setters, used as follows:

```
const aNote = new Note("key", "The Rain in Spain", "Falls mainly on  
the plain");  
const key = aNote.key;  
var title = aNote.title;  
aNote.title = "The Rain in Spain, which made me want to cry with joy";
```

New instances of a class are created with `new`. You access a getter or setter function as if it is a simple field on the object. Behind the scenes, the getter/setter function is invoked.

The preceding implementation is not the best because the `_title` and `_body` fields are publicly visible and there is no data-hiding or encapsulation. There is a technique to better hide the field data, which we'll go over in [Chapter 5, Your First Express Application](#).

You can test whether a given object is of a certain class by using the `instanceof` operator:

```
if (anotherNote instanceof Note) {  
    ... it's a Note, so act on it as a Note  
}
```

Finally, you declare a subclass using the `extends` operator, similar to how you would in other languages:

```
class LoveNote extends Note {  
    constructor(key, title, body, heart) {  
        super(key, title, body);  
        this._heart = heart;  
    }  
    get heart() { return this._heart; }  
    set heart(newHeart) { return this._heart = newHeart; }  
}
```

In other words, the `LoveNote` class has all the fields of `Note`, plus a new field named `heart`.

This was a brief introduction to JavaScript classes. By the end of this book, you'll have had lots of practice with this feature. The `EventEmitter` class gives us a practical use for classes and class inheritance.

The EventEmitter class

The `EventEmitter` object is defined in the `events` module of Node.js. Using the `EventEmitter` class directly means performing `require('events')`. In most cases, we don't do this. Instead, our typical use of `EventEmitter` objects is via an existing object that uses `EventEmitter` internally. However, there are some cases where needs dictate implementing an `EventEmitter` subclass.

Create a file named `pulser.mjs`, containing the following code:

```
import EventEmitter from 'events';

export class Pulser extends EventEmitter {
    start() {
        setInterval(() => {
            console.log(`[${new Date().toISOString()}] >>> pulse`);
            this.emit('pulse');
            console.log(`[${new Date().toISOString()}] <<< pulse`);
        }, 1000);
    }
}
```

This is an ES6 module that defines a class named `Pulser`. The class inherits from `EventEmitter` and provides a few methods of its own.

Another thing to examine is how `this.emit` in the callback function refers to the `Pulser` object instance. This implementation relies on the ES2015 arrow function. Before arrow functions, our callbacks used a regular function, and `this` would not have referred to the `Pulser` object instance. Instead, `this` would have referred to some other object related to the `setInterval` function. One of the attributes of arrow functions is that `this` inside the arrow function has the same value as `this` in the surrounding context. This means, in this case, that `this` does refer to the `Pulser` object instance.

Back when we had to use `function`, rather than an arrow function, we had to assign `this` to another variable, as follows:

```
class Pulser extends EventEmitter {
    start() {
        var self = this;
        setInterval(function() {
            self.emit(...);
        });
    }
}
```

What's different is the assignment of `this` to `self`. The value of `this` inside the function is different—it is related to the `setInterval` function—but the value of `self` remains the same in every enclosed scope. You'll see this trick used widely, so remember this in case you come across this pattern in code that you're maintaining.

If you want to use a simple `EventEmitter` object but with your own class name, the body of the extended class can be empty:

```
class HeartBeat extends EventEmitter {}  
const beatMaker = new HeartBeat();
```

The purpose of the `Pulser` class is to send a timed event once a second to any listeners. The `start` method uses `setInterval` to kick off a repeated callback execution that is scheduled for every second and calls `emit` to send the `pulse` events to any listeners.

Now, let's see how we can use the `Pulser` object. Create a new file called `pulsed.mjs`, containing the following code:

```
import { Pulser } from './pulser.mjs';  
  
// Instantiate a Pulser object  
const pulser = new Pulser();  
// Handler function  
pulser.on('pulse', () => {  
    console.log(`${new Date().toISOString()} pulse received`);  
});  
// Start it pulsing  
pulser.start();
```

Here, we create a `Pulser` object and consume its `pulse` events. Calling `pulser.on('pulse')` sets up an event listener for the `pulse` events to invoke the callback function. It then calls the `start` method to get the process going.

When it is run, you should see the following output:

```
$ node pulsed.mjs  
2020-01-06T06:12:29.530Z >>> pulse  
2020-01-06T06:12:29.534Z pulse received  
2020-01-06T06:12:29.534Z <<< pulse  
2020-01-06T06:12:30.538Z >>> pulse  
2020-01-06T06:12:30.539Z pulse received  
2020-01-06T06:12:30.539Z <<< pulse
```

For each `pulse` event received, a `pulse received` message is printed.

That gives you a little practical knowledge of the `EventEmitter` class. Let's now look at its operational theory.

The `EventEmitter` theory

With the `EventEmitter` class, your code emits events that other code can receive. This is a way of connecting two separated sections of your program, kind of like how quantum entanglement means two electrons can communicate with each other from any distance. It seems simple enough.

The event name can be anything that makes sense to you, and you can define as many event names as you like. Event names are defined simply by calling `.emit` with the event name. There's nothing formal to do and no registry of event names is required. Simply making a call to `.emit` is enough to define an event name.

By convention, the `error` event name indicates an error.



An object sends events using the `.emit` function. Events are sent to any listeners that have registered to receive events from the object. The program registers to receive an event by calling that object's `.on` method, giving the event name and an event handler function.

There is no central distribution point for all events. Instead, each instance of an `EventEmitter` object manages its own set of listeners and distributes its events to those listeners.

Often, it is required to send data along with an event. To do so, simply add the data as arguments to the `.emit` call, as follows:

```
this.emit('eventName', data1, data2, ...);
```

When the program receives the event, the data appears as arguments to the callback function. Your program listens to this event, as follows:

```
emitter.on('eventName', (data1, data2, ...theArgs) => {
  // act on event
});
```

There is no handshaking between event receivers and the event sender. That is, the event sender simply goes on with its business and it gets no notifications about any events that were received, any action taken, or any errors that occurred.

In this example, we used another one of the ES2015 features—the `rest` operator—used here in the form of `...theArgs`. The `rest` operator catches any number of remaining function parameters into an array. Since `EventEmitter` can pass along any number of parameters and the `rest` operator can automatically receive any number of parameters, it's a match made in heaven—or at least in the TC-39 committee.

We've now learned how to use JavaScript classes and how to use the `EventEmitter` class. What's next is examining how the `HTTPServer` object uses `EventEmitter`.

Understanding HTTP server applications

The `HTTPServer` object is the foundation of all Node.js web applications. The object itself is very close to the HTTP protocol, and its use requires knowledge of this protocol. Fortunately, in most cases, you'll be able to use an application framework, such as Express, to hide the HTTP protocol details. As application developers, we want to focus on business logic.

We already saw a simple HTTP server application in [Chapter 2, Setting Up Node.js](#). Because `HTTPServer` is an `EventEmitter` object, the example can be written in another way to make this fact explicit by separately adding the event listener:

```
import * as http from 'http';

const server = http.createServer();
server.on('request', (req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, World!\n');
});
server.listen(8124, '127.0.0.1');
console.log('Server running at http://127.0.0.1:8124');
```

Here, we created an HTTP server object, then attached a listener to the `request` event, and then told the server to listen to connections from `localhost` (`127.0.0.1`) on port 8124. The `listen` function causes the server to start listening and arranges to dispatch an event for every request arriving from a web browser.

The `request` event is fired any time an HTTP request arrives on the server. It takes a function that receives the `request` and `response` objects. The `request` object has data from the web browser, while the `response` object is used to gather data to be sent in the response.

Now, let's look at a server application that performs different actions based on the URL.

Create a new file named `server.mjs`, containing the following code:

```
import * as http from 'http';
import * as util from 'util';
import * as os from 'os';

const listenOn = 'http://localhost:8124';
const server = http.createServer();
server.on('request', (req, res) => {
    var requrl = new URL(req.url, listenOn);
    if (requrl.pathname === '/') homePage(req, res);
    else if (requrl.pathname === "/osinfo") osInfo(req, res);
    else {
        res.writeHead(404, {'Content-Type': 'text/plain'});
        res.end("bad URL " + req.url);
    }
});

server.listen(new URL(listenOn).port);
console.log(`listening to ${listenOn}`);

function homePage(req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(
`<html><head><title>Hello, world!</title></head>
<body><h1>Hello, world!</h1>
<p><a href='/osinfo'>OS Info</a></p>
</body></html>`);
}

function osInfo(req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end(
`<html><head><title>Operating System Info</title></head>
<body><h1>Operating System Info</h1>
<table>
<tr><th>TMP Dir</th><td>${os.tmpdir()}</td></tr>
<tr><th>Host Name</th><td>${os.hostname()}</td></tr>
<tr><th>OS Type</th><td>${os.type()} ${os.platform()}</td>
```

```
    ${os.arch()} ${os.release()}>/td></tr>
<tr><th>Uptime</th><td>${os.uptime() }  

${util.inspect(os.loadavg())}</td></tr>
<tr><th>Memory</th><td>total: ${os.totalmem()} free:  

${os.freemem()}</td></tr>
<tr><th>CPU's</th><td><pre>${util.inspect(os.cpus())}</pre></td></tr>
<tr><th>Network</th><td><pre>${util.inspect(os.networkInterfaces())}</pre></td></tr>
</table>
</body></html>` ;
}
```

The `request` event is emitted by `HTTPServer` every time a request arrives from a web browser. In this case, we want to respond differently based on the request URL, which arrives as `req.url`. This value is a string containing the URL from the HTTP request. Since there are many attributes to a URL, we need to parse the URL so that we can correctly match the pathname for one of two paths: `/` and `/osinfo`.

Parsing a URL with the `URL` class requires a **base URL**, which we've supplied in the `listenOn` variable. Notice how we're reusing this same variable in a couple of other places, using one string to configure multiple parts of the application.

Depending on the path, either the `homePage` or `osInfo` functions are called.

This is called **request routing**, where we look at attributes of the incoming request, such as the request path, and route the request to handler functions.

In the handler functions, the `req` and `res` parameters correspond to the `request` and `response` objects. Where `req` contains data about the incoming request, we send the response using `res`. The `writeHead` function sets up the return status (200 means success, while 404 means the page is not found) and the `end` function sends the response.

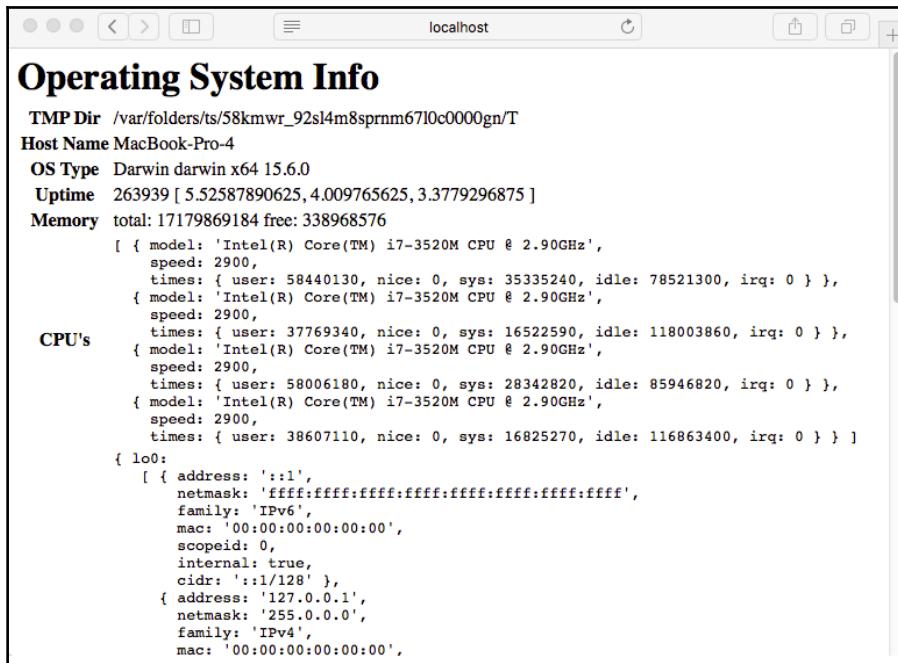
If the request URL is not recognized, the server sends back an error page using a 404 result code. The result code informs the browser about the status of the request, where a 200 code means everything is fine and a 404 code means the requested page doesn't exist. There are, of course, many other HTTP response codes, each with their own meaning.

There are plenty more functions attached to both objects, but that's enough to get us started.

To run it, type the following command:

```
$ node server.mjs
listening to http://localhost:8124
```

Then, if we paste the URL into a web browser, we see something like this:



This application is meant to be similar to PHP's `sysinfo` function. Node.js's `os` module is consulted to provide information about the computer. This example can easily be extended to gather other pieces of data.

A central part of any web application is the method of routing requests to request handlers. The `request` object has several pieces of data attached to it, two of which are useful for routing requests: the `request.url` and `request.method` fields.

In `server.mjs`, we consult the `request.url` data to determine which page to show after parsing using the `URL` object. Our needs are modest in this server, and a simple comparison of the `pathname` field is enough. Larger applications will use pattern matching to use part of the request URL to select the request handler function and other parts to extract request data out of the URL. We'll see this in action when we look at Express later in the *Getting started with Express* section.

Some web applications care about the HTTP verb that is used (GET, DELETE, POST, and so on) and so we must consult the `request.method` field of the `request` object. For example, POST is frequently used for any FORM submissions.

That gives us a taste of developing servers with Node.js. Along the way, we breezed past one big ES2015 feature—template strings. The template strings feature simplifies substituting values into strings. Let's see how that works.

ES2015 multiline and template strings

The previous example showed two of the new features introduced with ES2015: multiline and template strings. These features are meant to simplify our lives when creating text strings.

The existing JavaScript string representations use single quotes and double quotes. Template strings are delimited with the backtick character, which is also known as the **grave accent**:

```
`template string text`
```

Before ES2015, one way to implement a multiline string was to use the following construct:

```
[ "<html><head><title>Hello, world!</title></head>",
  "<body><h1>Hello, world!</h1>",
  "<p><a href='/osinfo'>OS Info</a></p>",
  "</body></html>"]
.join('\n')
```

This is an array of strings that uses the `join` function to smash them together into one string. Yes, this is the code used in the same example in previous versions of this book. This is what we can do with ES2015:

```
`<html><head><title>Hello, world!</title></head>
<body><h1>Hello, world!</h1>
<p><a href='/osinfo'>OS Info</a></p>
</body></html>`
```

This is more succinct and straightforward. The opening quote is on the first line, the closing quote is on the last line, and everything in between is part of our string.

The real purpose of the template strings feature is to support easily substituting values directly into strings. Many other programming languages support this ability, and now JavaScript does, too.

Pre-ES2015, a programmer would have written their code like this:

```
[ ...
  "<tr><th>OS Type</th><td>{ostype} {osplat} {osarch}
{osrelease}</td></tr>"
  ... ].join('\n')
.replace("{ostype}", os.type())
.replace("{osplat}", os.platform())
.replace("{osarch}", os.arch())
.replace("{osrelease}", os.release())
```

Similar to the previous snippet, this relied on the `replace` function to insert values into the string. Again, this is extracted from the same example that was used in previous versions of this book. With template strings, this can be written as follows:

```
`...<tr><th>OS Type</th><td>${os.type()} ${os.platform()} ${os.arch()}
${os.release()}</td></tr>...`
```

Within a template string, the part within the `${ ... }` brackets is interpreted as an expression. This can be a simple mathematical expression, a variable reference, or, as in this case, a function call.

Using template strings to insert data carries a security risk. Have you verified that the data is safe? Will it form the basis of a security attack? As always, data coming from an untrusted source, such as user input, must be properly encoded for the target context where the data is being inserted. In the example here, we should have used a function to encode this data as HTML, perhaps. But for this case, the data is in the form of simple strings and numbers and comes from a known, safe data source—the built-in `os` module—and so we know that this application is safe.

For this and many other reasons, it is often safer to use an external template engine. Applications such as Express make it easy to do so.

We now have a simple HTTP-based web application. To gain more experience with HTTP events, let's add to one to a module for listening to all HTTP events.

HTTP Sniffer – listening to the HTTP conversation

The events emitted by the `HTTPServer` object can be used for additional purposes beyond the immediate task of delivering a web application. The following code demonstrates a useful module that listens to all of the `HTTPServer` events. It could be a useful debugging tool, which also demonstrates how `HTTPServer` objects operate.

Node.js's `HTTPServer` object is an `EventEmitter` object, and HTTP Sniffer simply listens to every server event, printing out information pertinent to each event.

Create a file named `httpsniffer.mjs`, containing the following code:

```
import * as util from 'util';
import * as url from 'url';

const timestamp = () => { return new Date().toISOString(); }

export function sniffOn(server) {
    server.on('request', (req, res) => {
        console.log(`[${timestamp()}] request`);
        console.log(`[${timestamp()}] ${reqToString(req)}`);
    });

    server.on('close', errno => { console.log(`[${timestamp()}]
        close errno=${errno}`); });

    server.on('checkContinue', (req, res) => {
        console.log(`[${timestamp()}] checkContinue`);
        console.log(`[${timestamp()}] ${reqToString(req)}`);
        res.writeContinue();
    });

    server.on('upgrade', (req, socket, head) => {
        console.log(`[${timestamp()}] upgrade`);
        console.log(`[${timestamp()}] ${reqToString(req)}`);
    });

    server.on('clientError', () => { console.log('clientError'); });
}

// server.on('connection', e_connection);
}

export function reqToString(req) {
```

```
var ret = `request ${req.method} ${req.httpVersion} ${req.url}`  
    +'\\n';  
ret += JSON.stringify(url.parse(req.url, true)) +'\\n';  
var keys = Object.keys(req.headers);  
for (var i = 0, l = keys.length; i < l; i++) {  
    var key = keys[i];  
    ret += `${i} ${key}: ${req.headers[key]}` +'\\n';  
}  
if (req.trailers)  
    ret += util.inspect(req.trailers) +'\\n';  
return ret;  
}
```

The key here is the `sniffOn` function. When given an `HTTPServer` object, it attaches listener functions to each `HTTPServer` event to print relevant data. This gives us a fairly detailed trace of the HTTP traffic on an application.

In order to use it, make two simple modifications to `server.mjs`. To the top, add the following `import` statement:

```
import { sniffOn } from '../events/httpsniffer.mjs';
```

Then, change the server setup, as follows:

```
server.listen(new URL(listenOn).port);  
sniffOn(server);  
console.log(`listening to ${listenOn}`);
```

Here, we're importing the `sniffOn` function and then using it to attach listener methods to the `server` object.

With this in place, run the server as we did earlier. You can visit `http://localhost:8124/` in your browser and see the following console output:

```
$ node server.mjs  
listening to http://localhost:8124  
2020-01-05T02:33:09.864Z request  
2020-01-05T02:33:09.868Z request GET 1.1 /osinfo  
{"protocol":null,"slashes":null,"auth":null,"host":null,"port":null,"h  
ostname":null,"hash":null,"search":null,"query":{},"pathname":"/osinfo  
","path":"/osinfo","href":"/osinfo"}  
0 host: localhost:8124  
1 connection: keep-alive  
2 cache-control: max-age=0  
3 dnt: 1  
4 upgrade-insecure-requests: 1  
5 user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6)
```

```
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108
Safari/537.36
6 sec-fetch-user: ?1
7 accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image
/apng,*/*;q=0.8,application/signed-exchange;v=b3
8 sec-fetch-site: same-origin
9 sec-fetch-mode: navigate
10 referer: http://localhost:8124/
11 accept-encoding: gzip, deflate, br
12 accept-language: en-US,en;q=0.9
{}
```

You now have a tool for snooping on `HTTPServer` events. This simple technique prints a detailed log of event data. This pattern can be used for any `EventEmitter` objects. You can use this technique as a way to inspect the actual behavior of `EventEmitter` objects in your program.

Before we move on to using Express, we need to discuss why we use application frameworks at all.

Web application frameworks

The `HTTPServer` object is very close to the HTTP protocol. While this is powerful in the same way that driving a stick shift car gives you low-level control over the driving experience, typical web application programming is better done at a higher level. Does anyone use assembly language to write web applications? It's better to abstract away the HTTP details and concentrate on your application.

The Node.js developer community has developed quite a few application frameworks to help with different aspects of abstracting away HTTP protocol details. Of these frameworks, Express is the most popular, and Koa (<http://koajs.com/>) should be considered because it has fully integrated support for `async` functions.

The Express.js wiki has a list of frameworks built on top of Express.js or tools that work with it. This includes template engines, middleware modules, and more. The Express.js wiki is located at <https://github.com/expressjs/express/wiki>.

One reason to use a web framework is that they often have well-tested implementations of the best practices used in web application development for over 20 years. The usual best practices include the following:

- Providing a page for bad URLs (the 404 page)
- Screening URLs and forms for any injected scripting attacks
- Supporting the use of cookies to maintain sessions
- Logging requests for both usage tracking and debugging
- Authentication
- Handling static files, such as images, CSS, JavaScript, or HTML
- Providing cache-control headers to caching proxies
- Limiting things such as the page size or execution time

Web frameworks help you invest your time in a task without getting lost in the details of implementing the HTTP protocol. Abstracting away details is a time-honored way for programmers to be more efficient. This is especially true when using a library or framework that provides prepackaged functions that take care of the details.

With that in mind, let's turn to a simple application implemented with Express.

Getting started with Express

Express is perhaps the most popular Node.js web app framework. Express is described as being Sinatra-like, which refers to a popular Ruby application framework. It is also regarded as not being an opinionated framework, meaning the framework authors don't impose their opinions about structuring an application. This means Express is not at all strict about how your code is structured; you just write it the way you think is best.



You can visit the home page for Express at <http://expressjs.com/>.

As of the time of writing this book, Express 4.17 is the current version, and Express 5 is in alpha testing. According to the Express.js website, there are very few differences between Express 4 and Express 5.

Let's start by installing `express-generator`. While we can just start with writing some code, `express-generator` provides a blank starting application, which we'll take and modify.

Install `express-generator` using the following commands:

```
$ mkdir fibonacci  
$ cd fibonacci  
$ npm install express-generator@4.x
```

This is different from the suggested installation method on the Express website, which says to use the `-g` tag for a global installation. We're also using an explicit version number to ensure compatibility. As of the time of writing, `express-generator@5.x` does not exist, but it should exist sometime in the future. The instructions here are written for Express 4.x, and by explicitly naming the version, we're ensuring that we're all on the same page.

Earlier, we discussed how many people now recommend against installing modules globally. Maybe they would consider `express-generator` as an exception to that rule, or maybe not. In any case, we're not following the recommendation on the Express website, and toward the end of this section, we'll have to uninstall `express-generator`.

The result of this is that an `express` command is installed in the `./node_modules/.bin` directory:

```
$ ls node_modules/.bin/  
express
```

Run the `express` command, as follows:

```
$ ./node_modules/.bin/express --help  
  
Usage: express [options] [dir]  
  
Options:  
  
  --version output the version number  
  -e, --ejs add ejs engine support  
  --pug add pug engine support  
  --hbs add handlebars engine support  
  -H, --hogan add hogan.js engine support  
  -v, --view <engine> add view <engine> support  
  (dust|ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)  
  --no-view use static html instead of view engine  
  -c, --css <engine> add stylesheet <engine> support
```

```
(less|stylus|compass|sass) (defaults to plain css)
  --git add .gitignore
  -f, --force force on non-empty directory
  -h, --help output usage information
```

We probably don't want to type `./node_modules/.bin/express` every time we run the `express-generator` application, or, for that matter, any of the other applications that provide command-line utilities. Refer back to the discussion we had in Chapter 3, *Exploring Node.js Modules*, about adding this directory to the `PATH` variable. Alternatively, the `npx` command, also described in Chapter 3, *Exploring Node.js Modules*, is useful for this.

For example, try using the following instead of installing `express-generator`:

```
$ npx express-generator@4.x --help
npx: installed 10 in 4.26s

Usage: express [options] [dir]
...
```

This executes exactly the same, without having to install `express-generator` and (as we'll see in a moment) remembering to uninstall it when you're done using the command.

Now that you've installed `express-generator` in the `fibonacci` directory, use it to set up the blank framework application:

```
$ express --view=hbs --git .
destination is not empty, continue? [y/N] y

create : public/
create : public/javascripts/
create : public/images/
create : public/stylesheets/
create : public/stylesheets/style.css
create : routes/
create : routes/index.js
create : routes/users.js
create : views/
create : views/error.hbs
create : views/index.hbs
create : views/layout.hbs
create : .gitignore
create : app.js
create : package.json
create : bin/
create : bin/www
```

```
install dependencies:  
$ npm install  
  
run the app:  
$ DEBUG=fibonacci:* npm start
```

This creates a bunch of files for us, which we'll walk through in a minute. We asked it to initialize the use of the Handlebars template engine and to initialize a git repository.

The node_modules directory still has the express-generator module, which is no longer useful. We can just leave it there and ignore it, or we can add it to devDependencies of the package.json file that it generated. Most likely, we will want to uninstall it:

```
$ npm uninstall express-generator  
added 62 packages from 78 contributors, removed 9 packages and audited  
152 packages in 4.567s
```

This uninstalls the express-generator tool. The next thing to do is to run the blank application in the way that we're told. The npm start command relies on a section of the supplied package.json file:

```
"scripts": {  
  "start": "node ./bin/www"  
},
```

It's cool that the Express team showed us how to run the server by initializing the scripts section in package.json. The start script is one of the scripts that correspond to the npm sub-commands. The instructions we were given, therefore, say to run npm start.

The steps are as follows:

1. Install the dependencies with npm install.
2. Start the application by using npm start.
3. Optionally, modify package.json to always run with debugging.

To install the dependencies and run the application, type the following commands:

```
$ npm install  
$ DEBUG=fibonacci:* npm start  
  
> fibonacci@0.0.0 start /Users/David/chap04/fibonacci
```

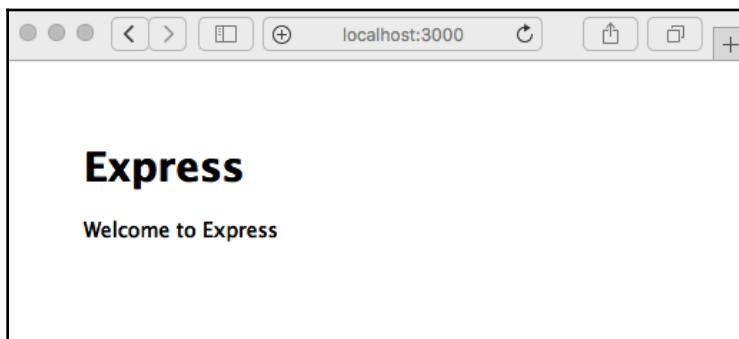
```
> node ./bin/www  
fibonacci:server Listening on port 3000 +0ms
```

Setting the `DEBUG` variable this way turns on the debugging output, which includes a message about listening on port 3000. Otherwise, we aren't told this information. This syntax is what's used in the Bash shell to run a command with an environment variable. If you get an error when running `npm start`, then refer to the next section.

We can modify the supplied `npm start` script to always run the app with debugging enabled. Change the `scripts` section to the following:

```
"scripts": {  
  "start": "DEBUG=fibonacci:* node ./bin/www"  
},
```

Since the output says it is listening on port 3000, we direct our browser to `http://localhost:3000/` and see the following output:



Cool, we have some running code. Before we start changing the code, we need to discuss how to set environment variables in Windows.

Setting environment variables in the Windows cmd.exe command line

If you're using Windows, the previous example may have failed, displaying an error that says `DEBUG` is not a known command. The problem is that the Windows shell, the `cmd.exe` program, does not support the Bash command-line structure.

Adding `VARIABLE=value` to the beginning of a command line is specific to some shells, such as Bash, on Linux and macOS. It sets that environment variable only for the command line that is being executed and is a very convenient way to temporarily override environment variables for a specific command.

Clearly, a solution is required if you want to be able to use your `package.json` file across different operating systems.



The best solution appears to be using the `cross-env` package in the npm repository; refer to <https://www.npmjs.com/package/cross-env> for more information.

With this package installed, commands in the `scripts` section in `package.json` can set environment variables just as in Bash on Linux/macOS. The use of this package looks as follows:

```
"scripts": {  
  "start": "cross-env DEBUG=fibonacci:* node ./bin/www"  
},  
"dependencies": {  
  ...  
  "cross-env": "^6.0.3"  
}
```

Then, the command is executed, as follows:

```
C:\Users\david\Documents\chap04\fibonacci>npm install  
... output from installing packages  
C:\Users\david\Documents\chap04\fibonacci>npm run start  
  
> fibonacci@0.0.0 start C:\Users\david\Documents\chap04\fibonacci  
> cross-env DEBUG=fibonacci:* node ./bin/www  
  
fibonacci:server Listening on port 3000 +0ms  
GET / 304 90.597 ms --  
GET /stylesheets/style.css 304 14.480 ms --
```

We now have a simple way to ensure the scripts in `package.json` are cross-platform. Our next step is a quick walkthrough of the generated application.

Walking through the default Express application

We now have a working, blank Express application; let's look at what was generated for us. We do this to familiarize ourselves with Express before diving in to start coding our **Fibonacci** application.

Because we used the `--view=hbs` option, this application is set up to use the Handlebars.js template engine.



For more information about Handlebars.js, refer to its home page at <http://handlebarsjs.com/>. The version shown here has been packaged for use with Express and is documented at <https://github.com/pillarjs/hbs>.

Generally speaking, a template engine makes it possible to insert data into generated web pages. The Express.js wiki has a list of template engines for Express (<https://github.com/expressjs/express/wiki#template-engines>).

Notice that the JavaScript files are generated as CommonJS modules. The `views` directory contains two files—`error.hbs` and `index.hbs`. The `hbs` extension is used for Handlebars files. Another file, `layout.hbs`, is the default page layout. Handlebars has several ways to configure layout templates and even partials (snippets of code that can be included anywhere).

The `routes` directory contains the initial routing setup—that is, code to handle specific URLs. We'll modify this later.

The `public` directory contains assets that the application doesn't generate but are simply sent to the browser. What's initially installed is a CSS file, `public/stylesheets/style.css`. The `package.json` file contains our dependencies and other metadata.

The `bin` directory contains the `www` script that we saw earlier. This is a Node.js script that initializes the `HTTPServer` objects, starts listening on a TCP port, and calls the last file that we'll discuss, `app.js`. These scripts initialize Express and hook up the routing modules, as well as other things.

There's a lot going on in the `www` and `app.js` scripts, so let's start with the application initialization. Let's first take a look at a couple of lines in `app.js`:

```
const express = require('express');
...
const app = express();
...
module.exports = app;
...
```

This means that `app.js` is a CommonJS module that exports the application object generated by the `express` module. Our task in `app.js` is to configure that application object. This task does not include starting the `HTTPServer` object, however.

Now, let's turn to the `bin/www` script. It is in this script where the HTTP server is started. The first thing to notice is that it starts with the following line:

```
#!/usr/bin/env node
```

This is a Unix/Linux technique to make a command script. It says to run the following as a script using the `node` command. In other words, we have Node.js code and we're instructing the operating system to execute that code using the Node.js runtime:

```
$ ls -l bin/www
-rwx----- 1 david  staff  1595 Feb  5  1970 bin/www
```

We can also see that the script was made executable by `express-generator`.

It calls the `app.js` module, as follows:

```
var app = require('../app');
...
var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);
...
var server = http.createServer(app);
...
server.listen(port);
server.on('error', onError);
server.on('listening', onListening);
```

Namely, it loads the module in `app.js`, gives it a port number to use, creates the `HTTPServer` object, and starts it up.

We can see where port 3000 comes from; it's a parameter to the `normalizePort` function. We can also see that setting the `PORT` environment variable will override the default port 3000. Finally, we can see that the `HTTPServer` object is created here and is told to use the application instance created in `app.js`. Try running the following command:

```
$ PORT=4242 DEBUG=fibonacci:* npm start
```

By specifying an environment variable for `PORT`, we can tell the application to listen in on port 4242, where you can ponder the meaning of life.

The `app` object is next passed to `http.createServer()`. A look at the Node.js documentation tells us that this function takes `requestListener`, which is simply a function that takes the `request` and `response` objects that we saw previously. Therefore, the `app` object is the same kind of function.

Finally, the `bin/www` script starts the server listening process on the port we specified.

Let's now go through `app.js` in more detail:

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');
```

This tells Express to look for templates in the `views` directory and to use the Handlebars templating engine.

The `app.set` function is used to set the application properties. It'll be useful to browse the API documentation as we go through (<http://expressjs.com/en/4x/api.html>).

Next is a series of `app.use` calls:

```
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', indexRouter);
app.use('/users', usersRouter);
```

The `app.use` function mounts middleware functions. This is an important piece of Express jargon, which we will discuss shortly. At the moment, let's say that middleware functions are executed during the processing of requests. This means all the features named here are enabled in `app.js`:

- Logging is enabled using the `morgan` request logger. Refer to <https://www.npmjs.com/package/morgan> for its documentation.
- The `body-parser` module handles parsing HTTP request bodies. Refer to <https://www.npmjs.com/package/body-parser> for its documentation.
- The `cookie-parser` module is used to parse HTTP cookies. Refer to <https://www.npmjs.com/package/cookie-parser> for its documentation.
- A static file web server is configured to serve the asset files in the `public` directory. Refer to <http://expressjs.com/en/starter/static-files.html> for its documentation.
- Two router modules—`routes` and `users`—to set up which functions handle which URLs.

The static file web server arranges to serve, via HTTP requests, the files in the named directory. With the configuration shown here, the `public/stylesheets/style.css` file is available at `http://HOST/stylesheets/style.css`.

We shouldn't feel limited to setting up an Express application this way. This is the recommendation of the Express team, but there is nothing constraining us from setting it up another way. For example, later in this book, we'll rewrite this entirely as ES6 modules, rather than sticking to CommonJS modules. One glaring omission is handlers for uncaught exceptions and unhandled Promise rejections. We'll go over both of these later in this book.

Next, we will discuss Express **middleware** functions.

Understanding Express middleware

Let's round out our walkthrough of `app.js` by discussing what Express middleware functions do for our application. Middleware functions are involved in processing requests and sending results to HTTP clients. They have access to the `request` and `response` objects and are expected to process their data and perhaps add data to these objects. For example, the cookie parser middleware parses HTTP cookie headers to record in the `request` object the cookies sent by the browser.

We have an example of this at the end of our script:

```
app.use(function(req, res, next) {  
  const err = new Error('Not found');  
  err.status = 404;  
  next(err);  
});
```

The comment says catch 404 and forward it to the error handler. As you probably know, an HTTP 404 status means the requested resource was not found. We need to tell the user that their request wasn't satisfied, and maybe show them something such as a picture of a flock of birds pulling a whale out of the ocean. This is the first step in doing this. Before getting to the last step of reporting this error, you need to learn how middleware works.

The name *middleware* implies software that executes in the middle of a chain of processing steps.



Refer to the documentation about middleware at
<http://expressjs.com/en/guide/writing-middleware.html>.

Middleware functions take three arguments. The first two—request and response—are equivalent to the `request` and `response` objects of the Node.js HTTP request object. Express expands these objects with additional data and capabilities. The last argument, `next`, is a callback function that controls when the request-response cycle ends, and it can be used to send errors down the middleware pipeline.

As an aside, one critique of Express is that it was written prior to the existence of Promises and async functions. Therefore, its design is fully enmeshed with the callback function pattern. We can still use async functions, but integrating with Express requires using the callback functions it provides.

The overall architecture is set up so that incoming requests are handled by zero or more middleware functions, followed by a router function, which sends the response. The middleware functions call `next`, and in a normal case, provide no arguments by calling `next()`. If there is an error, the middleware function indicates the error by calling `next(err)`, as shown here.

For each middleware function that executes, there is, in theory, several other middleware functions that have already been executed, and potentially several other functions still to be run. It is required to call `next` to pass control to the next middleware function.

What happens if `next` is not called? There is one case where we must not call `next`. In all other cases, if `next` is not called, the HTTP request will hang because no response will be given.

What is the one case where we must not call `next`? Consider the following hypothetical router function:

```
app.get('/hello', function(req, res) {  
    res.send('Hello World!');  
});
```

This does not call `next` but instead calls `res.send`. The HTTP response is sent for certain functions on the `response` object, such as `res.send` or `res.render`. This is the correct method for ending the request-response cycle, by sending a response (`res.send`) to the request. If neither `next` nor `res.send` are called, the request never gets a response and the requesting client will hang.

So, a middleware function does one of the following four things:

- Executes its own business logic. The request logger middleware shown earlier is an example of this.
- Modifies the `request` or `response` objects. Both `body-parser` and `cookie-parser` do this, looking for data to add to the `request` object.
- Calls `next` to proceed to the next middleware function or otherwise signals an error.
- Sends a response, ending the cycle.

The ordering of middleware execution depends on the order that they're added to the `app` object. The first function added is executed first, and so on.

The next thing to understand is request handlers and how they differ from middleware functions.

Contrasting middleware and request handlers

We've seen two kinds of middleware functions so far. In one, the first argument is the handler function. In the other, the first argument is a string containing a URL snippet and the second argument is the handler function.

What's actually going on is `app.use` has an optional first argument: the path that the middleware is mounted on. The path is a pattern match against the request URL, and the given function is triggered if the URL matches the pattern. There's even a method to supply named parameters in the URL:

```
app.use('/user/profile/:id', function(req, res, next) {
  userProfiles.lookup(req.params.id, (err, profile) => {
    if (err) return next(err);
    // do something with the profile
    // Such as display it to the user
    res.send(profile.display());
  });
});
```

This path specification has a pattern, `:id`, and the value will land in `req.params.id`. In an Express route, this `:id` pattern marks a **route parameter**. The pattern will match a URL segment, and the matching URL content will land and be available through the `req.params` object. In this example, we're suggesting a user profile service and that for this URL, we want to display information about the named user.

As Express scans the available functions to execute, it will try to match this pattern against the request URL. If they match, then the router function is invoked.

It is also possible to match based on the HTTP request method, such as GET or PUT. Instead of `app.use`, we would write `app.METHOD`—for example, `app.get` or `app.put`. The preceding example would, therefore, be more likely to appear as follows:

```
app.get('/user/profile/:id', function(req, res, next) {
  // ... as above
});
```

The required behavior of GET is to retrieve data, while the behavior of PUT is to store data. However, as the example was written above, it would match either of the HTTP methods when the handler function is only correct for the GET verb. However, using `app.get`, as is the case here, ensures that the application correctly matches the desired HTTP method.

Finally, we get to the `Router` object. This is the kind of middleware used explicitly for routing requests based on their URL. Take a look at `routes/users.js`:

```
const express = require('express');
const router = express.Router();
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});
module.exports = router;
```

We have a module that creates a `router` object, then adds one or more `router` functions. It makes the `Router` object available through `module.exports` so that `app.js` can use it. This `router` has only one route, but `router` objects can have any number of routes that you think is appropriate.

This one route matches a GET request on the `/` URL. That's fine until you notice that in `routes/index.js`, there is a similar `router` function that also matches GET requests on the `/` URL.

Back in `app.js`, `usersRouter` is added, as follows:

```
app.use('/users', usersRouter);
```

This takes the `router` object, with its zero-or-more `router` functions, and mounts it on the `/users` URL. As Express looks for a matching routing function, it first scans the functions attached to the `app` object, and for any `router` object, it scans its functions as well. It then invokes any routing functions that match the request.

Going back to the issue of the `/` URL, the fact that the `router` is *mounted on* the `/users` URL is important. That's because the actual URL it considers matching is the mount point (`/users`) concatenated with the URL in the `router` function.

The effect is that the mount prefix is stripped from the request URL for the purpose of matching against the `router` functions attached to the `router` object. So, with that mount point, an incoming URL of `/users/login` would be stripped to just `/login` in order to find a matching `router` function.

Since not everything goes according to plan, our applications must be capable of handling error indications and showing error messages to users.

Error handling

Now, we can finally get back to the generated `app.js` file, the 404 Error page not found error, and any other errors that the application might show to the user.

A middleware function indicates an error by passing a value to the `next` function call, namely by calling `next(err)`. Once Express sees the error, it skips over any remaining non-error routings and only passes the error to error handlers instead. An error handler function has a different signature than what we saw earlier.

In `app.js`, which we're examining, the following is our error handler, provided by `express-generator`:

```
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};

  res.status(err.status || 500);
  res.render('error');
});
```

Error handler functions take four parameters, with `err` added to the familiar `req`, `res`, and `next` functions.

Remember that `res` is the response object, and we use it to set up the HTTP response sent to the browser; even though there is an error, we still send a response.

Using `res.status` sets the HTTP response status code. In the simple application that we examined earlier, we used `res.writeHead` to set not only the status code but also the **Multipurpose Internet Mail Extensions (MIME)** type of the response.

The `res.render` function takes data and renders it through a template. In this case, we're using the template named `error`. This corresponds to the `views/error.hbs` file, which looks as follows:

```
<h1>{{message}}</h1>
<h2>{{error.status}}</h2>
<pre>{{error.stack}}</pre>
```

In a Handlebars template, the `{{value}}` markup means to substitute into the template the value of the expression or variable. The values referenced by this template—`message` and `error`—are provided by setting `res.locals` as shown here.

To see the error handler in action, let's add the following to `routes/index.js`:

```
router.get('/error', function(req, res, next) {
  next({
    status: 404,
    message: "Fake error"
  });
});
```

This is a route handler, and going by what we've said, it simply generates an error indication. In a real route handler, the code would make some kind of query, gathering up data to show to the user, and it would indicate an error only if something happened along the way. However, we want to see the error handler in action.

By calling `next (err)`, as mentioned, Express will call the error handler function, causing an error response to pop up in the browser:



Indeed, at the `/error` URL, we get the **Fake error** message, which matches the error data sent by the route handler function.

In this section, we've created for ourselves a foundation for how Express works. Let's now turn to an Express application that actually performs a function.

Creating an Express application to compute Fibonacci numbers

As we discussed in Chapter 1, *About Node.js* we'll be using an inefficient algorithm to calculate Fibonacci numbers to explore how to mitigate performance problems, and along the way, we'll learn how to build a simple REST service to offload computation to the backend server.

The Fibonacci numbers are the following integer sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Each Fibonacci number is the sum of the previous two numbers in the sequence. This sequence was discovered in 1202 by Leonardo of Pisa, who was also known as Fibonacci. One method to calculate entries in the Fibonacci sequence is using the recursive algorithm, which we discussed in Chapter 1, *About Node.js*. We will create an Express application that uses the Fibonacci implementation and along the way, we will get a better understanding of Express applications, as well as explore several methods to mitigate performance problems in computationally intensive algorithms.

Let's start with the blank application we created in the previous step. We named that application `Fibonacci` for a reason—we were thinking ahead!

In `app.js`, make the following changes to the top portion of the file:

```
const express = require('express');
const hbs = require('hbs');
const path = require('path');
const favicon = require('serve-favicon');
const logger = require('morgan');
const cookieParser = require('cookie-parser');
const bodyParser = require('body-parser');

const indexRouter = require('./routes/index');
const fibonacciRouter = require('./routes/fibonacci');

const app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');
hbs.registerPartials(path.join(__dirname, 'partials'));

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', indexRouter);
app.use('/fibonacci', fibonacciRouter);
```

Most of this is what express-generator gave us. The `var` statements have been changed to `const` for that little teensy bit of extra comfort. We explicitly imported the `hbs` module so that we could do some configuration. We also imported a router module for Fibonacci, which we'll see in a minute.

For the Fibonacci application, we don't need to support users and so we have deleted the routing module. The `routes/fibonacci.js` module, which we'll show next, serves to query a number for which we'll calculate the Fibonacci number.

In the top-level directory, create a file, `math.js`, containing the following extremely simplistic Fibonacci implementation:

```
exports.fibonacci = function(n) {
  if (n === 0) return 0;
  else if (n === 1 || n === 2) return 1;
  else return exports.fibonacci(n-1) + exports.fibonacci(n-2);
};
```

In the `views` directory, look at the file named `layout.hbs`, which was created by express-generator:

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{title}}</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    {{body}}
  </body>
</html>
```

This file contains the structure that we'll use for the HTML pages. Going by the Handlebars syntax, we can see that `{{title}}` appears within the HTML `title` tag. This means that when we call `res.render`, we should supply a `title` attribute. The `{{body}}` tag is where the view template content lands.

Change `views/index.hbs` to just contain the following:

```
<h1>{{title}}</h1>
{{> navbar}}
<p>Welcome to {{title}}</p>
```

This serves as the front page of our application. It will be inserted in place of `{{{body}}}` in `views/layout.hbs`. The marker, `{{> navbar}}`, refers to a partially named navbar object. Earlier, we configured a directory named `partials` to hold partials. Now, let's create a file, `partials/navbar.html`, containing the following:

```
<div class='navbar'>
<p><a href='/'>home</a> | <a href='/fibonacci'>Fibonacci's</a></p>
</div>
```

This will serve as a navigation bar that's included on every page.

Create a file, `views/fibonacci.hbs`, containing the following code:

```
<h1>{{title}}</h1>
{{> navbar}}
{{#if fiboval}}
  <p>Fibonacci for {{fibonum}} is {{fiboval}}</p>
  <hr/>
{{/if}}
<p>Enter a number to see its' Fibonacci number</p>
<form name='fibonacci' action='/fibonacci' method='get'>
<input type='text' name='fibonum' />
<input type='submit' value='Submit' />
</form>
```

If `fiboval` is set, this renders a message that for a given number (`fibonum`), we have calculated the corresponding Fibonacci number. There is also an HTML form that we can use to enter a `fibonum` value.

Because it is a GET form, when the user clicks on the **Submit** button, the browser will issue an HTTP GET method to the `/fibonacci` URL. What distinguishes one GET method on `/fibonacci` from another is whether the URL contains a query parameter named `fibonum`. When the user first enters the page, there is no `fibonum` number and so there is nothing to calculate. After the user has entered a number and clicked on **Submit**, there is a `fibonum` number and so something to calculate.



Remember that the files in `views` are templates into which data is rendered. They serve the **view** aspect of the **Model-View-Controller (MVC)** paradigm, hence the directory name.

In `routes/index.js`, change the `router` function to the following:

```
router.get('/', function(req, res, next) {
  res.render('index', { title: "Welcome to the Fibonacci Calculator"
});});
```

The anonymous object passed to `res.render` contains the data values we provide to the layout and view templates. We're now passing a new welcome message.

Then, finally, in the `routes` directory, create a file named `fibonacci.js`, containing the following code:

```
const express = require('express');
const router = express.Router();

const math = require('../math');
router.get('/', function(req, res, next) {
  if (req.query.fibonum) {
    // Calculate directly in this server
    res.render('fibonacci', {
      title: "Calculate Fibonacci numbers",
      fibonum: req.query.fibonum,
      fiboval: math.fibonacci(req.query.fibonum)
    });
  } else {
    res.render('fibonacci', {
      title: "Calculate Fibonacci numbers",
      fiboval: undefined
    });
  }
});

module.exports = router;
```

This route handler says it matches the `/` route. However, there is a route handler in `index.js` that matches the same route. We haven't made a mistake, however. The `router` object created by this module becomes `fibonacciRouter` when it lands in `app.js`. Refer back to `app.js` and you will see that `fibonacciRouter` is mounted on `/fibonacci`. The rule is that the actual URL path matched by a router function is the path that the router is mounted on plus the path given for the router function. In this case, that is `/fibonacci` plus `/`, and for a URL, that equates to `/fibonacci`.

The handler checks for the existence of `req.query.fibonum`. Express automatically parses the HTTP request URL and any query parameters will land in `req.query`. Therefore, this will trigger a URL such as `/fibonacci?fibonum=5`.

If this value is present, then we call `res.render('fibonacci')` with data including `fibonum`, the number for which we want its Fibonacci number, and `fiboval`, the corresponding Fibonacci number. Otherwise, we pass `undefined` for `fiboval`. If you refer back to the template, if `fiboval` is not set, then the user only sees the form to enter a `fibonum` number. Otherwise, if `fiboval` is set, both `fibonum` and `fiboval` are displayed.

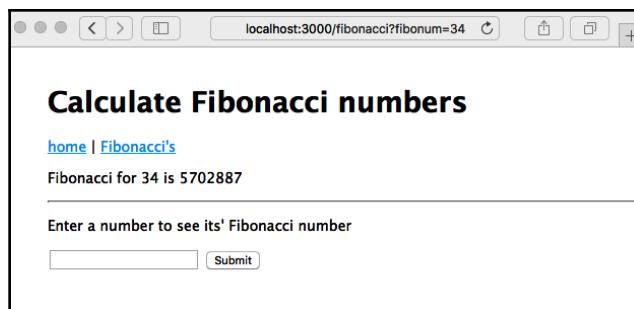
The package.json file is already set up, so we can use `npm start` to run the script and always have debugging messages enabled. Now, we're ready to do this:

```
$ npm start
> fibonacci@0.0.0 start /Users/david/chap04/fibonacci
> DEBUG=fibonacci:* node ./bin/www
fibonacci:server Listening on port 3000 +0ms
```

As this suggests, you can visit `http://localhost:3000/` and see what we have:



This page is rendered from the `views/index.hbs` template. Simply click on the **Fibonacci's** link to go to the next page, which is, of course, rendered from the `views/fibonacci.hbs` template. On that page, you'll be able to enter a number, click on the **Submit** button, and get an answer (hint—pick a number below 40 if you want your answer in a reasonable amount of time):



We asked you to enter a number less than 40. Go ahead and enter a larger number, such as 50, but go take a coffee break because this is going to take a while to calculate. Or, proceed on to reading the next section, where we will start to discuss the use of computationally intensive code.

Computationally intensive code and the Node.js event loop

This Fibonacci example is purposely inefficient to demonstrate an important consideration for your applications. What happens to the Node.js event loop when long computations are run? To see the effect, open two browser windows, each viewing the Fibonacci page. In one, enter the number 55 or greater, and in the other, enter 10. Note how the second window freezes, and if you leave it running long enough, the answer will eventually pop up in both windows. What's happening in the Node.js event loop is blocked from processing events because the Fibonacci algorithm is running and does not ever yield to the event loop.

Since Node.js has a single execution thread, processing requests depends on request handlers quickly returning to the event loop. Normally, the asynchronous coding style ensures that the event loop executes regularly.

This is true even for requests that load data from a server halfway around the globe because the asynchronous request is non-blocking and control is quickly returned to the event loop. The naive Fibonacci function we chose doesn't fit into this model because it's a long-running blocking operation. This type of event handler prevents the system from processing requests and stops Node.js from doing what it's meant to do—namely, to be a blisteringly fast web server.

In this case, the long-response-time problem is obvious. The response time to calculate a Fibonacci number quickly escalates to the point where you can take a vacation to Tibet, become a Lama, and perhaps get reincarnated as a llama in Peru in the time it takes to respond! However, it's also possible to create a long-response-time problem without it being as obvious as this one. Of the zillion-and-one asynchronous operations in a large web service, which one is both blocking and takes a long time to compute the result? Any blocking operations like this will cause a negative effect on the server throughput.

To see this more clearly, create a file named `fibotimes.js`, containing the following code:

```
const math = require('./math');
for (var num = 1; num < 80; num++) {
  let now = new Date().toISOString();
  console.log(` ${now} Fibonacci for ${num} = ${math.fibonacci(num)} `);
}
```

Now, run it. You will get the following output:

```
$ node fibotimes.js
2020-01-06T00:26:36.092Z Fibonacci for 1 = 1
2020-01-06T00:26:36.105Z Fibonacci for 2 = 1
2020-01-06T00:26:36.105Z Fibonacci for 3 = 2
2020-01-06T00:26:36.105Z Fibonacci for 4 = 3
2020-01-06T00:26:36.105Z Fibonacci for 5 = 5
...
2020-01-06T00:26:36.106Z Fibonacci for 10 = 55
2020-01-06T00:26:36.106Z Fibonacci for 11 = 89
2020-01-06T00:26:36.106Z Fibonacci for 12 = 144
2020-01-06T00:26:36.106Z Fibonacci for 13 = 233
2020-01-06T00:26:36.106Z Fibonacci for 14 = 377
...
2020-01-06T00:26:37.895Z Fibonacci for 40 = 102334155
2020-01-06T00:26:38.994Z Fibonacci for 41 = 165580141
2020-01-06T00:26:40.792Z Fibonacci for 42 = 267914296
2020-01-06T00:26:43.699Z Fibonacci for 43 = 433494437
2020-01-06T00:26:48.985Z Fibonacci for 44 = 701408733
...
2020-01-06T00:33:45.968Z Fibonacci for 51 = 20365011074
2020-01-06T00:38:12.184Z Fibonacci for 52 = 32951280099
^C
```

This quickly calculates the first 40 or so members of the Fibonacci sequence, but after the 40th member, it starts taking a couple of seconds per result and quickly degrades from there. It is untenable to execute code of this sort on a single-threaded system that relies on a quick return to the event loop. A web service containing code like this would give a poor performance to the users.

There are two general ways to solve this problem in Node.js:

- **Algorithmic refactoring:** Perhaps, like the Fibonacci function we chose, one of your algorithms is suboptimal and can be rewritten to be faster. Or, if it is not faster, it can be split into callbacks dispatched through the event loop. We'll look at one method for this in a moment.
- **Creating a backend service:** Can you imagine a backend server that is dedicated to calculating Fibonacci numbers? Okay, maybe not, but it's quite common to implement backend servers to offload work from frontend servers, and we will implement a backend Fibonacci server at the end of this chapter.

With that in mind, let's examine these possibilities.

Algorithmic refactoring

To prove that we have an artificial problem on our hands, here is a much more efficient Fibonacci function:

```
exports.fibonacciLoop = function(n) {
  var fibos = [];
  fibos[0] = 0;
  fibos[1] = 1;
  fibos[2] = 1;
  for (let i = 3; i <= n; i++) {
    fibos[i] = fibos[i-2] + fibos[i-1];
  }
  return fibos[n];
}
```

If we substitute a call to `math.fibonacciLoop` in place of `math.fibonacci`, the `fibotimes` program runs much faster. Even this isn't the most efficient implementation; for example, a simple, prewired lookup table is much faster at the cost of some memory.

Edit `fibotimes.js` as follows and rerun the script. The numbers will fly by so fast that your head will spin:

```
for (var num = 1; num < 8000; num++) {
  let now = new Date().toISOString();
  console.log(` ${now} Fibonacci for ${num} = ${math.fibonacciLoop(num)} `);
}
```

Sometimes, your performance problems will be this easy to optimize, but other times, they won't.

The discussion here isn't about optimizing mathematics libraries but about dealing with inefficient algorithms that affect event throughput in a Node.js server. For that reason, we will stick with the inefficient Fibonacci implementation.

It is possible to divide the calculation into chunks and then dispatch the computation of those chunks through the event loop. Add the following code to `math.js`:

```
module.exports.fibonacciAsync = function(n, done) {
    if (n === 0) done(undefined, 0);
    else if (n === 1 || n === 2) done(undefined, 1);
    else {
        setImmediate(() => {
            exports.fibonacciAsync(n-1, (err, val1) => {
                if (err) done(err);
                else setImmediate(() => {
                    exports.fibonacciAsync(n-2, (err, val2) => {
                        if (err) done(err);
                        else done(undefined, val1+val2);
                    });
                });
            });
        });
    }
};
```

This converts the `fibonacci` function from a synchronous function into a traditional callback-oriented asynchronous function. We're using `setImmediate` at each stage of the calculation to ensure that the event loop executes regularly and that the server can easily handle other requests while churning away on a calculation. It does nothing to reduce the computation required; this is still the inefficient Fibonacci algorithm. All we've done is spread the computation through the event loop.

In `fibotimes.js`, we can use the following:

```
const math = require('./math');

(async () => {
    for (var num = 1; num < 8000; num++) {
        await new Promise((resolve, reject) => {
            math.fibonacciAsync(num, (err, fibo) => {
                if (err) reject(err);
                else {
                    let now = new Date().toISOString();

```

```
        console.log(`\$now} Fibonacci for \$num} =  
        \$fibo});  
        resolve();  
    }  
})  
})  
})  
})().catch(err => { console.error(err); });
```

We're back to an inefficient algorithm, but one where calculations are distributed through the event loop. Running this version of `fibotimes.js` demonstrates its inefficiency. To demonstrate it in the server, we need to make a few changes.

Because it's an asynchronous function, we will need to change our router code. Create a new file, named `routes/fibonacci-async1.js`, containing the following code:

```
const express = require('express');  
const router = express.Router();  
  
const math = require('../math');  
  
router.get('/', function(req, res, next) {  
    if (req.query.fibonum) {  
        // Calculate using async-aware function, in this server  
        math.fibonacciAsync(req.query.fibonum, (err, fiboval) => {  
            if (err) next(err);  
            else {  
                res.render('fibonacci', {  
                    title: "Calculate Fibonacci numbers",  
                    fibonum: req.query.fibonum,  
                    fiboval: fiboval  
                });  
            }  
        });  
    } else {  
        res.render('fibonacci', {  
            title: "Calculate Fibonacci numbers",  
            fiboval: undefined  
        });  
    }  
});  
  
module.exports = router;
```

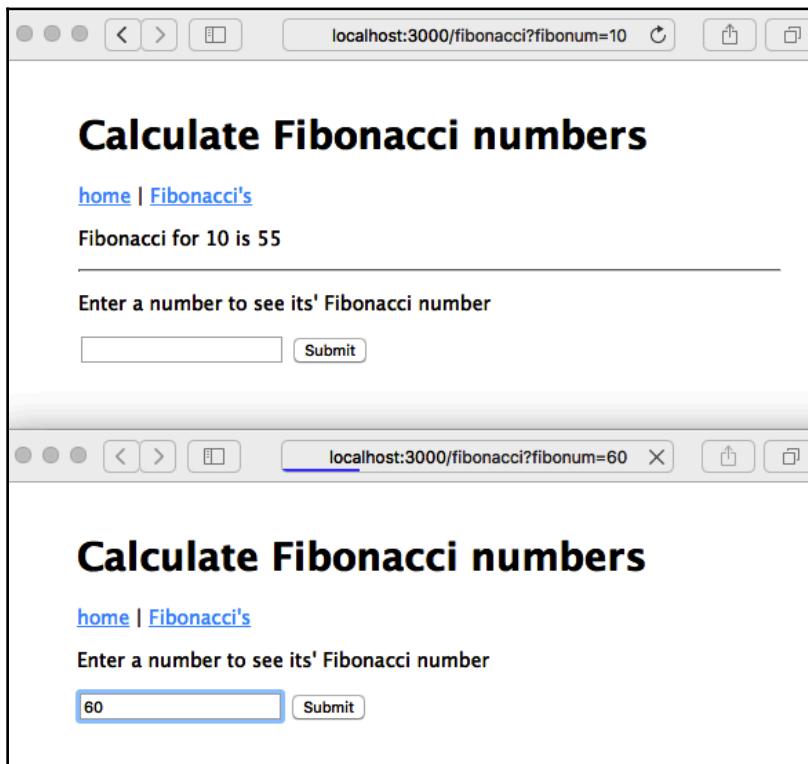
This is the same code as earlier, just rewritten for an asynchronous Fibonacci calculation. The Fibonacci number is returned via a callback function, and even though we have the beginnings of a callback pyramid, it is still manageable.

In `app.js`, make the following change to the application wiring:

```
// const fibonacci = require('./routes/fibonacci');
const fibonacci = require('./routes/fibonacci-async');
```

With this change, the server no longer freezes when calculating a large Fibonacci number. The calculation, of course, still takes a long time, but at least other users of the application aren't blocked.

You can verify this by again opening two browser windows in the application. Enter 60 in one window, and in the other, start requesting smaller Fibonacci numbers. Unlike with the original `fibonacci` function, using `fibonacciAsync` allows both windows to give answers, although if you really did enter 60 in the first window, you might as well take that three-month vacation to Tibet:



It's up to you and your specific algorithms to choose how best to optimize your code and handle any long-running computations you may have.

We've created a simple Express application and demonstrated that there is a flaw that affects performance. We've also discussed algorithmic refactoring, which just leaves us to discuss how to implement a backend service. But first, we need to learn how to create and access a REST service.

Making HTTPClient requests

Another way to mitigate computationally intensive code is to push the calculation to a backend process. To explore that strategy, we'll request computations from a backend Fibonacci server, using the `HTTPClient` object to do so. However, before we look at that, let's first talk in general about using the `HTTPClient` object.

Node.js includes an `HTTPClient` object, which is useful for making HTTP requests. It has the capability to issue any kind of HTTP request. In this section, we'll use the `HTTPClient` object to make HTTP requests similar to calling a REST web service.

Let's start with some code inspired by the `wget` or `curl` commands to make HTTP requests and show the results. Create a file named `wget.js`, containing the following code:

```
const http = require('http');
const url = require('url');
const util = require('util');

const argUrl = process.argv[2];
const parsedUrl = url.parse(argUrl, true);

// The options object is passed to http.request
// telling it the URL to retrieve
const options = {
  host: parsedUrl.hostname,
  port: parsedUrl.port,
  path: parsedUrl.pathname,
  method: 'GET'
};

if (parsedUrl.search) options.path += `?${parsedUrl.search}`;

const req = http.request(options);
// Invoked when the request is finished
req.on('response', res => {
  console.log(`STATUS: ${res.statusCode}`);
  console.log(`HEADERS: ${util.inspect(res.headers)}`);
  res.setEncoding('utf8');
```

```
res.on('data', chunk => { console.log(`BODY: ${chunk}`); });
res.on('error', err => { console.log(`RESPONSE ERROR: ${err}`); });
});
// Invoked on errors
req.on('error', err => { console.log(`REQUEST ERROR: ${err}`); });
req.end();
```

We invoke an HTTP request by using `http.request`, passing in an `options` object describing the request. In this case, we're making a GET request to the server described in a URL we provide on the command line. When the response arrives, the `response` event is fired and we can print out the response. Likewise, an `error` event is fired on errors, and we can print out the error.

This corresponds to the HTTP protocol, where the client sends a request and receives a response.

You can run the script as follows:

```
$ node wget.js http://example.com
STATUS: 200
HEADERS: {
  'accept-ranges': 'bytes',
  'cache-control': 'max-age=604800',
  'content-type': 'text/html; charset=UTF-8',
  date: 'Mon, 06 Jan 2020 02:29:51 GMT',
  etag: '"3147526947"',
  expires: 'Mon, 13 Jan 2020 02:29:51 GMT',
  'last-modified': 'Thu, 17 Oct 2019 07:18:26 GMT',
  server: 'ECS (sjc/4E73)',
  vary: 'Accept-Encoding',
  'x-cache': 'HIT',
  'content-length': '1256',
  connection: 'close'
}
BODY: <!doctype html>
<html>
...
```

Yes, `example.com` is a real website—visit it someday. There's more in the printout, namely the HTML of the page at `http://example.com/`. What we've done is demonstrated how to invoke an HTTP request using the `http.request` function.

The `options` object is fairly straightforward, with the `host`, `port`, and `path` fields specifying the URL that is requested. The `method` field must be one of the HTTP verbs (GET, PUT, POST, and so on). You can also provide a `headers` array for the headers in the HTTP request. For example, you might need to provide a cookie:

```
var options = {
  headers: { 'Cookie': '.. cookie value' }
};
```

The `response` object is itself an `EventEmitter` object that emits the `data` and `error` events. The `data` event is called as data arrives and the `error` event is, of course, called on errors.

The `request` object is a `WritableStream` object, which is useful for HTTP requests containing data, such as `PUT` or `POST`. This means the `request` object has a `write` function, which writes data to the requester. The data format in an HTTP request is specified by the standard MIME type, which was originally created to give us a better email service. Around 1992, the **World Wide Web (WWW)** community worked with the MIME standard committee, who were developing a format for multi-part, multi-media-rich electronic mail. Receiving fancy-looking email is so commonplace today that you might not be aware that email used to come in plaintext. MIME types were developed to describe the format of each piece of data, and the WWW community adopted this for use on the web. HTML forms will post with a content type of `multipart/form-data`, for example.

The next step in offloading some computation to a backend service is to implement the REST service and to make HTTP client requests to that service.

Calling a REST backend service from an Express application

Now that we've seen how to make HTTP client requests, we can look at how to make a REST query within an Express web application. What that effectively means is making an HTTP `GET` request to a backend server, which responds to the Fibonacci number represented by the URL. To do so, we'll refactor the Fibonacci application to make a Fibonacci server that is called from the application. While this is overkill for calculating Fibonacci numbers, it lets us see the basics of implementing a multi-tier application stack in Express.

Inherently, calling a REST service is an asynchronous operation. That means calling the REST service will involve a function call to initiate the request and a callback function to receive the response. REST services are accessed over HTTP, so we'll use the `HTTPClient` object to do so. We'll start this little experiment by writing a REST server and exercising it by making calls to the service. Then, we'll refactor the Fibonacci service to call that server.

Implementing a simple REST server with Express

While Express can also be used to implement a simple REST service, the parameterized URLs we showed earlier (`/user/profile/:id`) can act like parameters to a REST call. Express makes it easy to return data encoded in JSON format.

Now, create a file named `fiboserver.js`, containing the following code:

```
const math = require('./math');
const express = require('express');
const logger = require('morgan');
const app = express();
app.use(logger('dev'));
app.get('/fibonacci/:n', (req, res, next) => {
    math.fibonacciAsync(Math.floor(req.params.n), (err, val) => {
        if (err) next(`FIBO SERVER ERROR ${err}`);
        else {
            res.send({
                n: req.params.n,
                result: val
            });
        }
    });
});
app.listen(process.env.SERVERPORT);
```

This is a stripped-down Express application that gets right to the point of providing a Fibonacci calculation service. The one route it supports handles the Fibonacci computation using the same functions that we've already worked with.

This is the first time we've seen `res.send` used. It's a flexible way to send responses that can take an array of header values (for the HTTP response header) and an HTTP status code. As used here, it automatically detects the object, formats it as JSON text, and sends it with the correct `Content-Type` parameter.

In package.json, add the following to the scripts section:

```
"server": "cross-env SERVERPORT=3002 node ./fiboserver"
```

This automates launching our Fibonacci service.



Note that we're specifying the TCP/IP port via an environment variable and using that variable in the application. Some suggest that putting configuration data in the environment variable is the best practice.

Now, let's run it:

```
$ npm run server
> fibonacci@0.0.0 server /Users/David/chap04/fibonacci
> cross-env SERVERPORT=3002 node ./fiboserver
```

Then, in a separate command window, we can use the curl program to make some requests against this service:

```
$ curl -f http://localhost:3002/fibonacci/10
{"n": "10", "result": 55}
$ curl -f http://localhost:3002/fibonacci/11
{"n": "11", "result": 89}
$ curl -f http://localhost:3002/fibonacci/12
{"n": "12", "result": 144}
```

Over in the window where the service is running, we'll see a log of GET requests and how long each request took to process:

```
$ npm run server
> fibonacci@0.0.0 server /Users/David/chap04/fibonacci
> cross-env SERVERPORT=3002 node ./fiboserver

GET /fibonacci/10 200 0.393 ms - 22
GET /fibonacci/11 200 0.647 ms - 22
GET /fibonacci/12 200 0.772 ms - 23
```

That's easy—using curl, we can make HTTP GET requests. Now, let's create a simple client program, fiboclient.js, to programmatically call the Fibonacci service:

```
const http = require('http');
[
  "/fibonacci/30", "/fibonacci/20", "/fibonacci/10",
  "/fibonacci/9", "/fibonacci/8", "/fibonacci/7",
  "/fibonacci/6", "/fibonacci/5", "/fibonacci/4",
```

```
    "/fibonacci/3", "/fibonacci/2", "/fibonacci/1"
].forEach((path) => {
  console.log(`"${new Date().toISOString()}" requesting ${path}`);
  var req = http.request({
    host: "localhost",
    port: process.env.SERVERPORT,
    path,
    method: 'GET'
  }, res => {
    res.on('data', (chunk) => {
      console.log(`"${new Date().toISOString()}" BODY: ${chunk}`);
    });
  });
  req.end();
});
```

This is our good friend `http.request` with a suitable options object. We're executing it in a loop, so pay attention to the order that the requests are made versus the order the responses arrive.

Then, in `package.json`, add the following to the `scripts` section:

```
"scripts": {
  "start": "node ./bin/www",
  "server": "cross-env SERVERPORT=3002 node ./fiboserver",
  "client": "cross-env SERVERPORT=3002 node ./fiboclient"
}
```

Then, run the client app:

```
$ npm run client

> fibonacci@0.0.0 client /Volumes/Extra/nodejs/Node.js-14-Web-
Development/Chapter04/fibonacci
> cross-env SERVERPORT=3002 node ./fiboclient

2020-01-06T03:18:19.048Z requesting /fibonacci/30
2020-01-06T03:18:19.076Z requesting /fibonacci/20
2020-01-06T03:18:19.077Z requesting /fibonacci/10
2020-01-06T03:18:19.077Z requesting /fibonacci/9
2020-01-06T03:18:19.078Z requesting /fibonacci/8
2020-01-06T03:18:19.079Z requesting /fibonacci/7
2020-01-06T03:18:19.079Z requesting /fibonacci/6
2020-01-06T03:18:19.079Z requesting /fibonacci/5
2020-01-06T03:18:19.080Z requesting /fibonacci/4
2020-01-06T03:18:19.080Z requesting /fibonacci/3
2020-01-06T03:18:19.080Z requesting /fibonacci/2
2020-01-06T03:18:19.081Z requesting /fibonacci/1
```

```
2020-01-06T03:18:19.150Z BODY: {"n": "10", "result": 55}
2020-01-06T03:18:19.168Z BODY: {"n": "4", "result": 3}
2020-01-06T03:18:19.170Z BODY: {"n": "5", "result": 5}
2020-01-06T03:18:19.179Z BODY: {"n": "3", "result": 2}
2020-01-06T03:18:19.182Z BODY: {"n": "6", "result": 8}
2020-01-06T03:18:19.185Z BODY: {"n": "1", "result": 1}
2020-01-06T03:18:19.191Z BODY: {"n": "2", "result": 1}
2020-01-06T03:18:19.205Z BODY: {"n": "7", "result": 13}
2020-01-06T03:18:19.216Z BODY: {"n": "8", "result": 21}
2020-01-06T03:18:19.232Z BODY: {"n": "9", "result": 34}
2020-01-06T03:18:19.345Z BODY: {"n": "20", "result": 6765}
2020-01-06T03:18:24.682Z BODY: {"n": "30", "result": 832040}
```

We're building our way toward adding the REST service to the web application. At this point, we've proved several things, one of which is the ability to call a REST service in our program.

We also inadvertently demonstrated an issue with long-running calculations. You'll notice that the requests were made from the largest to the smallest, but the results appeared in a very different order. Why? This is because of the processing time required for each request, and the inefficient algorithm we're using. The computation time increases enough to ensure that larger request values have enough processing time to reverse the order.

What happens is that `fiboclient.js` sends all of its requests right away, and then each one waits for the response to arrive. Because the server is using `fibonacciAsync`, it will work on calculating all the responses simultaneously. The values that are quickest to calculate are the ones that will be ready first. As the responses arrive in the client, the matching response handler fires, and in this case, the result prints to the console. The results will arrive when they're ready, and not a millisecond sooner.

We now have enough on our hands to offload Fibonacci calculation to a backend service.

Refactoring the Fibonacci application to call the REST service

Now that we've implemented a REST-based server, we can return to the Fibonacci application, applying what we've learned to improve it. We will lift some of the code from `fiboclient.js` and transplant it into the application to do this. Create a new file, `routes/fibonacci-rest.js`, with the following code:

```
const express = require('express');
const router = express.Router();
const http = require('http');
const math = require('../math');

router.get('/', function (req, res, next) {
  if (req.query.fibonum) {
    var httpreq = http.request({
      host: "localhost",
      port: process.env.SERVERPORT,
      path: `/fibonacci/${Math.floor(req.query.fibonum)}`,
      method: 'GET'
    });
    httpreq.on('response', (response) => {
      response.on('data', (chunk) => {
        var data = JSON.parse(chunk);
        res.render('fibonacci', {
          title: "Calculate Fibonacci numbers",
          fibonum: req.query.fibonum,
          fiboval: data.result
        });
      });
      response.on('error', (err) => { next(err); });
    });
    httpreq.on('error', (err) => { next(err); });
    httpreq.end();
  } else {
    res.render('fibonacci', {
      title: "Calculate Fibonacci numbers",
      fiboval: undefined
    });
  }
});

module.exports = router;
```

This is a new variant of the Fibonacci route handler, this time calling the REST backend service. We've transplanted the `http.request` call from `fiboclient.js` and integrated the events coming from the `client` object with the Express route handler. In the normal path of execution, the `HTTPClient` issues a `response` event, containing a `response` object. When that object issues a `data` event, we have our result. The result is JSON text, which we can parse and then return to the browser as the response to its request.

In `app.js`, make the following change:

```
const index = require('./routes/index');
// const fibonacci = require('./routes/fibonacci');
// const fibonacci = require('./routes/fibonacci-async1');
// const fibonacci = require('./routes/fibonacci-await');
const fibonacci = require('./routes/fibonacci-rest');
```

This, of course, reconfigures it to use the new route handler. Then, in `package.json`, change the `scripts` entry to the following:

```
"scripts": {
  "start": "cross-env DEBUG=fibonacci:* node ./bin/www",
  "startrest": "cross-env DEBUG=fibonacci:* SERVERPORT=3002 node
./fiboserver",
  "server": "cross-env DEBUG=fibonacci:* SERVERPORT=3002 node
./bin/www",
  "client": "cross-env DEBUG=fibonacci:* SERVERPORT=3002 node
./fiboclient"
},
```

How can we have the same value for `SERVERPORT` for all three `scripts` entries? The answer is that the variable is used differently in different places. In `startrest`, this variable is used in `routes/fibonacci-rest.js` to know at which port the REST service is running. Likewise, in `client`, `fiboclient.js` uses this variable for the same purpose. Finally, in `server`, the `fiboserver.js` script uses the `SERVERPORT` variable to know which port to listen on.

In `start` and `startrest`, no value is given for `PORT`. In both cases, `bin/www` defaults to `PORT=3000` if a value is not specified.

In a command window, start the backend server, and in another, start the application. Open a browser window, as before, and make a few requests. You should see an output similar to the following:

```
$ npm run startrest  
  
> fibonacci@0.0.0 startrest /Users/David/chap04/fibonacci  
> cross-env DEBUG=fibonacci:* SERVERPORT=3002 node ./fiboserver  
  
GET /fibonacci/34 200 21124.036 ms - 27  
GET /fibonacci/12 200 1.578 ms - 23  
GET /fibonacci/16 200 6.600 ms - 23  
GET /fibonacci/20 200 33.980 ms - 24  
GET /fibonacci/28 200 1257.514 ms - 26
```

The output looks like this for the application:

```
$ npm run server  
  
> fibonacci@0.0.0 server /Users/David/chap04/fibonacci  
> cross-env DEBUG=fibonacci:* SERVERPORT=3002 node ./bin/www  
  
fibonacci:server Listening on port 3000 +0ms  
GET /fibonacci?fibonum=34 200 21317.792 ms - 548  
GET /stylesheets/style.css 304 20.952 ms --  
GET /fibonacci?fibonum=12 304 109.516 ms --  
GET /stylesheets/style.css 304 0.465 ms --  
GET /fibonacci?fibonum=16 200 83.067 ms - 544  
GET /stylesheets/style.css 304 0.900 ms --  
GET /fibonacci?fibonum=20 200 221.842 ms - 545  
GET /stylesheets/style.css 304 0.778 ms --  
GET /fibonacci?fibonum=28 200 1428.292 ms - 547  
GET /stylesheets/style.css 304 19.083 ms --
```

Because we haven't changed the templates, the screen will look exactly as it did earlier.

We may run into another problem with this solution. The asynchronous implementation of our inefficient Fibonacci algorithm may cause the Fibonacci service process to run out of memory. In the Node.js FAQs, <https://github.com/nodejs/node/wiki/FAQ>, it's suggested to use the `--max_old_space_size` flag. You'd add this to `package.json`, as follows:

```
"server": "cross-env SERVERPORT=3002 node ./fiboserver --  
max_old_space_size 5000",
```

However, the FAQs also say that if you're running into maximum memory space problems, your application should probably be refactored. This goes back to the point we made earlier that there are several approaches to addressing performance problems, one of which is the algorithmic refactoring of your application.

Why go through the trouble of developing this REST server when we could just directly use `fibonacciAsync`?

The main advantage is the ability to push the CPU load for this heavyweight calculation to a separate server. Doing so preserves the CPU capacity on the frontend server so that it can attend to the web browsers. GPU coprocessors are now widely used for numerical computing and can be accessed via a simple network API. The heavy computation can be kept separate, and you can even deploy a cluster of backend servers sitting behind a load balancer, evenly distributing requests. Decisions such as this are made all the time to create multi-tier systems.

What we've demonstrated is that it's possible to implement simple multi-tier REST services in a few lines of Node.js and Express. This whole exercise gave us a chance to think about computationally intensive code in Node.js and the value of splitting a larger service into multiple services.

Of course, Express isn't the only framework that can help us create REST services.

Some RESTful modules and frameworks

Here are a few available packages and frameworks to assist your REST-based projects:

- **Restify** (<http://restify.com/>): This offers both client-side and server-side frameworks for both ends of REST transactions. The server-side API is similar to Express.
- **Loopback** (<http://loopback.io/>): This is an offering from StrongLoop. It offers a lot of features and is, of course, built on top of Express.

In this section, we've done a big thing in creating a backend REST service.

Summary

You learned a lot in this chapter about Node.js's `EventEmitter` pattern, `HTTPClient`, and server objects, at least two ways to create an HTTP service, how to implement web applications, and even how to create a REST client and REST service integrated into a customer-facing web application. Along the way, we again explored the risks of blocking operations, the importance of keeping the event loop running, and a couple of ways to distribute work across multiple services.

Now, we can move on to implementing a more complete application: one for taking notes. We will use the Notes application in several upcoming chapters as a vehicle to explore the Express application framework, database access, deployment to cloud services or on your own server, user authentication, semi-real-time communication between users, and even hardening an application against several kinds of attacks. We'll end up with an application that can be deployed to cloud infrastructure.

There's still a lot to cover in this book, and it starts in the next chapter with creating a basic Express application.

2

Section 2: Developing the Express Application

The core of this book is developing an Express application from the initial concept of storing data in a database and supporting multiple users.

This section comprises the following chapters:

- Chapter 5, *Your First Express Application*
- Chapter 6, *Implementing the Mobile-First Paradigm*
- Chapter 7, *Data Storage and Retrieval*
- Chapter 8, *Authenticating Users with a Microservice*
- Chapter 9, *Dynamic Client/Server Interaction with Socket.IO*

5

Your First Express Application

Now that we've got our feet wet building an Express application for Node.js, let's start developing an application that performs a useful function. The application we'll build will keep a list of notes and will eventually have users who can send messages to each other. Over the course of this book, we will use it to explore some aspects of real Express web applications.

In this chapter, we'll start with the basic structure of an application, the initial UI, and the data model. We'll also lay the groundwork for adding persistent data storage and all the other features that we will cover in later chapters.

The topics covered in this chapter include the following:

- Using Promises and async functions in Express router functions
- JavaScript class definitions and data hiding in JavaScript classes
- The architecture of an Express application using the MVC paradigm
- Building an Express application
- Implementing the CRUD paradigm
- Express application theming and Handlebars templates

To get started, we will talk about integrating Express router callbacks with async functions.

Exploring Promises and `async` functions in Express router functions

Before we get into developing our application, we need to take a deeper look at using the `Promise` class and `async` functions with Express because Express was invented before these features existed, and so it does not directly integrate with them. While we should be using `async` functions wherever possible, we have to be aware of how to properly use them in certain circumstances, such as in an Express application.

The rules in Express for handling asynchronous execution are as follows:

- Synchronous errors are caught by Express and cause the application to go to the error handler.
- Asynchronous errors must be reported by calling `next (err)`.
- A successfully executing middleware function tells Express to invoke the next middleware by calling `next ()`.
- A router function that returns a result to the HTTP request does not call `next ()`.

In this section, we'll discuss three ways to use Promises and `async` functions in a way that is compatible with these rules.

Both Promises and `async` functions are used for deferred and asynchronous computation and can make intensely nested callback functions a thing of the past:

- A `Promise` class represents an operation that hasn't completed yet but is expected to be completed in the future. We've used Promises already, so we know that the `.then` or `.catch` functions are invoked asynchronously when the promised result (or error) is available.
- Inside an `async` function, the `await` keyword is available to automatically wait for a `Promise` to resolve. It returns the result of a `Promise`, or else throws errors, in the natural location at the next line of code, while also accommodating asynchronous execution.

The magic of `async` functions is that we can write asynchronous code that looks like synchronous code. It's still asynchronous code—meaning it works correctly with the Node.js event loop—but instead of results and errors landing inside callback functions, errors are thrown naturally as exceptions and results naturally land on the next line of code.

Because this is a new feature in JavaScript, there are several traditional asynchronous coding practices with which we must correctly integrate. You may come across some other libraries for managing asynchronous code, including the following:

- The `async` library is a collection of functions for various asynchronous patterns. It was originally completely implemented around the callback function paradigm, but the current version can handle `async` functions and is available as an ES6 package. Refer to <https://www.npmjs.com/package/async> for more information.
- Before Promises were standardized, at least two implementations were available: Bluebird (<http://bluebirdjs.com/>) and Q (<https://www.npmjs.com/package/q>). Nowadays, we focus on using the standard, built-in `Promise` object, but both of these packages offer additional features. What's more likely is that we will come across older code that uses these libraries.

These and other tools were developed to make it easier to write asynchronous code and to solve the **pyramid of doom** problem. This is named after the shape that the code takes after a few layers of nesting. Any multistage process written as callbacks can quickly escalate to code that is nested many levels deep. Consider the following example:

```
router.get('/path/to/something', (req, res, next) => {
  doSomething(req.query.arg1, req.query.arg2, (err, data1) => {
    if (err) return next(err);
    doAnotherThing(req.query.arg3, req.query.arg2, data1, (err2,
      data2) => {
      if (err2) return next(err2);
      somethingCompletelyDifferent(req.query.arg1, req.query.arg42,
        (err3, data3) => {
          if (err3) return next(err3);
          doSomethingElse((err4, data4) => {
            if (err4) return next(err4);
            res.render('page', { data1, data2, data3, data4 });
          });
        });
      });
    });
});
```

We don't need to worry about the specific functions, but we should instead recognize that one callback tends to lead to another. Before you know it, you've landed in the middle of a deeply nested structure like this. Rewriting this as an `async` function will make it much clearer. To get there, we need to examine how Promises are used to manage asynchronous results, as well as get a deeper understanding of `async` functions.

A Promise is either in an unresolved or resolved state. This means that we create a Promise using `new Promise`, and initially, it is in the unresolved state. The Promise object transitions to the resolved state, where either its `resolve` or `reject` functions are called. If the `resolve` function is called, the Promise is in a successful state, and if instead its `reject` function is called, the Promise is in a failed state.

More precisely, Promise objects can be in one of three states:

- **Pending:** This is the initial state, which is neither fulfilled nor rejected.
- **Fulfilled:** This is the final state, where it executes successfully and produces a result.
- **Rejected:** This is the final state, where execution fails.

We generate a Promise in the following way:

```
function asyncFunction(arg1, arg2) {
  return new Promise((resolve, reject) => {
    // perform some task or computation that's asynchronous
    // for any error detected:
    if (errorDetected) return reject(dataAboutError);
    // When the task is finished
    resolve(theResult);
  });
}
```

A function like this creates the Promise object, giving it a callback function, within which is your asynchronous operation. The `resolve` and `reject` functions are passed into that function and are called when the Promise is resolved as either a success or failure state. A typical use of `new Promise` is a structure like this:

```
function readFile(filename) {
  return new Promise((resolve, reject) => {
    fs.readFile(filename, (err, data) => {
      if (err) reject(err);
      else resolve(data);
    });
  });
}
```

This is the pattern that we use when *promisifying* an asynchronous function that uses callbacks. The asynchronous code executes, and in the callback, we invoke either `resolve` or `reject`, as appropriate. We can usually use the `util.promisify` Node.js function to do this for us, but it's very useful to know how to construct this as needed.

Your caller then uses the function, as follows:

```
asyncFunction(arg1, arg2)
  .then((result) => {
    // the operation succeeded
    // do something with the result
    return newResult;
  })
  .catch(err => {
    // an error occurred
  });

```

The `Promise` object is fluid enough that the function passed in a `.then` handler can return something, such as another `Promise`, and you can chain the `.then` calls together. The value returned in a `.then` handler (if any) becomes a new `Promise` object, and in this way, you can construct a chain of `.then` and `.catch` calls to manage a sequence of asynchronous operations.

With the `Promise` object, a sequence of asynchronous operations is called a **Promise chain**, consisting of chained `.then` handlers, as we will see in the next section.

Promises and error handling in Express router functions

It is important that all errors are correctly handled and reported to Express. With synchronous code, Express will correctly catch a thrown exception and send it to the error handler. Take the following example:

```
app.get('/', function (req, res) {
  throw new Error('BROKEN');
});
```

Express catches that exception and does the right thing, meaning it invokes the error handler, but it does not see a thrown exception in asynchronous code. Consider the following error example:

```
app.get('/', (req, res) => {
  fs.readFile('/does-not-exist', (err, data) => {
    if (err) throw new Error(err);
    // do something with data, like
    res.send(data);
  });
});
```

This is an example of the error indicator landing in an inconvenient place in the callback function. The exception is thrown in a completely different stack frame than the one invoked by Express. Even if we arranged to return a Promise, as is the case with an `async` function, Express doesn't handle the Promise. In this example, the error is lost; the caller would never receive a response and nobody would know why.

It is important to reliably catch any errors and respond to the caller with results or errors. To understand this better, let's rewrite the pyramid of doom example:

```
router.get('/path/to/something', (req, res, next) => {
  let data1, data2, data3, data4;
  doSomething(req.query.arg1, req.query.arg2)
    .then(_data1 => {
      data1 = _data1;
      return doAnotherThing(req.query.arg3, req.query.arg2, data1);
    })
    .then(_data2 => {
      data2 = _data2;
      return somethingCompletelyDifferent(req.query.arg1,
        req.query.arg42);
    })
    .then(_data3 => {
      data3 = _data3;
      return doSomethingElse();
    })
    .then(_data4 => {
      data4 = _data4;
      res.render('page', { data1, data2, data3, data4 });
    })
    .catch(err => { next(err); });
});
```

This is rewritten using a Promise chain, rather than nested callbacks. What had been a deeply nested pyramid of callback functions is now arguably a little cleaner thanks to Promises.

The `Promise` class automatically captures all the errors and searches down the chain of operations attached to the `Promise` to find and invoke the first `.catch` function. So long as no errors occur, each `.then` function in the chain is executed in turn.

One advantage of this is that error reporting and handling is much easier. With the callback paradigm, the nature of the callback pyramid makes error reporting trickier, and it's easy to miss adding the correct error handling to every possible branch of the pyramid. Another advantage is that the structure is flatter and, therefore, easier to read.

To integrate this style with Express, notice the following:

- The final step in the `Promise` chain uses `res.render` or a similar function to return a response to the caller.
- The final `catch` function reports any errors to Express using `next(err)`.

If instead we simply returned the `Promise` and it was in the `rejected` state, Express would not handle that failed rejection and the error would be lost.

Having looked at integrating asynchronous callbacks and `Promise` chains with Express, let's look at integrating `async` functions.

Integrating `async` functions with Express router functions

There are two problems that need to be addressed that are related to asynchronous coding in JavaScript. The first is the pyramid of doom, an unwieldily nested callback structure. The second is the inconvenience of where results and errors are delivered in an asynchronous callback.

To explain, let's reiterate the example that Ryan Dahl gives as the primary Node.js idiom:

```
db.query('SELECT ..etc..', function(err, resultSet) {  
    if (err) {  
        // Instead, errors arrive here  
    } else {  
        // Instead, results arrive here  
    }  
});  
// We WANT the errors or results to arrive here
```

The goal here is to avoid blocking the event loop with a long operation. Deferring the processing of results or errors using callback functions is an excellent solution and is the founding idiom of Node.js. The implementation of callback functions led to this pyramid-shaped problem. Promises help flatten the code so that it is no longer in a pyramid shape. They also capture errors, ensuring delivery to a useful location. In both cases, errors and results are buried inside an anonymous function and are not delivered to the next line of code.

Generators and the iteration protocol are an intermediary architectural step that, when combined with Promises, lead to the `async` function. We won't use either of these in this book, but they are worth learning about.



For the documentation for the iteration protocol, refer to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols.

For the documentation for the generator functions, refer to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator.

We've already used `async` functions and learned about how they let us write clean-looking asynchronous code. For example, the `db.query` example as an `async` function looks as follows:

```
async function dbQuery(params) {
  const resultSet = await db.query('SELECT ..etc..');
  // results and errors land here
  return resultSet;
}
```

This is much cleaner, with results and errors landing where we want them to.

However, to discuss integration with Express, let's return to the pyramid of doom example from earlier, rewriting it as an `async` function:

```
router.get('/path/to/something', async (req, res, next) => {
  try {
    const data1 = await doSomething(req.query.arg1, req.query.arg2);
    const data2 = await doAnotherThing(req.query.arg3,
                                         req.query.arg2, data1);
    const data3 = await somethingCompletelyDifferent(req.query.arg1,
                                                    req.query.arg4);
    const data4 = await doSomethingElse();
    res.render('page', { data1, data2, data3, data4 });
  } catch (err) {
    next(err);
  }
});
```

```
    } catch(err) {
      next(err);
    }
});
```

Other than `try/catch`, this example is very clean compared to its earlier forms, both as a callback pyramid and as a Promise chain. All the boilerplate code is erased, and the intent of the programmer shines through clearly. Nothing is lost inside a callback function. Instead, everything lands on the next line of code where it is convenient.

The `await` keyword looks for a Promise. Therefore, `doSomething` and the other functions are expected to return a Promise, and `await` manages its resolution. Each of these functions could be an `async` function, and thereby automatically returns a Promise, or it could explicitly create a Promise to manage an asynchronous function call. A generator function is also involved, but we don't need to know how that works. We just need to know that `await` manages the asynchronous execution and the resolution of the Promise.

More importantly, each statement with an `await` keyword executes asynchronously. That's a side effect of `await`—managing asynchronous execution to ensure the asynchronous result or error is delivered correctly. However, Express cannot catch an asynchronous error and requires us to notify it of asynchronous results using `next()`.

The `try/catch` structure is needed for integration with Express. For the reasons just given, we must explicitly catch asynchronously delivered errors and notify Express with `next(err)`.

In this section, we discussed three methods for notifying Express about asynchronously delivered errors. The next thing to discuss is some architectural choices to structure the code.

Architecting an Express application in the MVC paradigm

Express doesn't enforce an opinion on how you should structure the **Model, View, and Controller (MVC)** modules of your application, or whether you should follow any kind of MVC paradigm at all. The MVC pattern is widely used and involves three main architectural pieces. The **controller** accepts inputs or requests from the user, converting that into commands sent to the model. The **model** contains the data, logic, and rules by which the application operates. The **view** is used to present results to the user.

As we learned in the previous chapter, the blank application created by the Express generator provides two aspects of the MVC model:

- The `views` directory contains template files, controlling the display portion, corresponding to the view.
- The `routes` directory contains code implementing the URLs recognized by the application and coordinates the data manipulation required to generate the response to each URL. This corresponds to the controller.

Since the router functions also call the function to generate the result using a template, we cannot strictly say that the router functions are the controller and that the `views` templates are the view. However, it's close enough to the MVC model for it to be a useful analogy.

This leaves us with a question of where to put the model code. Since the same data manipulation can be used by multiple router functions, clearly the router functions should use a standalone module (or modules) containing the model code. This will also ensure a clean separation of concerns—for example, to ease the unit testing of each.

The approach we'll use is to create a `models` directory as a sibling of the `views` and `routes` directories. The `models` directory will hold modules to handle data storage and other code that we might call **business logic**. The API of the modules in the `models` directory will provide functions to create, read, update, or delete data items—a **Create, Read, Update, and Delete/Destroy (CRUD)** model—and other functions necessary for the view code to do its thing.

The CRUD model includes the four basic operations of persistent data storage. The `Notes` application is structured as a CRUD application to demonstrate the implementation each of these operations.

We'll use functions named `create`, `read`, `update`, and `destroy` to implement each of the basic operations.



We're using the `destroy` verb, rather than `delete`, because `delete` is a reserved word in JavaScript.

With that architectural decision in mind, let's proceed with creating the `Notes` application.

Creating the Notes application

Since we're starting a new application, we can use the Express generator to give us a starting point. It is not absolutely necessary to use this tool since we can definitely write the code ourselves. The advantage, however, is that it gives us a fully fleshed out starting point:

```
$ mkdir notes
$ cd notes
$ npx express-generator@4.x --view=hbs --git .
destination is not empty, continue? [y/N] y

  create : .
  create : ./package.json
  create : ./app.js
  create : ./gitignore
  create : ./public
  create : ./routes
  create : ./routes/index.js
  create : ./routes/users.js
  create : ./views
  create : ./views/index.hbs
  create : ./views/layout.hbs
  create : ./views/error.hbs
  create : ./bin
  create : ./bin/www
  create : ./public/stylesheets
  create : ./public/stylesheets/style.css

install dependencies:
$ cd . && npm install

run the app:
```

```
$ DEBUG=notes:* npm start  
  
  create : ./public/javascripts  
  create : ./public/images  
$ npm install  
added 82 packages and removed 5 packages in 97.188s
```

As in the previous chapter, we will use `cross-env` to ensure that the scripts run cross-platform. Start by changing `package.json` to have the following `scripts` section:

```
"scripts": {  
  "start": "cross-env DEBUG=notes:* node ./app.mjs"  
}
```

The supplied script uses `bin/www`, but shortly, we'll restructure the generated code to put everything into a single ES6 script named `app.mjs`.

Then, install `cross-env`, as follows:

```
$ npm install cross-env --save
```

With `cross-env`, the scripts are executable on either Unix-like systems or Windows.

If you wish, you can run `npm start` and view the blank application in your browser. Instead, let's rewrite this starting-point code using ES6 modules, and also combine the contents of `bin/www` with `app.mjs`.

Rewriting the generated router module as an ES6 module

Let's start with the `routes` directory. Since we won't have a `Users` concept right now, delete `users.js`. We need to convert the JavaScript files into ES6 format, and we can recall that the simplest way for a module to be recognized as an ES6 module is to use the `.mjs` extension. Therefore, rename `index.js` to `index.mjs`, rewriting it as follows:

```
import { default as express } from 'express';  
export const router = express.Router();  
  
router.get('/', async (req, res, next) => {  
  //... placeholder for Notes home page code  
  res.render('index', { title: 'Notes' });  
});
```

We'll finish this up later, but what we've done is restructured the code we were given. We can import the Express package, and then export the `router` object. Adding router functions is, of course, the done in the same way, whether it is a CommonJS or an ES6 module. We made the router callback an `async` function because it will be using `async` code.

We'll need to follow the same pattern for any other router modules we create.

Having converted this to an ES6 module, the next step is to merge code from `bin/www` and `app.js` into an ES6 module named `app.mjs`.

Creating the Notes application wiring – `app.mjs`

Since the `express-generator` tool gives us a slightly messy application structure that does not use ES6 modules, let's reformulate the code it gave us appropriately. The first, `app.mjs`, contains the *wiring* of the application, meaning it configures the objects and functions from which the application is built while not containing any functions of its own. The other code, `appsupport.mjs`, contains the callback functions that appeared in the generated `app.js` and `bin/www` modules.

In `app.mjs`, start with this:

```
import { default as express } from 'express';
import { default as hbs } from 'hbs';
import * as path from 'path';
// import * as favicon from 'serve-favicon';
import { default as logger } from 'morgan';
import { default as cookieParser } from 'cookie-parser';
import { default as bodyParser } from 'body-parser';
import * as http from 'http';
import { approotdir } from './aprootdir.mjs';
const __dirname = approotdir;
import {
  normalizePort, onError, onListening, handle404, basicErrorHandler
} from './appsupport.mjs';

import { router as indexRouter } from './routes/index.mjs';
// import { router as notesRouter } from './routes/notes.mjs';
```

The generated `app.js` code had a series of `require` statements. We have rewritten them to use corresponding `import` statements. We also added code to calculate the `__filename` and `__dirname` variables, but presented a little differently. To support this, add a new module, `aprootdir.mjs`, containing the following:

```
import * as path from 'path';
import * as url from 'url';
const __filename = url.fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
export const approotdir = __dirname;
```

In the `dirname-fixed.mjs` example in Chapter 3, *Exploring Node.js Modules*, we imported specific functions from the `path` and `url` core modules. We have used that code and then exported the value for `__dirname` as `aprootdir`. Other parts of the `Notes` application simply need the pathname of the root directory of the application in order to calculate the required pathnames.

Return your attention to `app.mjs` and you'll see that the router modules are imported as `indexRouter` and `notesRouter`. For the moment, `notesRouter` is commented out, but we'll get to that in a later section.

Now, let's initialize the `express` application object:

```
export const app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'hbs');
hbs.registerPartials(path.join(__dirname, 'partials'));

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

// Router function lists
app.use('/', indexRouter);
// app.use('/notes', notesRouter);

// error handlers
// catch 404 and forward to error handler
app.use(handle404);
app.use(basicErrorHandler);
```

```
export const port = normalizePort(process.env.PORT || '3000');
app.set('port', port);
```

This should look familiar to the `app.js` code we used in the previous chapter. Instead of inline functions, however, they're pushed into `appsupport.mjs`.

The `app` and `port` objects are exported in case some other code in the application needs those values.

This section of code creates and configures the Express application instance. To make it a complete running server, we need the following code:

```
export const server = http.createServer(app);

server.listen(port);
server.on('error', onError);
server.on('listening', onListening);
```

This section of code wraps the Express application in an HTTP server and gets it listening to HTTP requests. The `server` object is also exported in case other code wants to access it.

Compare `app.mjs` with the generated `app.js` and `bin/www` code and you will see that we've covered everything in those two modules except for the inline functions. These inline functions could be written at the end of `app.mjs`, but we've elected instead to create a second module to hold them.

Create `appsupport.mjs` to hold the inline functions, starting with the following:

```
import { port } from './app.mjs';

export function normalizePort(val) {
  const port = parseInt(val, 10);
  if (isNaN(port)) {
    return val;
  }
  if (port >= 0) {
    return port;
  }
  return false;
}
```

This function handles safely converting a port number string that we might be given into a numerical value that can be used in the application. The `isNaN` test is used to handle cases where instead of a TCP port number, we want to use a **named pipe**. Look carefully at the other functions and you'll see that they all accommodate either a numerical port number or a string described as a pipe:

```
export function onError(error) {
  if (error.syscall !== 'listen') {
    throw error;
  }
  const bind = typeof port === 'string'
    ? 'Pipe ' + port
    : 'Port ' + port;

  switch (error.code) {
    case 'EACCES':
      console.error(`#${bind} requires elevated privileges`);
      process.exit(1);
      break;
    case 'EADDRINUSE':
      console.error(`#${bind} is already in use`);
      process.exit(1);
      break;
    default:
      throw error;
  }
}
```

The preceding code handles errors from the HTTP server object. Some of these errors will simply cause the server to exit:

```
import { server } from './app.mjs';
export function onListening() {
  const addr = server.address();
  const bind = typeof addr === 'string'
    ? 'pipe ' + addr
    : 'port ' + addr.port;
  console.log(`Listening on ${bind}`);
}
```

The preceding code prints a user-friendly message saying where the server is listening for HTTP connections. Because this function needs to reference the `server` object, we have imported it:

```
export function handle404(req, res, next) {
  const err = new Error('Not Found');
  err.status = 404;
```

```
    next(err);
}

export function basicErrorHandler(err, req, res, next) {
  // Defer to built-in error handler if headersSent
  // See: http://expressjs.com/en/guide/error-handling.html
  if (res.headersSent) {
    return next(err)
  }
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ?
    err : {};
  // render the error page
  res.status(err.status || 500);
  res.render('error');
}
```

These were previously inline functions implementing error handling for the Express application.

The result of these changes is that `app.mjs` is now clean of distracting code, and it instead focuses on connecting together the different parts that make up the application. Since Express is not opinionated, it does not care that we restructured the code like this. We can structure the code in any way that makes sense to us and that correctly calls the Express API.

Since this application is about storing data, let's next talk about the data storage modules.

Implementing the Notes data storage model

Remember that we decided earlier to put data model and data storage code into a directory named `models` to go along with the `views` and `routes` directories.

Together, these three directories will separately store the three sides of the MVC paradigm.

The idea is to centralize the implementation details of storing data. The data storage modules will present an API for storing and manipulating application data, and over the course of this book, we'll make several implementations of this API. To switch between one storage engine to another, we will just require a configuration change. The rest of the application will use the same API methods, regardless of the storage engine being used.

To start, let's define a pair of classes to describe the data model. Create a file named `models/Notes.mjs` with the following code in it:

```
const _note_key = Symbol('key');
const _note_title = Symbol('title');
const _note_body = Symbol('body');

export class Note {
  constructor(key, title, body) {
    this[_note_key] = key;
    this[_note_title] = title;
    this[_note_body] = body;
  }

  get key() { return this[_note_key]; }
  get title() { return this[_note_title]; }
  set title(newTitle) { this[_note_title] = newTitle; }
  get body() { return this[_note_body]; }
  set body(newBody) { this[_note_body] = newBody; }
}

export class AbstractNotesStore {
  async close() { }
  async update(key, title, body) { }
  async create(key, title, body) { }
  async read(key) { }
  async destroy(key) { }
  async keylist() { }
  async count() { }
}
```

This defines two classes—`Note` and `AbstractNotesStore`—whose purpose is as follows:

- The `Note` class describes a single note that our application will manage.
- The `AbstractNotesStore` class describes methods for managing some note instances.

In the `Note` class, `key` is how we look for the specific note, and `title` and `body` are the content of the note. It uses an important data hiding technique, which we'll discuss in a minute.

The `AbstractNotesStore` class documents the methods that we'll use for accessing notes from a data storage system. Since we want the `Notes` application to implement the CRUD paradigm, we have the `create`, `read`, `update`, and `destroy` methods, plus a couple more to assist in searching for notes. What we have here is an empty class that serves to document the API, and we will use this as the base class for several storage modules that we'll implement later.

The `close` method is meant to be used when we're done with a datastore. Some datastores keep an open connection to a server, such as a database server, and the `close` method should be used to close that connection.

This is defined with `async` functions because we'll store data in the filesystem or in databases. In either case, we need an asynchronous API.

Before implementing our first data storage model, let's talk about data hiding in JavaScript classes.

Data hiding in ES-2015 class definitions

In many programming languages, class definitions let us designate some data fields as private and others as public. This is so that programmers can hide implementation details. However, writing code on the Node.js platform is all about JavaScript, and JavaScript, in general, is very lax about everything. So, by default, fields in an instance of a JavaScript class are open to any code to access or modify.

One concern arises if you have several modules all adding fields or functions to the same object. How do you guarantee that one module won't step on fields added by another module? By default, in JavaScript, there is no such guarantee.

Another concern is hiding implementation details so that the class can be changed while knowing that internal changes won't break other code. By default, JavaScript fields are open to all other code, and there's no guarantee other code won't access fields that are meant to be private.

The technique used in the `Note` class gates access to the fields through getter and setter functions. These in turn set or get values stored in the instance of the class. By default, those values are visible to any code, and so these values could be modified in ways that are incompatible with the class. The best practice when designing classes is to localize all manipulation of class instance data to the member functions. However, JavaScript makes the fields visible to the world, making it difficult to follow this best practice. The pattern used in the `Note` class is the closest we can get in JavaScript to data hiding in a class instance.

The technique we use is to name the fields using instances of the `Symbol` class. `Symbol`, another ES-2015 feature, is an opaque object with some interesting attributes that make it attractive for use as keys for private fields in objects. Consider the following code:

```
$ node
Welcome to Node.js v12.13.0.
Type ".help" for more information.
> Symbol('a') === Symbol('a')
false
> let b = Symbol('b')
undefined
> console.log(b)
Symbol(b)
undefined
> let b1 = Symbol('b')
undefined
> console.log(b1)
Symbol(b)
undefined
> b === b1
false
> b === b
true
```

Creating a `Symbol` instance is done with `Symbol('symbol-name')`. The resulting `Symbol` instance is a unique identifier, and even if you call `Symbol('symbol-name')` again, the uniqueness is preserved. Each `Symbol` instance is unique from all other `Symbol` instances, even ones that are formed from the same string. In this example, the `b` and `b1` variables were both formed by calling `Symbol('b')`, but they are not equivalent.

Let's see how we can use a `Symbol` instance to attach fields to an object:

```
> const obj = {};
undefined
> obj[Symbol('b')] = 'b';
'b'
> obj[Symbol('b')] = 'b1';
'b1'
> obj
{ [Symbol(b)]: 'b', [Symbol(b)]: 'b1' }
```

We've created a little object, then used those `Symbol` instances as field keys to store data in the object. Notice that when we dump the object's contents, the two fields both register as `Symbol(b)`, but they are two separate fields.

With the `Note` class, we have used the `Symbol` instances to provide a small measure of data hiding. The actual values of the `Symbol` instances are hidden inside `Notes.mjs`. This means the only code that can directly access the fields is the code running inside `Notes.mjs`:

```
> let note = new Note('key', 'title', 'body')
undefined
> note
Note {
  [Symbol(key)]: 'key',
  [Symbol(title)]: 'title',
  [Symbol(body)]: 'body'
}
> note[Symbol('key')] = 'new key'
'new key'
> note
Note {
  [Symbol(key)]: 'key',
  [Symbol(title)]: 'title',
  [Symbol(body)]: 'body',
  [Symbol(key)]: 'new key'
}
```

With the `Note` class defined, we can create a `Note` instance, and then dump it and see the resulting fields. The keys to these fields are indeed `Symbol` instances. These `Symbol` instances are hidden inside the module. The fields themselves are visible to code outside the module. As we can see here, an attempt to subvert the instance with `note[Symbol('key')] = 'new key'` does not overwrite the field but instead adds a second field.

With our data types defined, let's start implementing the application, beginning with a simple in-memory datastore.

Implementing an in-memory Notes datastore

Eventually, we will create a `Notes` data storage module that persists the notes to long-term storage. But to get us started, let's implement an in-memory datastore so that we can get on with implementing the application. Because we designed an abstract base class, we can easily create new implementations of that class for various storage services.

Create a file named `notes-memory.mjs` in the `models` directory with the following code:

```
import { Note, AbstractNotesStore } from './Notes.mjs';

const notes = [];

export class InMemoryNotesStore extends AbstractNotesStore {

    async close() { }

    async update(key, title, body) {
        notes[key] = new Note(key, title, body);
        return notes[key];
    }

    async create(key, title, body) {
        notes[key] = new Note(key, title, body);
        return notes[key];
    }

    async read(key) {
        if (notes[key]) return notes[key];
        else throw new Error(`Note ${key} does not exist`);
    }

    async destroy(key) {
        if (notes[key]) {
            delete notes[key];
        } else throw new Error(`Note ${key} does not exist`);
    }

    async keylist() {
        return Object.keys(notes);
    }
}
```

```
    }

    async count() {
      return notes.length;
    }
}
```

This should be fairly self-explanatory. The notes are stored in a private array, named `notes`. The operations, in this case, are defined in terms of adding or removing items in that array. The key object for each `Note` instance is used as the index to the `notes` array, which in turn holds the `Note` instance. This is simple, fast, and easy to implement. It does not support any long-term data persistence, and any data stored in this model will disappear when the server is killed.

We need to initialize an instance of `NotesStore` so that it can be used in the application. Let's add the following to `app.mjs`, somewhere near the top:

```
import { InMemoryNotesStore } from './models/notes-memory.mjs';
export const NotesStore = new InMemoryNotesStore();
```

This creates an instance of the class and exports it as `NotesStore`. This will work so long as we have a single `NotesStore` instance, but in [Chapter 7, Data Storage and Retrieval](#), we will change this around to support dynamically selecting a `NotesStore` instance.

We're now ready to start implementing the web pages and associated code for the application, starting with the home page.

The Notes home page

We're going to modify the starter application to support creating, editing, updating, viewing, and deleting notes. Let's start by changing the home page to show a list of notes, and have a top navigation bar linking to an **ADD Note** page so that we can always add a new note.

There's no change required in `app.mjs` because the home page is generated in routes controlled in this router module:

```
import { router as indexRouter } from './routes/index.mjs';
...
app.use('/', indexRouter);
```

In `app.mjs`, we configured the Handlebars template engine to use the `partials` directory to hold partial files. Therefore, make sure you create that directory.

To implement the home page, update `routes/index.mjs` to the following:

```
import * as express from 'express';
import { NotesStore as notes } from '../app.mjs';
export const router = express.Router();

/* GET home page. */
router.get('/', async (req, res, next) => {
  try {
    const keylist = await notes.keylist();
    // console.log(`keylist ${util.inspect(keylist)}`);
    const keyPromises = keylist.map(key => {
      return notes.read(key);
    });
    const notelist = await Promise.all(keyPromises);
    // console.log(util.inspect(notelist));
    res.render('index', { title: 'Notes', notelist: notelist });
  } catch (err) {
    next(err);
  }
});
```

We showed the outline for this earlier, and having defined the `Notes` data storage model, we can fill in this function.

This uses the `AbstractNotesStore` API that we designed earlier. The `keylist` method returns a list of the key values for notes currently stored by the application. Then, it uses the `read` method to retrieve each note and pass that list to a template that renders the home page. This template will render a list of the notes.

What's the best way to retrieve all the notes? We could have written a simple `for` loop, as follows:

```
const keylist = await notes().keylist();
const notelist = [];
for (key of keylist) {
  let note = await notes.read(key);
  notelist.push({ key: note.key, title: note.title });
}
```

This has the advantage of being simple to read since it's a simple `for` loop. The problem is that this loop reads the notes one at a time. It's possible that reading the notes in parallel is more efficient since there's an opportunity to interweave the processing.

The `Promise.all` function executes an array of Promises in parallel, rather than one at a time. The `keyPromises` variable ends up being an array of Promises, each of which is executing `notes.read` to retrieve a single note.

The `map` function in the arrays converts (or maps) the values of an input array to produce an output array with different values. The output array has the same length as the input array, and the entries are a one-to-one mapping of the input value to an output value. In this case, we map the keys in `keylist` to a Promise that's waiting on a function that is reading each note. Then, `Promise.all` waits for all the Promises to resolve into either success or failure states.

The output array, `notelist`, will be filled with the notes once all the Promises succeed. If any Promises fail, they are rejected—in other words, an exception will be thrown instead.

The `notelist` array is then passed into the `view` template that we're about to write.

But first, we need a page layout template. Create a file, `views/layout.hbs`, containing the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{title}}</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    {{> header }}
    {{body}}
  </body>
</html>
```

This is the file generated by `express-generator`, with the addition of a `header` partial for the page header.

Remember that in the Fibonacci application, we used a *partial* to store the HTML snippet for the navigation. Partials are just that—HTML template snippets that can be reused in one or more templates. In this case, the `header` partial will appear on every page and serve as a common navigation bar across the application. Create `partials/header.hbs`, containing the following:

```
<header>
  <h1>{{ title }}</h1>
  <div class='navbar'>
    <p><a href='/'>Home</a> | <a href='/notes/add'>ADD Note</a></p>
```

```
</div>
</header>
```

This simply looks for a variable, `title`, which should have the page title. It also outputs a navigation bar containing a pair of links—one to the home page and another to `/notes/add`, where the user will be able to add a new note.

Now, let's rewrite `views/index.hbs` to this:

```
<ul>
{{#each notelist}}
<li>{{ key }}:
<a href="/notes/view?key={{ key }}">{{ title }}</a>
</li>
{{/each}}
</ul>
```

This simply steps through the array of note data and formats a simple listing. Each item links to the `/notes/view` URL with a `key` parameter. We have yet to write code to handle that URL, but will obviously display the note. Another thing to note is that no HTML for the list is generated if `notelist` is empty.

There is, of course, a whole lot more that could be put into this. For example, it's easy to add jQuery support to every page just by adding the appropriate `script` tags here.

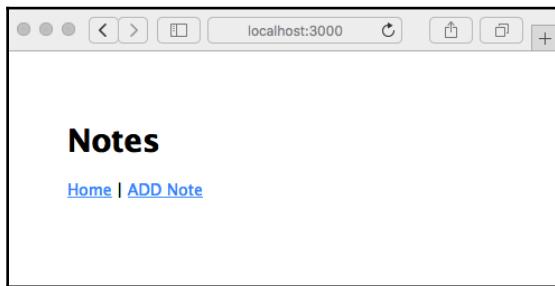
We have now written enough to run the application, so let's view the home page:

```
$ DEBUG=notes:* npm start

> notes@0.0.0 start /Users/David/chap05/notes
> node ./bin/www

notes:server Listening on port 3000 +0ms
GET / 200 87.300 ms - 308
GET /stylesheets/style.css 200 27.744 ms - 111
```

If we visit `http://localhost:3000`, we will see the following page:



Because there aren't any notes (yet), there's nothing to show. Clicking on the **Home** link just refreshes the page. Clicking on the **ADD Note** link throws an error because we haven't (yet) implemented that code. This shows that the provided error handler in `app.mjs` is performing as expected.

Having implemented the home page, we need to implement the various pages of the application. We will start with the page for creating new notes, and then we will implement the rest of the CRUD support.

Adding a new note – create

If we click on the **ADD Note** link, we get an error because the application doesn't have a route configured for the `/notes/add` URL; we need to add one. To do that, we need a controller module for the notes that defines all the pages for managing notes in the application.

In `app.mjs`, uncomment the two lines dealing with `notesRouter`:

```
import { router as indexRouter } from './routes/index.mjs';
import { router as notesRouter } from './routes/notes.mjs';
...
app.use('/', indexRouter);
app.use('/notes', notesRouter);
```

We'll end up with this in `app.mjs`. We import both routers and then add them to the application configuration.

Create a file named `routes/notes.mjs` to hold `notesRouter`, starting with the following content:

```
// const util = require('util');
import { default as express } from 'express';
import { NotesStore as notes } from '../app.mjs';
```

```
export const router = express.Router();

// Add Note.
router.get('/add', (req, res, next) => {
  res.render('noteedit', {
    title: "Add a Note",
    docreate: true,
    notekey: '',
    note: undefined
  });
});
```

This handles the /notes/add URL corresponding to the link in `partials/header.hbs`. It simply renders a template, `noteedit`, using the provided data.

In the `views` directory, add the corresponding template, named `noteedit.hbs`, containing the following:

```
<form method='POST' action='/notes/save'>
<input type='hidden' name='docrete' value='<%=
  docreate ? "create" : "update">'>
<p>Key:
{{#if docreate}}
  <input type='text' name='notekey' value=''/>
{{else}}
  {{#if note }}{{notekey}}{{/if}}
  <input type='hidden' name='notekey'
  value='{{#if note }}{{notekey}}{{/if}}'>
{{/if}}
</p>
<p>Title: <input type='text' name='title'
  value='{{#if note }}{{note.title}}{{/if}}' /></p>
<br/><textarea rows=5 cols=40 name='body'>
  {{#if note }}{{note.body}}{{/if}}</textarea>
<br/><input type='submit' value='Submit' />
</form>
```

This template supports both creating new notes and updating existing notes. We'll reuse this template to support both scenarios via the `docrete` flag.

Notice that the `note` and `notekey` objects passed to the template are empty in this case. The template detects this condition and ensures that the input areas are empty. Additionally, a flag, `docrete`, is passed in so that the form records whether it is being used to create or update a note. At this point, we're adding a new note, so no `note` objects exist. The template code is written defensively to not throw errors.

When creating HTML forms like this, you have to be careful with using whitespace in the elements holding the values. Consider a scenario where the `<textarea>` element was instead formatted like this:

```
<br/><textarea rows=5 cols=40 name='body'>
{{#if note }}{{note.body}}{{/if}}
</textarea>
```

By normal coding practices, this looks alright, right? It's nicely indented, with the code arranged for easy reading. The problem is that extra whitespace ends up being included in the `body` value when the form is submitted to the server. That extra whitespace is added because of the nicely indented code. To avoid that extra whitespace, we need to use the angle brackets in the HTML elements that are directly adjacent to the Handlebars code to insert the value. Similar care must be taken with the elements with the `value=` attributes, ensuring no extra whitespace is within the `value` string.

This template is a form that will post its data to the `/notes/save` URL. If you were to run the application now, it would give you an error message because no route is configured for that URL.

To support the `/notes/save` URL, add it to `routes/notes.mjs`:

```
// Save Note (update)
router.post('/save', async (req, res, next) => {
  try {
    let note;
    if (req.body.doCreate === "create") {
      note = await notes.create(req.body.notekey,
        req.body.title, req.body.body);
    } else {
      note = await notes.update(req.body.notekey,
        req.body.title, req.body.body);
    }
    res.redirect('/notes/view?key=' + req.body.notekey);
  } catch (err) { next(err); }
});
```

Because this URL will also be used for both creating and updating notes, we check the `doCreate` flag to call the appropriate model operation.

Both `notes.create` and `notes.update` are `async` functions, meaning we must use `await`.

This is an HTTP POST handler. Because of the `bodyParser` middleware, the form data is added to the `req.body` object. The fields attached to `req.body` correspond directly to elements in the HTML form.

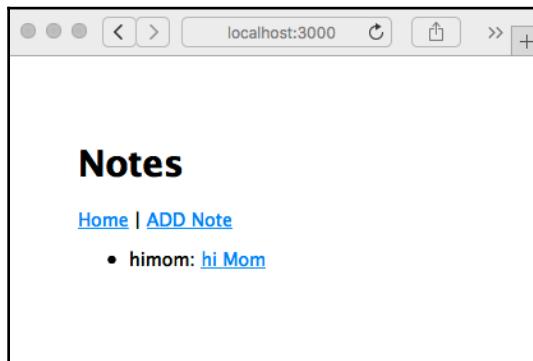
In this, and most of the other router functions, we use the `try/catch` construct that we discussed earlier to ensure errors are caught and forwarded correctly to Express. The difference between this and the preceding `/notes/add` router function is whether the router uses an `async` callback function. In this case, it is an `async` function, whereas for `/notes/add`, it is not `async`. Express knows how to handle errors in non-`async` callbacks, but it does not know how to handle errors in `async` callback functions.

Now, we can run the application again and use the **Add a Note** form:

The screenshot shows a web browser window with the URL `localhost:3000/notes/add` in the address bar. The page title is **Add a Note**. Below the title are two links: [Home](#) and [ADD Note](#). There are two input fields: one labeled **Key:** with the value `himom`, and another labeled **Title:** with the value `Hi Mom`. Below these is a large text area containing the text `This is where we say thanks for everything`. At the bottom of the form is a **Submit** button.

However, upon clicking on the **Submit** button, we get an error message. This is because there isn't anything (yet) to implement the `/notes/view` URL.

You can modify the URL in the Location box to revisit `http://localhost:3000`, and you'll see something similar to the following screenshot on the home page:



The note is actually there; we just need to implement `/notes/view`. Let's get on with that.

Viewing notes – read

Now that we've looked at how to create notes, we need to move on to reading them. This means implementing controller logic and view templates for the `/notes/view` URL.

Add the following router function to `routes/notes.mjs`:

```
// Read Note (read)
router.get('/view', async (req, res, next) => {
  try {
    let note = await notes.read(req.query.key);
    res.render('noteview', {
      title: note ? note.title : "",
      notekey: req.query.key, note: note
    });
  } catch (err) { next(err); }
});
```

Because this route is mounted on a router handling `/notes`, this route handles `/notes/view`.

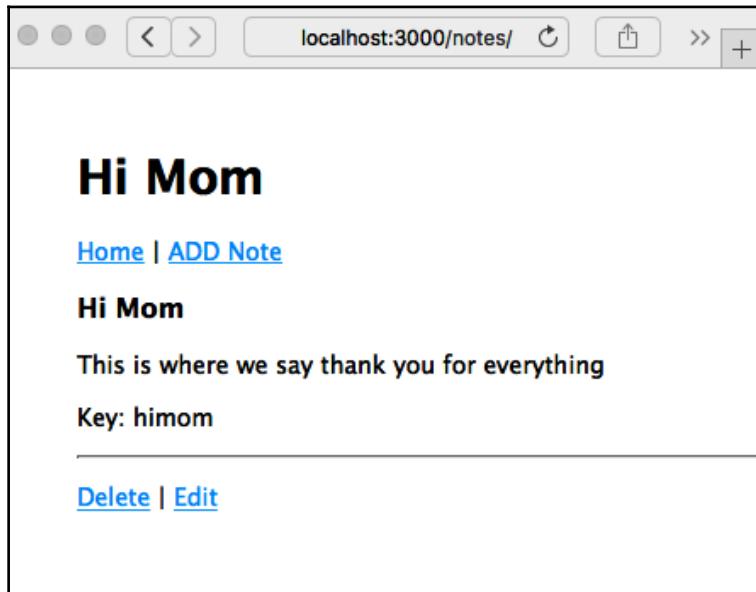
The handler simply calls `notes.read` to read the note. If successful, the note is rendered with the `noteview` template. If something goes wrong, we'll instead display an error to the user through Express.

Add the `noteview.hbs` template to the `views` directory, referenced by the following code:

```
 {{#if note}}<h3>{{ note.title }}</h3>{{/if}}
 {{#if note}}<p>{{ note.body }}</p>{{/if}}
 <p>Key: {{ notekey }}</p>
 {{#if notekey }}
 <hr/>
 <p><a href="/notes/destroy?key={{notekey}}">Delete</a>
 | <a href="/notes/edit?key={{notekey}}">Edit</a></p>
 {{/if}}
```

This is straightforward; we are taking data out of the `note` object and displaying it using HTML. At the bottom are two links—one to `/notes/destroy` to delete the note and the other to `/notes/edit` to edit it.

Neither of these corresponding codes exists at the moment, but that won't stop us from going ahead and executing the application:



As expected, with this code, the application correctly redirects to /notes/view, and we can see our handiwork. Also, as expected, clicking on either the **Delete** or **Edit** links will give us an error because the code hasn't yet been implemented.

We'll next create the code to handle the **Edit** link and later, one to handle the **Delete** link.

Editing an existing note – update

Now that we've looked at the `create` and `read` operations, let's look at how to update or edit a note.

Add the following router function to `routes/notes.mjs`:

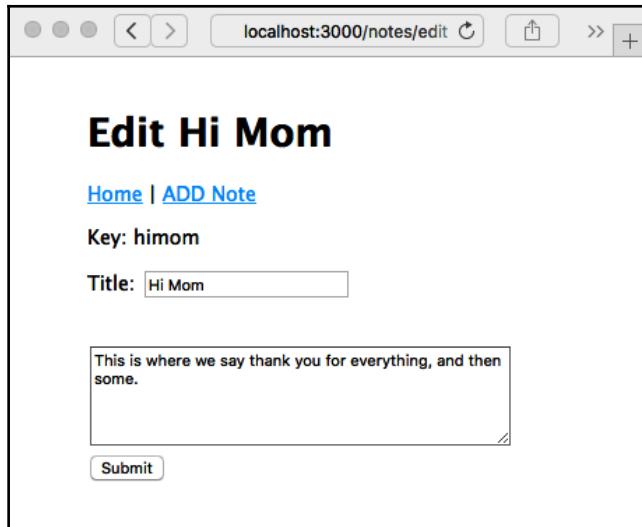
```
// Edit note (update)
router.get('/edit', async (req, res, next) => {
  try {
    const note = await notes.read(req.query.key);
    res.render('noteedit', {
      title: note ? ("Edit " + note.title) : "Add a Note",
      docreate: false,
      notekey: req.query.key, note: note
    });
  } catch (err) { next(err); }
});
```

This handles the /notes/edit URL.

We're reusing the `noteedit.hbs` template because it can be used for both the `create` and `update/edit` operations. Notice that we pass `false` for `docrete`, informing the template that it is to be used for editing.

In this case, we first retrieve the `note` object and then pass it through to the template. This way, the template is set up for editing, rather than note creation. When the user clicks on the **Submit** button, we end up in the same /notes/save route handler shown in the preceding screenshot. It already does the right thing—calling the `notes.update` method in the model, rather than `notes.create`.

Because that's all we need to do, we can go ahead and rerun the application:



Click on the **Submit** button here and you will be redirected to the /notes/view screen, where you will then be able to read the newly edited note. Back at the /notes/view screen, we've just taken care of the **Edit** link, but the **Delete** link still produces an error.

Therefore, we next need to implement a page for deleting notes.

Deleting notes – destroy

Now, let's look at how to implement the /notes/destroy URL to delete notes.

Add the following router function to routes/notes.mjs:

```
// Ask to Delete note (destroy)
router.get('/destroy', async (req, res, next) => {
  try {
    const note = await notes.read(req.query.key);
    res.render('notedestroy', {
      title: note ? note.title : "",
      notekey: req.query.key, note: note
    });
  } catch (err) { next(err); }
});
```

Destroying a note is a significant step, if only because there's no trash can to retrieve it from if the user makes a mistake. Therefore, we need to ask the user whether they're sure that they want to delete the note. In this case, we retrieve the note and then render the following page, displaying a question to ensure they definitely want to delete the note.

Add a `notedestroy.hbs` template to the `views` directory:

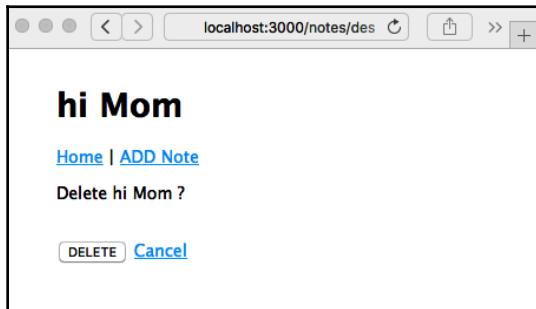
```
<form method='POST' action='/notes/destroy/confirm'>
<input type='hidden' name='notekey' value='{{#if
note}}{{notekey}}{{/if}}'>
<p>Delete {{note.title}}?</p>
<br/><input type='submit' value='DELETE' />
<a href="/notes/view?key={{#if note}}{{notekey}}{{/if}}">Cancel</a>
</form>
```

This is a simple form that asks the user to confirm by clicking on the button. The **Cancel** link just sends them back to the `/notes/view` page. Clicking on the **Submit** button generates a POST request on the `/notes/destroy/confirm` URL.

This URL needs a request handler. Add the following code to `routes/notes.mjs`:

```
// Really destroy note (destroy)
router.post('/destroy/confirm', async (req, res, next) => {
  try {
    await notes.destroy(req.body.notekey);
    res.redirect('/');
  } catch (err) { next(err); }
});
```

This calls the `notes.destroy` function in the model. If it succeeds, the browser is redirected to the home page. If not, an error message is shown to the user. Rerunning the application, we can now view it in action:



Now that everything is working in the application, you can click on any button or link and keep all the notes you want.

We've implemented a bare-bones application for managing notes. Let's now see how to change the look, since in the next chapter, we'll implement a mobile-first UI.

Theming your Express application

The Express team has done a decent job of making sure Express applications look okay out of the gate. Our Notes application won't win any design awards, but at least it isn't ugly. There's a lot of ways to improve it, now that the basic application is running. Let's take a quick look at theming an Express application. In Chapter 6, *Implementing the Mobile-First Paradigm*, we'll take a deeper dive into this, focusing on that all-important goal of addressing the mobile market.

If you're running the Notes application using the recommended method, `npm start`, a nice log of activity is being printed in your console window. One of these is the following:

```
GET /stylesheets/style.css 304 0.702 ms - -
```

This is due to the following line of code, which we put into `layout.hbs`:

```
<link rel='stylesheet' href='/stylesheets/style.css' />
```

This file was autogenerated for us by the Express generator at the outset and was dropped in the `public` directory. The `public` directory is managed by the Express static file server, using the following line in `app.mjs`:

```
app.use(express.static(path.join(__dirname, 'public')));
```

Therefore, the CSS stylesheet is at `public/stylesheets/style.css`, so let's open it and take a look:

```
body {  
  padding: 50px;  
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;  
}  
  
a {  
  color: #00B7FF;  
}
```

Something that leaps out is that the application content has a lot of whitespace at the top and left-hand sides of the screen. The reason for this is that the `body` tags have the `padding: 50px` style. Changing it is a quick business.

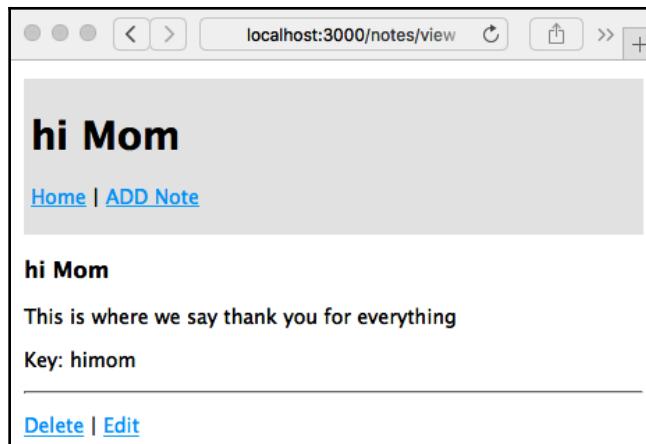
Since there is no caching in the Express static file server, we can simply edit the CSS file and reload the page, and the CSS will be reloaded as well.

Let's make a couple of tweaks:

```
body {  
  padding: 5px;  
  ..  
}  
..  
header {  
  background: #eeeeee;  
  padding: 5px;  
}
```

This changes the padding and also adds a gray box around the header area.

As a result, we'll have the following:



We're not going to win any design awards with this either, but there's the beginning of some branding and theming possibilities. More importantly, it proves that we can make edits to the theming.

Generally speaking, through the way that we've structured the page templates, applying a site-wide theme is just a matter of adding appropriate code to `layout.hbs`, along with appropriate stylesheets and other assets.

In Chapter 6, *Implementing the Mobile-First Paradigm*, we will look at a simple method to add these frontend libraries to your application.

Before closing out this chapter, we want to think ahead to scaling the application to handle multiple users.

Scaling up – running multiple Notes instances

Now that we've got ourselves a running application, you'll have played around a bit and created, read, updated, and deleted many notes.

Suppose for a moment that this isn't a toy application, but one that is interesting enough to draw millions of users a day. Serving a high load typically means adding servers, load balancers, and many other things. A core part of this is to have multiple instances of the application running at the same time to spread the load.

Let's see what happens when you run multiple instances of the `Notes` application at the same time.

The first thing is to make sure the instances are on different ports. In `app.mjs`, you'll see that setting the `PORT` environment variable controls the port being used. If the `PORT` variable is not set, it defaults to `http://localhost:3000`, or what we've been using all along.

Let's open up `package.json` and add the following lines to the `scripts` section:

```
"scripts": {  
  "start": "cross-env DEBUG=notes:* node ./app.mjs",  
  "server1": "cross-env DEBUG=notes:* PORT=3001 node ./app.mjs",  
  "server2": "cross-env DEBUG=notes:* PORT=3002 node ./app.mjs"  
},
```

The `server1` script runs on `PORT 3001`, while the `server2` script runs on `PORT 3002`. Isn't it nice to have all of this documented in one place?

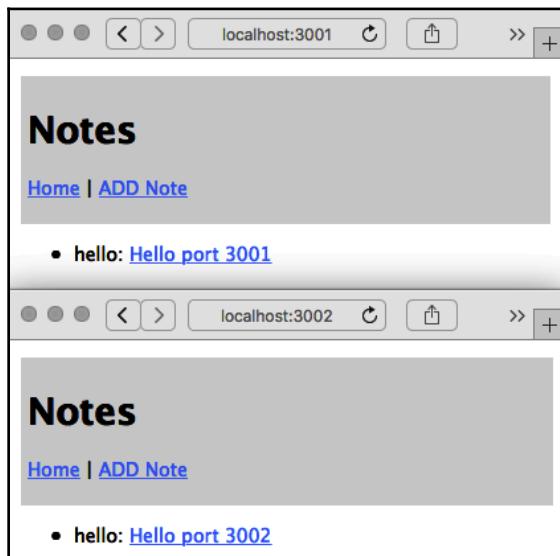
Then, in one command window, run the following:

```
$ npm run server1  
  
> notes@0.0.0 server1 /Users/David/chap05/notes  
> cross-env DEBUG=notes:* PORT=3001 node ./bin/www  
  
notes:server Listening on port 3001 +0ms
```

In another command window, run the following:

```
$ npm run server2  
  
> notes@0.0.0 server2 /Users/David/chap05/notes  
> cross-env DEBUG=notes:* PORT=3002 node ./bin/www  
  
notes:server Listening on port 3002 +0ms
```

This gives us two instances of the Notes application. Use two browser windows to visit `http://localhost:3001` and `http://localhost:3002`. Enter a couple of notes, and you might see something like this:



After editing and adding some notes, your two browser windows could look as in the preceding screenshot. The two instances do not share the same data pool; each is instead running in its own process and memory space. You add a note to one and it does not show on the other screen.

Additionally, because the model code does not persist data anywhere, the notes are not saved. You might have written the greatest Node.js programming book of all time, but as soon as the application server restarts, it's gone.

Typically, you run multiple instances of an application to scale performance. That's the old *throw more servers at it* trick. For this to work, the data, of course, must be shared, and each instance must access the same data source. Typically, this involves a database, and when it comes to user identity information, it might even entail armed guards.

All that means databases, more data models, unit testing, security implementation, a deployment strategy, and much more. Hold on—we'll get to all of that soon!

Summary

We've come a long way in this chapter.

We started by looking at the pyramid of doom and how Promise objects and async functions can help us tame asynchronous code. Because we're writing an Express application, we looked at how to use async functions in Express. We'll be using these techniques throughout this book.

We quickly moved on to writing the foundation of a real application with Express. At the moment, our application keeps its data in memory, but it has the basic functionality of what will become a note-taking application that supports real-time collaborative commenting on notes.

In the next chapter, we'll dip our toes into the water of responsive, mobile-friendly web design. Due to the growing popularity of mobile computing devices, it's become necessary to address mobile devices first before desktop computer users. In order to reach those millions of users a day, the Notes application users need a good user experience when using their smartphones.

In the following chapters, we'll keep growing the capabilities of the Notes application, starting with database storage models. But first, we have an important task in the next chapter—implementing a mobile-first UI using Bootstrap.

6

Implementing the Mobile-First Paradigm

Now that our first Express application is usable, we should act on the mantra of this age of software development: mobile-first. Mobile devices, whether they be smartphones, tablet computers, automobile dashboards, refrigerator doors, or bathroom mirrors, are taking over the world.

The primary considerations in designing for mobiles are the small screen sizes, the touch-oriented interaction, the fact that there's no mouse, and the somewhat different **User Interface (UI)** expectations. In 1997-8, when streaming video was first developed, video producers had to learn how to design video experiences for a viewport the size of a fig newton (an American snack food). Today, application designers have to contend with an application window the size of a playing card.

With the *Notes* application, our UI needs are modest, and the lack of a mouse doesn't make any difference to us.

In this chapter, we won't do much Node.js development. Instead, we'll do the following:

- Modify the Notes application templates for better mobile presentation.
- Edit Bootstrap SASS files to customize application theming.
- Install a third-party Bootstrap theme.
- Learn about Bootstrap 4.5, a popular framework for responsive UI design.

As of the time of writing, Bootstrap v5 has just entered the alpha phase. That makes it premature to adopt at this time, but we may wish to do so in the future. Going by the migration guide, much of Bootstrap will stay the same, or very similar, in version 5. However, the biggest change in version 5 is the dropping of the requirement for jQuery. Because we use jQuery fairly heavily in Chapter 9, *Dynamic Client/Server Interaction with Socket.IO*, this is a significant consideration.

By completing the tasks in the preceding list, we'll dip our toes in the water of what it means to be a full-stack web engineer. The goal of this chapter is to gain an introduction to an important part of application development, namely the UI, and one of the leading toolkits for web UI development.

Rather than just do mobile-first development because it's the popular thing, let's first try to understand the problem being solved.

Understanding the problem – the Notes app isn't mobile-friendly

Let's start by quantifying the problem. We need to explore how well (or not) the application behaves on a mobile device. This is simple to do:

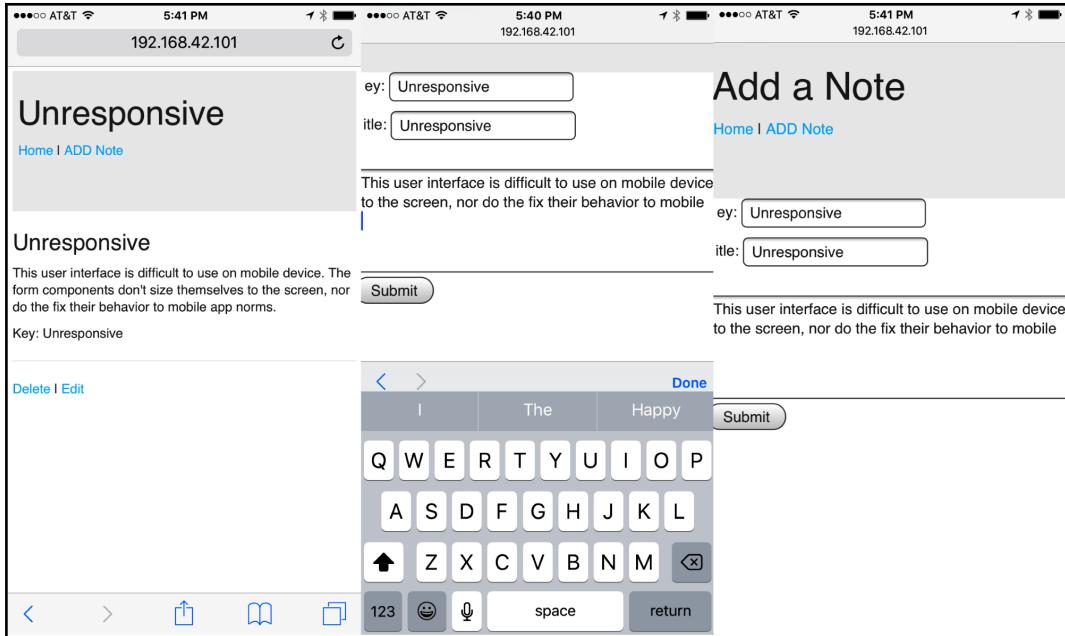
1. Start the *Notes* application. Determine the IP address of the host system.
2. Using your mobile device, connect to the service using the IP address, and browse around the *Notes* application, putting it through its paces and noting any difficulties.

Another way to approach this is to use your desktop browser, resizing it to be very narrow. The Chrome DevTools also includes a mobile device emulator. Either way, you can mimic the small screen size of a smartphone on your desktop.

To see a real UI problem on a mobile screen, edit `views/noteedit.hbs` and make this change:

```
<br/><textarea rows=5 cols=80 name='body'>
>{{#if note }}{{note.body}}{{/if}}</textarea>
```

What's changed is that we've added the `cols=80` parameter to set its width to be fixed at 80 columns. We want this `textarea` element to be overly large so that you can experience how a non-responsive web app appears on a mobile device. View the application on a mobile device and you'll see something like one of the screens in this screenshot:



Viewing a note works well on an iPhone 6, but the screen for editing/adding a note is not good. The text entry area is so wide that it runs off the side of the screen. Even though interaction with FORM elements works well, it's clumsy. In general, browsing the *Notes* application gives an acceptable mobile user experience that doesn't suck, but won't make our users leave rave reviews.

In other words, we have an example of a screen that works well on the developers' laptop but is horrid on the target platform. By following the mobile-first paradigm, the developer is expected to constantly check the behavior in a mobile web browser, or else the mobile view in the Chrome developer tool, and to design accordingly.

This gives us an idea of the sort of problem that responsive web design aims to correct. Before implementing a mobile-first design in our *Notes* app, let's discuss some of the theory behind responsive web design.

Learning the mobile-first paradigm theory

Mobile devices have a smaller screen, are generally touch-oriented, and have different user experience expectations than a desktop computer.

To accommodate smaller screens, we use **responsive web design** techniques. This means designing the application to accommodate the screen size and ensuring websites provide optimal viewing and interaction across a wide range of devices. Techniques include changing font sizes, rearranging elements on the screen, using collapsible elements that open when touched, and resizing images or videos to fit available space. This is called **responsive** because the application responds to device characteristics by making these changes.



By *mobile-first*, we mean that you design the application to work well on a mobile device first, and then move on to devices with larger screens. It's about prioritizing mobile devices first.

The primary technique is using media queries in stylesheets to detect device characteristics. Each media query section targets a range of devices, using a CSS declaration to appropriately restyle content.

Let's consult a concrete example. The **Twenty Twelve** theme for WordPress has a straightforward responsive design implementation. It's not built with any framework, so you can see clearly how the mechanism works, and the stylesheet is small enough to be easily digestible. We're not going to use this code anywhere; instead, it is intended as a useful example of implementing a responsive design.



You can refer to the source code for the Twenty Twelve theme in the WordPress repository at
<https://themes.svn.wordpress.org/twentytwelve/1.9/style.css>

.

The stylesheet starts with a number of **resets**, where the stylesheet overrides some typical browser style settings with clear defaults. Then, the bulk of the stylesheet defines styling for mobile devices. Toward the bottom of the stylesheet is a section labeled **Media queries** where, for certain sized screens, the styles defined for mobile devices are overridden to work on devices with larger screens.

It does this with the following two media queries:

```
@media screen and (min-width: 600px) { /* Screens above 600px width */  
}  
@media screen and (min-width: 960px) { /* Screens above 960px width */  
}
```

The first segment of the stylesheet configures the page layout for all devices. Next, for any browser viewport at least 600px wide, it reconfigures the page to display on the larger screen. Then, for any browser viewport at least 960px wide, it is reconfigured again. The stylesheet has a final media query to cover print devices.

These widths are what's called a **breakpoint**. Those threshold viewport widths are the points where the design changes itself around. You can see breakpoints in action by going to any responsive website, then resizing the browser window. Watch how the design jumps at certain sizes. Those are the breakpoints chosen by the author of that website.

There's a wide range of differing opinions about the best strategy to choose your breakpoints. Do you target specific devices or do you target general characteristics? The Twenty Twelve theme did fairly well on mobile devices using only two viewport-size media queries. The CSS-Tricks blog has posted an extensive list of specific media queries for every known device, which is available at <https://css-tricks.com/snippets/css/media-queries-for-standard-devices/>.

We should at least target these devices:

- **Small:** This includes iPhone 5 SE.
- **Medium:** This can refer to tablet computers or larger smartphones.
- **Large:** This includes larger tablet computers or smaller desktop computers.
- **Extra-large:** This refers to larger desktop computers and other large screens.
- **Landscape/portrait:** You may want to create a distinction between landscape mode and portrait mode. Switching between the two of course changes viewport width, possibly pushing it past a breakpoint. However, your application may need to behave differently in the two modes.

That's enough theory of responsive web design to get us started. In our *Notes* application, we'll work on using touch-friendly UI components and using Bootstrap to scale the user experience based on screen size. Let's get started.

Using Twitter Bootstrap on the Notes application

Bootstrap is a mobile-first framework consisting of HTML5, CSS3, and JavaScript code providing a comprehensive set of world-class, responsive web design components. It was developed by engineers at Twitter and then released to the world in August 2011.

The framework includes code to retrofit modern features onto older browsers, a responsive 12-column grid system, and a long list of components (some using JavaScript) for building web applications and websites. It's meant to provide a strong foundation on which to build your application.



Refer to <http://getbootstrap.com> for more details about Bootstrap.

With this introduction to Bootstrap, let's proceed to set it up.

Setting up Bootstrap

The first step is to duplicate the code you created in the previous chapter. If, for example, you created a directory named `chap05/notes`, then create one named `chap06/notes` from the content of `chap05/notes`.

Now, we need to go about adding Bootstrap's code in the *Notes* application. The Bootstrap website suggests loading the required CSS and JavaScript files out of the Bootstrap (and jQuery) public CDN. While that's easy to do, we won't do this for two reasons:

- It violates the principle of keeping all dependencies local to the application and not relying on global dependencies.
- It makes our application dependent on whether the CDN is functioning.
- It prevents us from generating a custom theme.

Instead, we'll install a local copy of Bootstrap. There are several ways to install Bootstrap locally. For example, the Bootstrap website offers a downloadable TAR/GZIP archive (tarball). The better approach is an automated dependency management tool, and fortunately, the npm repository has all the packages we need.

The most straightforward choice is to use the Bootstrap (<https://www.npmjs.com/package/bootstrap>), Popper.js (<https://www.npmjs.com/package/popper.js>), and jQuery (<https://www.npmjs.com/package/jquery>) packages in the npm repository. These packages provide no Node.js modules and instead are frontend code distributed through npm. Many frontend libraries are distributed through the npm repository.

We install the packages using the following command:

```
$ npm install bootstrap@4.5.x --save
npm WARN bootstrap@4.5.0 requires a peer of jquery@1.9.1 - 3 but none
is installed. You must install peer dependencies yourself.
npm WARN bootstrap@4.5.0 requires a peer of popper.js@^1.16.0 but none
is installed. You must install peer dependencies yourself.

+ bootstrap@4.5.0
...
$ npm install jquery@3.5.x --save
+ jquery@3.5.1
$ npm install popper.js@1.16.x --save
+ popper.js@1.16.0
```

As we can see here, when we install Bootstrap, it helpfully tells us the corresponding versions of jQuery and Popper.js to use. But according to the Bootstrap website, we are to use a different version of jQuery than what's shown here. Instead, we are to use jQuery 3.5.x instead of 1.9.1, because 3.5.x has many security issues fixed.

On the npm page for the Popper.js package (<https://www.npmjs.com/package/popper.js>), we are told this package is deprecated, and that Popper.js v2 is available from the `@popperjs/core` npm package. However, the Bootstrap project tells us to use this version of Popper.js, so that's what we'll stick with.



The Bootstrap *Getting Started* documentation explicitly says to use jQuery 3.5.1 and Popper 1.16.0, as of the time of writing, as you can see at <https://getbootstrap.com/docs/4.5/getting-started/introduction/>.

What's most important is to see what got downloaded:

```
$ ls node_modules/bootstrap/dist/*
... directory contents
$ ls node_modules/jquery/dist
... directory contents
$ ls node_modules/popper.js/dist
... directory contents
```

Within each of these directories are the CSS and JavaScript files that are meant to be used in the browser. More importantly, these files are located in a given directory whose pathname is known—specifically, the directories we just inspected.

Let's see how to configure our Notes app to use those three packages on the browser side, as well as set up Bootstrap support in the page layout templates.

Adding Bootstrap to the Notes application

In this section, we'll first load Bootstrap CSS and JavaScript in the page layout templates, and then we'll ensure that the Bootstrap, jQuery, and Popper packages are available to be used. We have made sure those libraries are installed in `node_modules`, so we need to make sure Notes knows to serve the files as static assets to web browsers.

On the Bootstrap website, they give a recommended HTML structure for pages. We'll be interpolating from their recommendation to instead use the local copies of Bootstrap, jQuery, and Popper that we just installed.



Refer to the *Getting Started* page at <https://getbootstrap.com/docs/4.5/getting-started/introduction/>.

What we'll do is modify `views/layout.hbs` to match the template recommended by Bootstrap, by making the changes shown in bold text:

```
<!doctype html>
<html lang="en">
  <head>
    <title>{{title}}</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, shrink-to-
      fit=no">

    <link rel="stylesheet"
      href="/assets/vendor/bootstrap/css/bootstrap.min.css">
    <link rel='stylesheet' href='/assets/stylesheets/style.css' />
  </head>
  <body>
    {{> header }}
    {{> body}}
    <!-- jQuery first, then Popper.js, then Bootstrap JS -->
```

```
<script src="/assets/vendor/jquery/jquery.min.js"></script>
<script src="/assets/vendor/popper.js/popper.min.js"></script>
<script src=
    "/assets/vendor/bootstrap/js/bootstrap.min.js"></script>
</body>
</html>
```

This is largely the template shown on the Bootstrap site, incorporated into the previous content of `views/layout.hbs`. Our own stylesheet is loaded following the Bootstrap stylesheet, giving us the opportunity to override anything in Bootstrap we want to change. What's different is that instead of loading Bootstrap, Popper.js, and jQuery packages from their respective CDNs, we use the path `/assets/vendor/product-name` instead.



This is the same as recommended on the Bootstrap website except the URLs point to our own site rather than relying on the public CDN. The pathname prefix, `/assets/vendor`, is routinely used to hold code provided by a third party.

This `/assets/vendor` URL is not currently recognized by the *Notes* application. To add this support, edit `app.mjs` to add these lines:

```
app.use(express.static(path.join(__dirname, 'public')));
app.use('/assets/vendor/bootstrap', express.static(
  path.join(__dirname, 'node_modules', 'bootstrap', 'dist')));
app.use('/assets/vendor/jquery', express.static(
  path.join(__dirname, 'node_modules', 'jquery', 'dist')));
app.use('/assets/vendor/popper.js', express.static(
  path.join(__dirname, 'node_modules', 'popper.js', 'dist', 'umd')));
```

We're again using the `express.static` middleware to serve asset files to browsers visiting the *Notes* application. Each of these pathnames is where npm installed the Bootstrap, jQuery, and Popper libraries.

There is a special consideration for the Popper.js library. In the `popper.js/dist` directory, the team distributes a library in the ES6 module syntax. At this time, we cannot trust all browsers to support ES6 modules. In `popper.js/dist/umd` is a version of the Popper.js library that works in all browsers. We have therefore set the directory appropriately.

Within the `public` directory, we have a little house-keeping to do. When express-generator set up the initial project, it generated `public/images`, `public/javascripts`, and `public/stylesheets` directories. Hence the URLs for each start with `/images`, `/javascripts`, and `/stylesheets`. It's cleaner to give such files a URL starting with the `/assets` directory. To implement that change, start by moving the files around as follows:

```
$ mkdir public/assets
$ mv public/images/ public/javascripts/ public/stylesheets/
  public/assets/
```

We now have our asset files, including Bootstrap, Popper.js, and jQuery, all available to the *Notes* application under the `/assets` directory. Referring back to `views/layout.hbs`, notice that we said to change the URL for our stylesheet to `/assets/stylesheets/style.css`, which matches this change.

We can now try this out by running the application:

```
$ npm start
> notes@0.0.0 start /Users/David/chap06/notes
> cross-env DEBUG=notes:* node ./bin/www

  notes:server Listening on port 3000 +0ms
GET / 200 306.660 ms - 883
GET /stylesheets/style.css 404 321.057 ms - 2439
GET /assets/stylesheets/style.css 200 160.371 ms - 165
GET /assets/vendor/bootstrap/js/bootstrap.min.js 200 157.459 ms -
50564
GET /assets/vendor/popper.js/popper.min.js 200 769.508 ms - 18070
GET /assets/vendor/jquery/jquery.min.js 200 777.988 ms - 92629
GET /assets/vendor/bootstrap/css/bootstrap.min.css 200 788.028 ms -
127343
```

The onscreen differences are minor, but this is the necessary proof that the CSS and JavaScript files for Bootstrap are being loaded. We have accomplished the first major goal—using a modern, mobile-friendly framework to implement a mobile-first design.

Before going on to modify the look of the application, let's talk about other available frameworks.

Alternative layout frameworks

Bootstrap isn't the only JavaScript/CSS framework providing a responsive layout and useful components. Of course, all the other frameworks have their own claims to fame. As always, it is up to each project team to choose the technologies they use, and of course, the marketplace is always changing as new libraries become available. We're using Bootstrap in this project because of its popularity. These other frameworks are worthy of a look:

- Pure.css (<https://purecss.io/>): A responsive CSS framework with an emphasis on a small code footprint.
- Picnic CSS (<https://picnicss.com/>): A responsive CSS framework emphasizing small size and beauty.
- Bulma (<https://bulma.io/>): A responsive CSS framework that's self-billed as very easy to use.
- Shoelace (<https://shoelace.style/>): A CSS framework emphasizing using future CSS, meaning it uses CSS constructs at the bleeding edge of CSS standardization. Since most browsers don't support those features, cssnext (<http://cssnext.io/>) is used to retrofit that support. Shoelace uses a grid layout system based on Bootstrap's grid.
- PaperCSS (<https://www.getpapercss.com/>): An informal CSS framework that looks like it was hand-drawn.
- Foundation (<https://foundation.zurb.com/>): Self-described as the most advanced responsive frontend framework in the world.
- Base (<http://getbase.org/>): A lightweight modern CSS framework.

HTML5 Boilerplate (<https://html5boilerplate.com/>) is an extremely useful basis from which to code the HTML and other assets. It contains the current best practices for the HTML code in web pages, as well as tools to normalize CSS support and configuration files for several web servers.

Browser technologies are also improving rapidly, with layout techniques being one area. The Flexbox and CSS Grid layout systems are a significant advance in making HTML content layout much easier than older techniques.

Flexbox and CSS Grids

Two new technologies impacting web application development are these new CSS layout methodologies. The CSS3 committee has been working on several fronts, including page layout.

In the distant past, we used nested HTML tables for page layout. That is a bad memory that we don't have to revisit. More recently, we've been using a box model using `<div>` elements, and even at times using absolute or relative placement techniques. All these techniques have been suboptimal in several ways, some more than others.

One popular layout technique is to divide the horizontal space into columns and assign a certain number of columns to each thing on the page. With some frameworks, we can even have nested `<div>` elements, each with their own set of columns. Bootstrap 3, and other modern frameworks, used that layout technique.

The two new CSS layout methodologies, Flexbox (https://en.wikipedia.org/wiki/CSS_Flex_box_layout) and CSS Grids (https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout), are a significant improvement over all previous methodologies. We are mentioning these technologies because they're both worthy of attention.

With Bootstrap 4, the Bootstrap team chose to go with Flexbox. Therefore, under the hood are Flexbox CSS constructs.

Having set up Bootstrap, and having learned some background to responsive web design, let's dive right in and start implementing responsive design in *Notes*.

Mobile-first design for the Notes application

When we added CSS and JavaScript for Bootstrap et al., that was only the start. To implement a responsive mobile-friendly design, we need to modify every template to use Bootstrap components. Bootstrap's features, in version 4.x, are grouped into four areas:

- **Layout:** Declarations to control the layout of HTML elements, supporting different layouts based on device size

- **Content:** For regularizing the look of HTML elements, typography, images, tables, and more
- **Components:** A comprehensive set of UI elements including navigation bars, buttons, menus, popups, forms, carousels, and more to make it easy to implement applications
- **Utilities:** Additional tools to aid in tweaking the presentation and layout of HTML elements

The Bootstrap documentation is full of what we might call *recipes*, to implement the structure of HTML elements for certain Bootstrap components or effects. A key to the implementation is that Bootstrap effects are triggered by adding the correct HTML class declaration to each HTML component.

Let's start with page layout using Bootstrap.

Laying the Bootstrap grid foundation

Bootstrap uses a 12-column grid system to control layout, giving applications a responsive mobile-first foundation on which to build. When correctly set up, a layout using Bootstrap components can automatically rearrange components for different sized screens between extra small up to large desktop computers. The method relies on `<div>` elements with classes to describe the role each `<div>` plays in the layout.

The basic layout pattern in Bootstrap is as follows:

```
<div class="container-fluid"> <!-- or just .container -->
  <div class="row">
    <div class="col-sm-3">Column 1 content</div> <!-- 25% -->
    <div class="col-sm-9">Column 2 content</div> <!-- 75% -->
  </div>
  <div class="row">
    <div class="col-sm-3">Column 1 content</div> <!-- 25% -->
    <div class="col-sm-6">Column 2 content</div> <!-- 50% -->
    <div class="col-sm-3">Column 3 content</div> <!-- 25% -->
  </div>
</div>
```

This is a generic Bootstrap layout example, not anything we're putting into the *Notes* app. Notice how each layer of the layout relies on different class declarations. This fits Bootstrap's pattern of declaring behavior by using classes.

In this case, we're showing a typical page layout of a container, containing two rows, with two columns on the first row and three columns on the second. The outermost layer uses the `.container` or `.container-fluid` elements. Containers provide a means to center or horizontally pad the content. Containers marked as `.container-fluid` act as if they have `width: 100%`, meaning they expand to fill the horizontal space.

A `.row` is what it sounds like, a "row" of a structure that's somewhat like a table. Technically, a row is a wrapper for columns. Containers are wrappers for rows, and rows are wrappers for columns, and columns contain the content displayed to our users.

Columns are marked with variations of the `.col` class. With the basic column class, `.col`, the columns are divided equally into the available space. You can specify a numerical column count to assign different widths to each column. Bootstrap supports up to 12 numbered columns, hence each row in the example adds up to 12 columns.

You can also specify a breakpoint to which the column applies:

- Using `col-xs` targets extra-small devices (smartphones, $<576\text{px}$).
- Using `col-sm` targets small devices ($\geq 576\text{px}$).
- Using `col-md` targets medium devices ($\geq 768\text{px}$).
- Using `col-lg` targets large devices ($\geq 992\text{px}$).
- Using `col-xl` targets extra-large devices ($\geq 1200\text{px}$).

Specifying a breakpoint, for example, `col-sm`, means that the declaration applies to devices matching that breakpoint or larger. Hence, in the example shown earlier, the column definitions were applied to `col-sm`, `col-md`, `col-lg`, and `col-xl` devices, but not to `col-xs` devices.

The column count is appended to the class name. That means using `col-#` when not targeting a breakpoint, for example, `col-4`, or `col-{breakpoint}-#` when targeting a breakpoint, for example, `col-md-4`, to target a space four columns wide on medium devices. If the columns add up to more than 12, the columns beyond the twelfth column wrap around to become a new row. The word `auto` can be used instead of a numerical column count to size the column to the natural width of its contents.

It's possible to mix and match to target multiple breakpoints:

```
<div class="container-fluid">
  <div class="row">
    <div class="col-xs-9 col-md-3 col-lg-6">Column 1 content</div>
    <div class="col-xs-3 col-md-9 col-lg-6">Column 2 content</div>
  </div>
  ...
</div>
```

This declares three different layouts, one for extra-small devices, another for medium devices, and the last for large devices.



The grid system can do a lot more. For details, see the documentation at <https://getbootstrap.com/docs/4.5/layout/overview/>.

This introduction gives us enough knowledge to start modifying the *Notes* application. Our next task is to better understand the structure of the application pages.

Responsive page structure for the Notes application

We could go through a whole user experience analysis of *Notes*, or get designers involved, and get the perfect page design for each screen of the *Notes* application. But the current *Notes* application design is the result of a developer coding up page designs that are functional and not ugly. Let's start by discussing the logic behind the structure of the page designs we have. Consider the following structure:

```
<!DOCTYPE html>
<html>
  <head> .. headerStuff </head>
  <body>
    .. pageHeader
    .. main content
    .. bottomOfPageStuff
  </body>
</html>
```

This is the general structure of the pages in *Notes*. The page content has two visible rows: the header and the main content. At the bottom of the page are invisible things such as the JavaScript files for Bootstrap and jQuery.

As it currently stands, the header contains a title for each page as well as navigation links so the user can browse the application. The content area is what changes from page to page, and is either about viewing content or editing content. The point is that for every page we have two sections for which to handle layout.

The question is whether `views/layout.hbs` should have any visible page layout. This template is used for the layout of every page in the application. The content of those pages is different enough that it seems `layout.hbs` cannot have any visible elements.

That's the decision we'll stick with for now. The next thing to set up is an icon library we can use for graphical buttons.

Using icon libraries and improving visual appeal

The world around us isn't constructed of words, but instead things. Hence, pictorial elements and styles, such as icons, can help computer software to be more comprehensible. Creating a good user experience should make our users reward us with more likes in the app store.

There are several icon libraries that can be used on a website. The Bootstrap team has a curated list at <https://getbootstrap.com/docs/4.5/extend/icons/>. For this project, we'll use Feather Icons (<https://feathericons.com/>). It is a conveniently available npm package at <https://www.npmjs.com/package/feather-icons>.

To install the package, run this command:

```
$ npm install feather-icons@4.25.x --save
```

You can then inspect the downloaded package and see that `./node_modules/feather-icons/dist/feather.js` contains browser-side code, making it easy to use the icons.

We make that directory available by mounting it in `app.mjs`, just as we did for the Bootstrap and jQuery libraries. Add this code to `app.mjs`:

```
app.use('/assets/vendor/feather-icons', express.static(
  path.join(__dirname, 'node_modules', 'feather-icons', 'dist')));
```

Going by the documentation, we must put this at the bottom of `views/layout.hbs` to enable `feather-icons` support:

```
<script src="/assets/vendor/feather-icons/feather.js"></script>
<script>
  feather.replace();
</script>
```

This loads the browser-side library and then invokes that library to cause the icons to be used.

To use one of the icons, use a `data-feather` attribute specifying one of the icon names, like so:

```
<i data-feather="circle"></i>
```

As suggested by the icon name, this will display a circle. The Feather Icons library looks for elements with the `data-feather` attribute, which the Feather Icons library uses to identify the SVG file to use. The Feather Icons library completely replaces the element where it finds the `data-feather` attribute. Therefore, if you want the icon to be a clickable link, it's necessary to wrap the icon definition with an `<a>` tag, rather than adding `data-feather` to the `<a>` tag.

Let's now redesign the page header to be a navigation bar, and use one of the Feather icons.

Responsive page header navigation bar

The header section we designed before contains a page title and a little navigation bar. Bootstrap has several ways to spiff this up, and even give us a responsive navigation bar that neatly collapses to a menu on small devices.

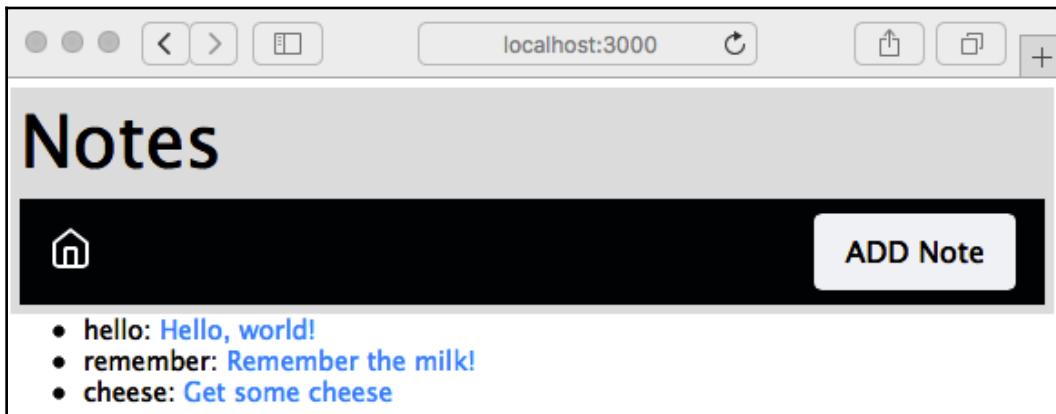
In `views/header.hbs`, make this change:

```
<header class="page-header">
<h1>{{ title }}</h1>
<nav class="navbar navbar-expand-md navbar-dark bg-dark">
  <a class="navbar-brand" href="/"><i data-feather="home"></i></a>
```

```
<button class="navbar-toggler" type="button"
    data-toggle="collapse" data-target="#navbarSupportedContent"
    aria-controls="navbarSupportedContent"
    aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
</button>
<div class="collapse navbar-collapse" id="navbarSupportedContent">
    <div class="navbar-nav col">
        {{#if breadcrumb}}
        <a class="nav-item nav-link" href='{{breadcrumb.url}}'>
            {{breadcrumb.title}}</a>
        {{/if}}
    </div>
    <a class="nav-item nav-link btn btn-light col-auto"
        href='/notes/add'>ADD Note</a>
</div>
</nav>
</header>
```

Adding `class="page-header"` informs Bootstrap that this is, well, the page header. Within that, we have the `<h1>` header as before, providing the page title, and then a responsive Bootstrap navbar.

By default, the navbar is expanded—meaning the components inside the navbar are visible—because of the `navbar-expand-md` class. This navbar uses a `navbar-toggler` button that governs the responsiveness of the navbar. By default, this button is hidden and the body of the navbar is visible. If the screen is small enough, the `navbar-toggler` is switched so it's visible and the body of the navbar becomes invisible, and when clicking on the now-visible `navbar-toggler`, a menu drops down containing the body of the navbar:



We chose the Feather Icons' *home* icon because that link goes to the *home page*. It's intended that the middle portion of the navbar will contain a breadcrumb trail as we navigate around the *Notes* application.

The **ADD Note** button is glued to the right-hand side with a little Flexbox magic. The container is a Flexbox, meaning we can use the Bootstrap classes to control the space consumed by each item. The breadcrumb area is the blank spot between the home icon and the **ADD Note** button. It is empty in this case, but the `<div>` element that would contain it is there and declared with `class="col"`, meaning that it takes up a column unit. The **ADD Note** button is, on the other hand, declared with `class="col-auto"`, meaning it takes up only the room required for itself. Therefore, the empty breadcrumb area that will expand to fill the available space, while the **ADD Note** button fills only its own space, and is therefore pushed over to the side.

Because it's the same application, the functionality all works; we're simply working on the presentation. We've added a few notes but the presentation of the list on the front page leaves a lot to be desired. The small size of the title is not very touch-friendly since it doesn't present a large target area for a fingertip. And can you explain why the `notekey` value has to be displayed on the home page? With that in mind, let's move on to fixing up the front page.

Improving the Notes list on the front page

The current home page has some simple text list that's not terribly touch-friendly, and showing the `key` at the front of the line might be inexplicable to the user. Let's fix this.

Edit `views/index.hbs` as follows, with the changed lines shown in bold:

```
<div class="container-fluid">
  <div class="row">
    <div class="col-12 btn-group-vertical" role="group">
      {{#each notelist}}
        <a class="btn btn-lg btn-block btn-outline-dark"
           href="/notes/view?key={{ key }}">{{ title }}</a>
      {{/each}}
    </div>
  </div>
</div>
```

The first change is to switch away from using a list and to use a vertical button group. The button group is a Bootstrap component that's what it sounds like, a group of buttons. By making the text links look and behave like buttons, we're improving the UI, especially its touch-friendliness. We chose the `btn-outline-dark` button style because it looks good in the UI. We use large buttons (`btn-lg`) that fill the width of the container (`btn-block`).

We eliminated showing the `notekey` value to the user. This information doesn't add anything to the user experience. Running the application, we get the following:



This is beginning to take shape, with a decent-looking home page that handles resizing very nicely and is touch-friendly. The buttons have been enlarged nicely to be large enough for big fingers to easily tap.

There's still something more to do with this since the header area is taking up a fair amount of space. We should always feel free to rethink a plan as we look at intermediate results. Earlier, we created a design for the header area, but on reflection, that design looks to be too large. The intention had been to insert a breadcrumb trail just to the right of the home icon, and to leave the `<h1>` title at the top of the header area. But this takes up too much vertical space, so we can tighten up the header and possibly improve the appearance.

Edit `partials/header.hbs` with the following line in bold:

```
<header class="page-header">
<nav class="navbar navbar-expand-md navbar-dark bg-dark">
  <a class="navbar-brand" href="/"><i data-feather="home"></i></a>
  <button class="navbar-toggler" type="button"
    data-toggle="collapse" data-target="#navbarSupportedContent">
```

```
        aria-controls="navbarSupportedContent"
        aria-expanded="false"
        aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
</button>
<div class="collapse navbar-collapse" id="navbarSupportedContent">
    <span class="navbar-text col">{{ title }}</span>
    <a class="nav-item nav-link btn btn-light col-auto"
       href='/notes/add'>ADD Note</a>
</div>
</nav>
</header>
```

This removes the `<h1>` tag at the top of the header area, immediately tightening up the presentation.

Within the `navbar-collapse` area, we've replaced what had been intended as the breadcrumb with a simple `navbar-text` component containing the page title. To keep the **ADD Note** button glued to the right, we're maintaining the `class="col"` and `class="col-auto"` settings:



Which header area design is better? That's a good question. Since beauty is in the eye of the beholder, both designs are probably equally good. What we have demonstrated is the ease with which we can update the design by editing the template files.

Let's now take care of the page for viewing notes.

Cleaning up the note viewing experience

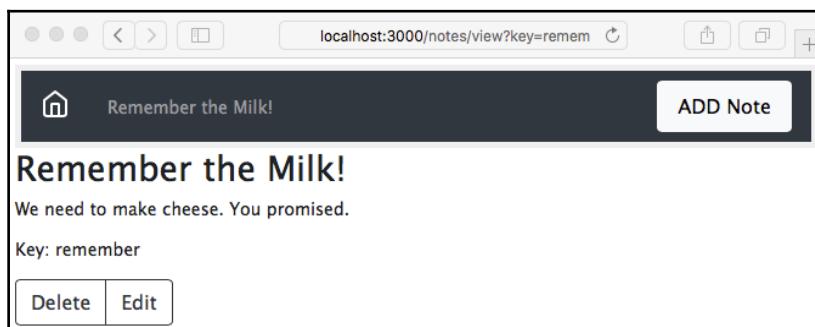
Viewing a note isn't bad, but the user experience can be improved. For example, the user does not need to see the `notekey`, meaning we might remove that from the display. Additionally, Bootstrap has nicer-looking buttons we can use.

In `views/noteview.hbs`, make these changes:

```
<div class="container-fluid">
  <div class="row"><div class="col-xs-12">
    {{#if note}}<h3>{{ note.title }}</h3>{{/if}}
    {{#if note}}<p>{{ note.body }}</p>{{/if}}
    <p>Key: {{ notekey }}</p>
  </div></div>
  {{#if notekey }}
    <div class="row"><div class="col-xs-12">
      <div class="btn-group">
        <a class="btn btn-outline-dark"
          href="/notes/destroy?key={{notekey}}"
          role="button">Delete</a>
        <a class="btn btn-outline-dark"
          href="/notes/edit?key={{notekey}}"
          role="button">Edit</a>
      </div>
    </div></div>
  {{/if}}
</div>
```

We have declared two rows, one for the note, and another for buttons for actions related to the note. Both are declared to consume all 12 columns, and therefore take up the full available width. The buttons are again contained within a button group, but this time a horizontal group rather than vertical.

Running the application, we get the following:



Do we really need to show the notekey to the user? We'll leave it there, but that's an open question for the user experience team. Otherwise, we've improved the note-reading experience.

Next on our list is the page for adding and editing notes.

Cleaning up the add/edit note form

The next major glaring problem is the form for adding and editing notes. As we said earlier, it's easy to get the text input area to overflow a small screen. Fortunately, Bootstrap has extensive support for making nice-looking forms that work well on mobile devices.

Change the form in `views/noteedit.hbs` to this:

```
<form method='POST' action='/notes/save'>
  <div class="container-fluid">
    {{#if docreate}}
      <input type='hidden' name='docreate' value="create">
    {{else}}
      <input type='hidden' name='docreate' value="update">
    {{/if}}
    <div class="form-group row align-items-center">
      <label for="notekey" class="col-1 col-form-label">Key</label>
      {{#if docreate }}
        <div class="col">
          <input type='text' class="form-control"
            placeholder="note key" name='notekey' value=''/>
        </div>
      {{else}}
        {{#if note }}
          <span class="input-group-text">{{notekey}}</span>
        {{/if}}
        <input type='hidden' name='notekey'
          value='{{#if note }}{{notekey}}{{/if}} '>
      {{/if}}
    </div>

    <div class="form-group row">
      <label for="title" class="col-1 col-form-label">Title</label>
      <div class="col">
        <input type="text" class="form-control"
          id='title' name='title' placeholder="note title"
          value='{{#if note }}{{note.title}}{{/if}}'>
      </div>
```

```
</div>

<div class="form-group row">
<textarea class="form-control" name='body'
rows="5">{{#if note }}{{note.body}} {{/if}}</textarea>
</div>
<button type="submit" class="btn btn-default">Submit</button>
</div>
</form>
```

There's a lot going on here. What we've done is reorganize the `form` so Bootstrap can do the right things with it. The first thing to note is that we have several instances of this:

```
<div class="form-group row"> .. </div>
```

The entire form is contained within a `container-fluid`, meaning that it will automatically stretch to fit the screen. The form has three of these rows with the `form-group` class.

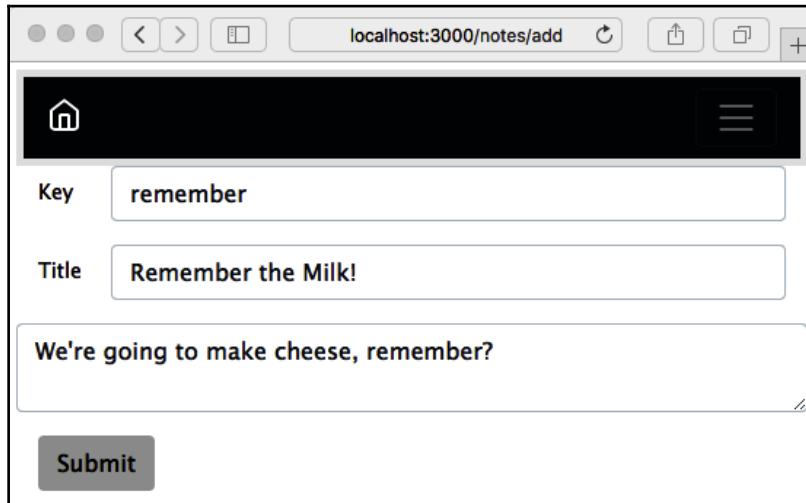
Bootstrap uses `form-group` elements to add structure to forms and to encourage proper use of `<label>` elements, along with other form elements. It's good practice to use a `<label>` element with every `<input>` element to improve assistive behavior in the browser, rather than simply leaving some dangling text.

For horizontal layout, notice that for each `row` there is a `<label>` with a `col-1` class, and the `<input>` element is contained within a `<div>` that has a `col` class. The effect is that the `<label>` has a controlled width and that the labels all have the same width, while the `<input>` elements take up the rest of the horizontal space.

Every form element has `class="form-control"`. Bootstrap uses this to identify the controls so it can add styling and behavior.

The `placeholder='key'` attribute puts sample text in an otherwise empty text input element. It disappears as soon as the user types something and is an excellent way to prompt the user with what's expected.

Finally, we changed the **Submit** button to be a Bootstrap button. These look nice, and Bootstrap makes sure that they work great:



The result looks good and works well on the iPhone. It automatically sizes itself to whatever screen it's on. Everything behaves nicely. In the preceding screenshot, we've resized the window small enough to cause the navbar to collapse. Clicking on the so-called hamburger icon on the right (the three horizontal lines) causes the navbar contents to pop up as a menu.

We have learned how to improve forms using Bootstrap. We have a similar task in the form to confirm deleting notes.

Cleaning up the delete-note window

The window used to verify the user's choice to delete a note doesn't look bad, but it can be improved.

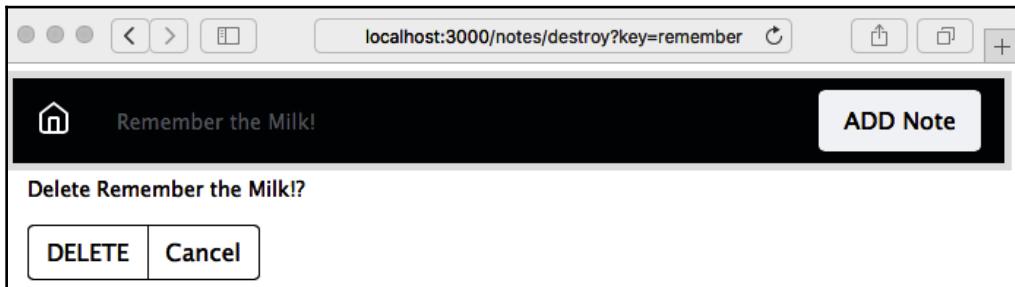
Edit `views/notedestroy.hbs` to contain the following:

```
<form method='POST' action='/notes/destroy/confirm'>
  <div class="container-fluid">
    <input type='hidden' name='notekey' value='
      {{#if note}}{{notekey}}{{/if}}'>
    <p class="form-text">Delete {{note.title}}?</p>
    <div class="btn-group">
      <button type="submit" value='DELETE'
        class="btn btn-outline-dark">DELETE</button>
      <a class="btn btn-outline-dark"
        href="/notes/view?key={{#if note}}{{notekey}}{{/if}}"
        role="button">
```

```
        Cancel</a>
    </div>
</div>
</form>
```

We've reworked it to use similar Bootstrap form markup. The question about deleting the note is wrapped with `class="form-text"` so that Bootstrap can display it properly.

The buttons are wrapped with `class="btn-group"` as before. The buttons have exactly the same styling as on other screens, giving a consistent look across the application:



There is an issue in that the title text in the navbar does not use the word `Delete`. In `routes/notes.mjs`, we can make this change:

```
// Ask to Delete note (destroy)
router.get('/destroy', async (req, res, next) => {
  var note = await notes.read(req.query.key);
  res.render('notedestroy', {
    title: note ? `Delete ${note.title}` : "",
    notekey: req.query.key, note: note
  });
});
```

What we've done is to change the `title` parameter passed to the template. We'd done this in the `/notes/edit` route handler and seemingly missed doing so in this handler.

That handles rewriting the `Notes` application to use Bootstrap. Having a complete Bootstrap-based UI, let's look at what it takes to customize the Bootstrap look and feel.

Customizing a Bootstrap build

One reason to use Bootstrap is that you can easily build a customized version. The primary reason to customize a Bootstrap build is to adjust the theme from the default. While we can use `stylesheet.css` to adjust the presentation, it's much more effective to adjust theming the Bootstrap way. That means changing the SASS variables and recompiling Bootstrap to generate a new `bootstrap.css` file.

Bootstrap stylesheets are built using the build process described in the `package.json` file. Therefore, customizing a Bootstrap build means first downloading the Bootstrap source tree, making modifications, then using the `npm run dist` command to build the distribution. By the end of this section, you'll know how to do all that.

The Bootstrap uses SASS, which is one of the CSS preprocessors used to simplify CSS development. In Bootstrap's code, one file (`scss/_variables.scss`) contains variables used throughout the rest of Bootstrap's `.scss` files. Change one variable and it automatically affects the rest of Bootstrap.



The official documentation on the Bootstrap website (<https://getbootstrap.com/docs/4.5/getting-started/build-tools/>) is useful for reference on the build process.

If you've followed the directions given earlier, you have a directory, `chap06/notes`, containing the *Notes* application source code. Create a directory named `chap06/notes/theme`, within which we'll set up a custom Bootstrap build process.

In order to have a clear record of the steps involved, we'll use a `package.json` file in that directory to automate the build process. There isn't any Node.js code involved; `npm` is also a convenient tool to automate the software build processes.

To start, we need a script for downloading the Bootstrap source tree from <https://github.com/twbs/bootstrap>. While the `bootstrap` npm package includes SASS source files, it isn't sufficient to build Bootstrap, and therefore we must download the source tree. What we do is navigate to the GitHub repository, click on the **Releases** tab, and select the URL for the most recent release. But instead of downloading it manually, let's automate the process.

With `theme/package.json` can contain this `scripts` section:

```
{  
  "scripts": {  
    "download": "wget -O -https://  
      github.com/twbs/bootstrap/archive/v4.5.0.tar.gz | tar xvfz -",  
    "postdownload": "cd bootstrap-4.5.0 && npm install"  
  }  
}
```

This will automatically download and unpack the Bootstrap source distribution, and then the `postdownload` step will run `npm install` to install the dependencies declared by the Bootstrap project. This gets the source tree all set up and ready to modify and build.

Type this command:

```
$ npm run download
```

This executes the steps to download and unpack the Bootstrap source tree. The scripts we gave will work for a Unix-like system, but if you are on Windows it will be easiest to run this in the Windows Subsystem for Linux.

This much only installs the tools necessary to build Bootstrap. The documentation on the Bootstrap website also discusses installing *Bundler* from the Ruby Gems repository, but that tool only seems to be required to bundle the built distribution. We do not need that tool, so skip that step.

To build Bootstrap, let's add the following lines to the `scripts` section in our `theme/package.json` file:

```
"scripts": {  
  ...  
  "clean": "rm -rf bootstrap-4.5.0",  
  "build": "cd bootstrap-4.5.0 && npm run dist",  
  "watch": "cd bootstrap-4.5.0 && npm run watch"  
  ...  
}
```

Obviously, you'll need to adjust these directory names when a new Bootstrap release is issued.

In the Bootstrap source tree, running `npm run dist` builds Bootstrap using a process recorded in the `Bootstrap package.json` file. Likewise, `npm run watch` sets up an automated process to scan for changed files and rebuilds Bootstrap upon changing any file. Running `npm run clean` will delete the Bootstrap source tree. By adding these lines to our `theme/package.json` file, we can start this in the Terminal and we can now rerun the build as needed without having to scratch our heads, struggling to remember what to do.

To avoid having the Bootstrap source code checked into your Git repository, add a `theme/.gitignore` file:

```
bootstrap-4.*
```

This will tell Git to not commit the Bootstrap source tree to the source repository. There's no need to commit third-party sources to your source tree since we have recorded in the `package.json` file the steps required to download the sources.

Now run a build with this command:

```
$ npm run build
```

The built files land in the `theme/bootstrap-4.5.0/dist` directory. The content of that directory will match the contents of the `npm` package for Bootstrap.

Before proceeding, let's take a look around the Bootstrap source tree. The `scss` directory contains the SASS source that will be compiled into the Bootstrap CSS files. To generate a customized Bootstrap build will require a few modifications in that directory.

The `bootstrap-4.5.0/scss/bootstrap.scss` file contains `@import` directives to pull in all Bootstrap components. The `file bootstrap-4.5.0/scss/_variables.scss` contains definitions used in the remainder of the Bootstrap SASS source. Editing or overriding these values will change the look of websites using the resulting Bootstrap build.

For example, these definitions determine the main color values:

```
$white: #fff !default;  
$gray-100: #f8f9fa !default;  
...  
$gray-800: #343a40 !default;  
...  
$blue: #007bff !default;  
...  
$red: #dc3545 !default;
```

```
$orange: #fd7e14 !default;
$yellow: #ffc107 !default;
$green: #28a745 !default;
...
$primary: $blue !default;
$secondary: $gray-600 !default;
$success: $green !default;
$info: $cyan !default;
$warning: $yellow !default;
$danger: $red !default;
$light: $gray-100 !default;
$dark: $gray-800 !default;
```

These are similar to normal CSS statements. The `!default` attribute designates these values as the default. Any `!default` values can be overridden without editing `_variables.scss`.

To create a custom theme we could change `_variables.scss`, then rerun the build. But what if Bootstrap makes a considerable change to `_variables.scss` that we miss? It's better to instead create a second file that overrides values in `_variables.scss`.

With that in mind, create a file, `theme/_custom.scss`, containing the following:

```
$white: #fff !default;
$gray-900: #212529 !default;
$body-bg: $gray-900 !default;
$body-color: $white !default;
```

This reverses the values for the `$body-bg` and `$body-color` settings in `_variables.scss`. The Notes app will now use white text on a dark background, rather than the default white background with dark text. Because these declarations do not use `!default`, they'll override the values in `_variables.scss`.

Then, make a copy of `scss/bootstrap.scss` in the `theme` directory and modify it like so:

```
@import "custom";
@import "functions";
@import "variables";
...
```

This adds an `@import` header for the `_custom.scss` file we just created. That way, Bootstrap will load our definitions during the build process.

Finally, add this line to the `scripts` section of `theme/package.json`:

```
"prebuild": "cp _custom.scss bootstrap.scss bootstrap-4.5.0/scss",
"postbuild": "mkdir -p dist && cp -r bootstrap-4.5.0/dist .",
```

With these scripts, before building Bootstrap, these two files will be copied in place, and afterward, the built files will be copied to a directory named `dist`. The `prebuild` step lets us commit our copy of the `_custom.scss` and `bootstrap.scss` files into our source repository, while being free to delete the Bootstrap source at any time. Likewise, the `postbuild` step lets us commit the custom theme we built to the source repository.

Next, rebuild Bootstrap:

```
$ npm run build

> @ prebuild /Users/David/chap06/notes/theme
> cp _custom.scss bootstrap.scss bootstrap-4.5.0/scss

> @ build /Users/David/chap06/notes/theme
> cd bootstrap-4.5.0 && npm run dist
...
...
```

While that's building, let's modify `notes/app.mjs` to mount the build directory:

```
// app.use('/assets/vendor/bootstrap', express.static(
// path.join(__dirname, 'node_modules', 'bootstrap', 'dist')));
app.use('/assets/vendor/bootstrap', express.static(
  path.join(__dirname, 'theme', 'dist')));
```

What we've done is switch from the Bootstrap configuration in `node_modules` to what we just built in the `theme` directory.

Then reload the application, and you'll see the coloring change.

There are two changes required to get this exact presentation. The button elements we used earlier have the `btn-outline-dark` class, which works well on a light background. Because the background is now dark, these buttons need to use light coloring.

To change the buttons, in `views/index.hbs`, make this change:

```
<a class="btn btn-lg btn-block btn-outline-light"
  href="/notes/view?key={{ key }}"> {{ title }} </a>
```

Make a similar change in `views/noteview.hbs`:

```
<a class="btn btn-outline-light" href="/notes/destroy?key={{notekey}}"  
    role="button"> Delete </a>  
<a class="btn btn-outline-light" href="/notes/edit?key={{notekey}}"  
    role="button"> Edit </a>
```

That's cool, we can now rework the Bootstrap color scheme any way we want. Don't show this to your user experience team, because they'll throw a fit. We did this to prove the point that we can edit `_custom.scss` and change the Bootstrap theme.

The next thing to explore is using a pre-built, third-party Bootstrap theme.

Using third-party custom Bootstrap themes

If all this is too complicated for you, several websites provide pre-built Bootstrap themes, or else simplified tools to generate a Bootstrap build. To get our feet wet, let's download a theme from Bootswatch (<https://bootswatch.com/>). This is both a collection of free and open source themes and a build system for generating custom Bootstrap themes (<https://github.com/thomasspark/bootswatch/>).

Let's use the **Minty** theme from Bootswatch to explore the needed changes. You can download the theme from the website or add the following to the `scripts` section of `package.json`:

```
"dl-minty": "mkdir -p minty && npm run dl-minty-css && npm run dl-  
minty-min-css",  
"dl-minty-css": "wget https://bootswatch.com/4/minty/bootstrap.css -O  
minty/bootstrap.css",  
"dl-minty-min-css": "wget  
https://bootswatch.com/4/minty/bootstrap.min.css -O  
minty/bootstrap.min.css"
```

This will download the prebuilt CSS files for our chosen theme. In passing, notice that the Bootswatch website offers `_variables.scss` and `_bootswatch.scss` files, which should be usable with a workflow similar to what we implemented in the previous section. The GitHub repository matching the Bootswatch website has a complete build procedure for building custom themes.

Perform the download with the following command:

```
$ npm run dl-minty

> notes@0.0.0 dl-minty /Users/David/chap06/notes
> mkdir -p minty && npm run dl-minty-css && npm run dl-minty-min-css

> notes@0.0.0 dl-minty-css /Users/David/chap06/notes
> wget https://bootswatch.com/4/minty/bootstrap.css -O
minty/bootstrap.css

> notes@0.0.0 dl-minty-min-css /Users/David/chap06/notes
> wget https://bootswatch.com/4/minty/bootstrap.min.css -O
minty/bootstrap.min.css
```

In app.mjs we will need to change the Bootstrap mounts to separately mount the JavaScript and CSS files. Use the following:

```
// app.use('/assets/vendor/bootstrap', express.static(
// path.join(__dirname, 'node_modules', 'bootstrap', 'dist')));
// app.use('/assets/vendor/bootstrap', express.static(
// path.join(__dirname, 'theme', 'bootstrap-4.0.0', 'dist')));
app.use('/assets/vendor/bootstrap/js', express.static(
  path.join(__dirname, 'node_modules', 'bootstrap', 'dist', 'js')));
app.use('/assets/vendor/bootstrap/css', express.static(
  path.join(__dirname, 'minty')));
```

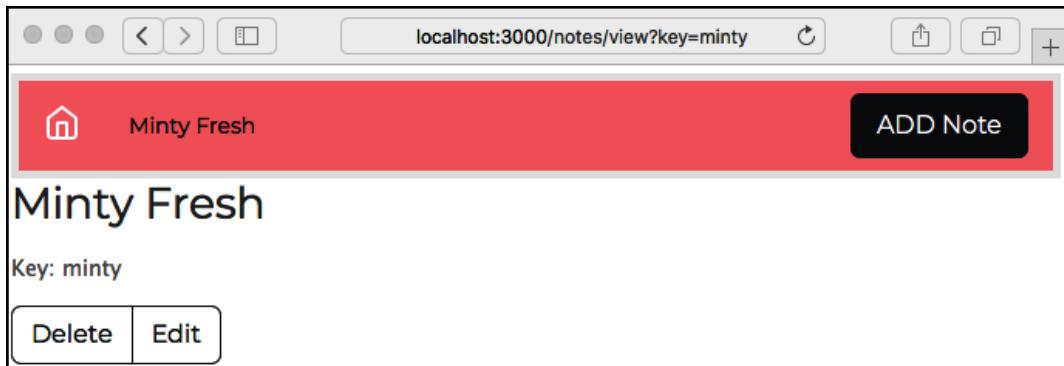
Instead of one mount for /vendor/bootstrap, we now have two mounts for each of the subdirectories. While the Bootswatch team provides bootstrap.css and bootstrap.min.css, they do not provide the JavaScript source. Therefore, we use the /vendor/bootstrap/css mount point to access the CSS files you downloaded from the theme provider, and the /vendor/bootstrap/js mount point to access the JavaScript files in the Bootstrap npm package.

Because Minty is a light-colored theme, the buttons now need to use the dark style. We had earlier changed the buttons to use a light style because of the dark background. We must now switch from btn-outline-light back to btn-outline-dark. In partials/header.hbs, the color scheme requires a change in the navbar content:

```
<div class="collapse navbar-collapse" id="navbarSupportedContent">
  <span class="navbar-text text-dark col">{{ title }}</span>
  <a class="nav-item nav-link btn btn-dark col-auto"
    href='/notes/add'>ADD Note</a>
</div>
```

We selected the `text-dark` and `btn-dark` classes to provide some contrast against the background.

Re-run the application and you'll see something like this:



With that, we have completed our exploration of customizing the look and feel of a Bootstrap-based application. We can now wrap up the chapter.

Summary

The possibilities for using Bootstrap are endless. While we covered a lot of material, we have only touched the surface, and we could have done much more with our *Notes* application. But since our focus in this book is not on the UI, but on the backend Node.js code, we purposely limited ourselves to making the application work acceptably well on mobile devices.

You learned what the Twitter Bootstrap framework can do by using it to implement a simple responsive website design. Even the little changes we made improved the way the *Notes* app looks and feels. We also created a customized Bootstrap theme, along with using a third-party theme, to explore how easy it is to make a Bootstrap build look unique.

Now, we want to get back to writing Node.js code. We left off Chapter 5, *Your First Express Application*, with the problem of persistence, where the *Notes* application could be stopped and restarted without losing our notes. In Chapter 7, *Data Storage and Retrieval*, we'll dive into using several database engines to store our data.

7

Data Storage and Retrieval

In the previous two chapters, we built a small and somewhat useful application for storing notes, and then made it work on mobile devices. While our application works reasonably well, it doesn't store these notes anywhere on a long-term basis, meaning the notes are lost when you stop the server, and if you run multiple instances of `Notes`, each instance has its own set of notes. Our next step is to introduce a database tier to persist the notes to long-term storage.

In this chapter, we will look at database support in Node.js, with the goal being to gain exposure to several kinds of databases. For the `Notes` application, the user should see the same set of notes for any `Notes` instance accessed, and the user should be able to reliably access notes at any time.

We'll start with the `Notes` application code used in the previous chapter. We started with a simple in-memory data model, using an array to store the notes, and then made it mobile-friendly. In this chapter, we will cover the following topics:

- The relationship between databases and asynchronous code
- Configuring the logging of operational and debugging information
- Catching important system errors
- Using `import()` to enable the runtime selection of the database to use
- Implementing data persistence for the `Notes` objects using several database engines
- Designing simple configuration files with YAML

The first step is to duplicate the code from the previous chapter. For instance, if you were working in `chap06/notes`, duplicate it and change its name to `chap07/notes`.

Let's start by reviewing a little theory on why database code in Node.js is asynchronous.

Let's get started!

Remembering that data storage requires asynchronous code

By definition, external data storage systems require asynchronous coding techniques, such as the ones we discussed in previous chapters. The core principle of the Node.js architecture is that any operation that requires a long time to perform must have an asynchronous API in order to keep the event loop running. The access time to retrieve data from a disk, another process, or a database always needs to take sufficient time to require deferred execution.

The existing `Notes` data model is an in-memory datastore. In theory, in-memory data access does not require asynchronous code and, therefore, the existing model module could use regular functions, rather than `async` functions.

We know that `Notes` should use databases and it requires an asynchronous API to access the `Notes` data. For this reason, the existing `Notes` model API uses `async` functions, so in this chapter, we can persist the `Notes` data to databases.

That was a useful refresher. Let's now talk about one of the administrative details required for a production application—using a logging system to store the usage data.

Logging and capturing uncaught errors

Before we get into databases, we have to address one of the attributes of a high-quality web application—managing logged information, including normal system activity, system errors, and debugging information. Logs give us an insight into the behavior of the system. They answer the following questions for the developers:

- How much traffic is the application getting?
- If it's a website, which pages are people hitting the most?
- How many errors occur and of what kind? Do attacks occur? Are malformed requests being sent?

Log management is also an issue. Unless managed well, log files can quickly fill the disk space. So, it becomes high priority to process old logs, hopefully extracting useful data before deleting the old logs. Commonly, this includes **log rotation**, which means regularly moving the existing log file to an archive directory and then starting with a fresh log file. Afterward, processing can occur to extract useful data, such as errors or usage trends. Just as your business analyst looks at profit/loss statements every few weeks, your DevOps team needs various reports to know whether there are enough servers to handle the traffic. Furthermore, log files can be screened for security vulnerabilities.

When we used the Express generator to initially create the `Notes` application, it configured an activity-logging system using `morgan` with the following code:

```
import { default as logger } from 'morgan';
..
app.use(logger('dev'));
```

This module is what prints messages about HTTP requests on the terminal window. We'll look at how to configure this in the next section.



Visit <https://github.com/expressjs/morgan> for more information about `morgan`.

Another useful type of logging is debugging messages about an application. Debugging traces should be silent in most cases; they should only print information when debugging is turned on, and the level of detail should be configurable.

The Express team uses the `debug` package for debugging logs. These are turned on using the `DEBUG` environment variable, which we've already seen in use. We will see how to configure this shortly and put it to use in the `Notes` application. For more information, refer to <https://www.npmjs.com/package/debug>.

Finally, the application might generate uncaught exceptions or unhandled Promises. The `uncaughtException` and `unhandledRejection` errors must be captured, logged, and dealt with appropriately. We do not use the word *must* lightly; these errors *must* be handled.

Let's get started.

Request logging with morgan

The `morgan` package generates log files from the HTTP traffic arriving on an Express application. It has two general areas for configuration:

- Log format
- Log location

As it stands, Notes uses the `dev` format, which is described as a concise status output for developers. This can be used to log web requests as a way to measure website activity and popularity. The Apache log format already has a large ecosystem of reporting tools and, sure enough, `morgan` can produce log files in this format.

To enable changing the logging format, simply change the following line in `app.mjs`:

```
app.use(logger(process.env.REQUEST_LOG_FORMAT || 'dev'));
```

This is the pattern we are following throughout this book; namely, to have a default value baked into the application and to use an environment variable to override the default. If we don't supply a configuration value through the environment variable, the program uses the `dev` format. Next, we need to run Notes, as follows:

```
$ REQUEST_LOG_FORMAT=common npm start
> notes@0.0.0 start /Users/david/chap07/notes
> cross-env node ./app.mjs
::1 -- [12/Jan/2020:05:51:21 +0000] "GET / HTTP/1.1" 304 -
::1 -- [12/Jan/2020:05:51:21 +0000] "GET
/vendor/bootstrap/css/bootstrap.min.css HTTP/1.1" 304 -
::1 -- [12/Jan/2020:05:51:21 +0000] "GET
/assets/stylesheets/style.css HTTP/1.1" 304 -
::1 -- [12/Jan/2020:05:51:21 +0000] "GET
/vendor/bootstrap/js/bootstrap.min.js HTTP/1.1" 304 -
```

To revert to the previous logging output, simply do not set this environment variable. If you've looked at Apache access logs, this logging format will look familiar. The `::1` notation at the beginning of the line is IPV6 notation for `localhost`, which you may be more familiar with as `127.0.0.1`.

Looking at the documentation for `morgan`, we learn that it has several predefined logging formats available. We've seen two of them—the `dev` format is meant to provide developer-friendly information, while the `common` format is compatible with the Apache log format. In addition to these predefined formats, we can create a custom log format by using various tokens.

We could declare victory on request logging and move on to debugging messages. However, let's look at logging directly to a file. While it's possible to capture `stdout` through a separate process, `morgan` is already installed on Notes and it provides the capability to direct its output to a file.

The `morgan` documentation suggests the following:

```
// create a write stream (in append mode)
const accessLogStream =
  fs.createWriteStream(`$__dirname__/access.log`, {flags: 'a'});
// setup the logger
app.use(morgan('combined', {stream: accessLogStream}));
```

However, this has a problem; it's impossible to perform log rotation without killing and restarting the server. The phrase *log rotation* refers to a DevOps practice of keeping log file snapshots, where each snapshot covers a few hours of activity. Typically, an application server will not keep a file handle continuously open to the log file, and the DevOps team can write a simple script that runs every few hours and uses the `mv` command to move log files around and the `rm` command to delete old files. Unfortunately, `morgan`, when configured as it is here, keeps a continuously open file handle to the log file.

Instead, we'll use the `rotating-file-stream` package. This package even automates the log rotation task so that the DevOps team doesn't have to write a script for that purpose.



For the documentation on this, refer to the package page at <https://www.npmjs.com/package/rotating-file-stream>.

First, install the package:

```
$ npm install rotating-file-stream --save
```

Then, add the following code to `app.mjs`:

```
import { default as rfs } from 'rotating-file-stream';
...
app.use(logger(process.env.REQUEST_LOG_FORMAT || 'dev', {
  stream: process.env.REQUEST_LOG_FILE ?
    rfs.createStream(process.env.REQUEST_LOG_FILE, {
      size: '10M',           // rotate every 10 MegaBytes written
      interval: '1d',        // rotate daily
      compress: 'gzip' // compress rotated files
    })
  }
});
```

```
    })
    : process.stdout
});
```

In the `import` section at the top, we're loading `rotating-file-stream` as `rfs`. If the `REQUEST_LOG_FILE` environment variable is set, we'll take that as the filename to record to. The `stream` argument to `morgan` simply takes a writable stream. If `REQUEST_LOG_FILE` is not set, we use a `?:` operator to supply the value of `process.stdout` as the writable stream. If it is set, then we use `rfs.createStream` to create a writable stream that handles log rotation through the `rotating-file-stream` module.

In `rfs.createStream`, the first argument is the filename of the log file and the second is an options object describing the behavior to use. Quite a comprehensive set of options are available for this. The configuration shown here rotates the log file when it reaches 10 megabytes in size (or after 1 day) and the rotated log file is compressed using the `gzip` algorithm.

It's possible to set up multiple logs. For example, if we wanted to log to the console, in addition to logging to the file, we could add the following `logger` declaration:

```
if (process.env.REQUEST_LOG_FILE) {
  app.use(logger(process.env.REQUEST_LOG_FORMAT || 'dev'));
}
```

If the `REQUEST_LOG_FILE` variable is set, the other logger will direct logging to the file. Then, because the variable is set, this logger will be created and will direct logging to the console. Otherwise, if the variable is not set, the other logger will send logging to the console and this logger will not be created.

We use these variables as before, specifying them on the command line, as follows:

```
$ REQUEST_LOG_FILE=log.txt REQUEST_LOG_FORMAT=common DEBUG=notes:*
node ./app.mjs
```

With this configuration, an Apache format log will be created in `log.txt`. After making a few requests, we can inspect the log:

```
$ ls -l log.txt
-rw-r--r-- 1 david admin 18831 Jan 17 21:10 log.txt
$ head log.txt
::1 - - [18/Jan/2020:04:32:02 +0000] "GET / HTTP/1.1" 304 -
::1 - - [18/Jan/2020:04:32:02 +0000] "GET
/assets/vendor/bootstrap/css/bootstrap.min.css HTTP/1.1" 304 -
::1 - - [18/Jan/2020:04:32:02 +0000] "GET
```

```
/assets/stylesheets/style.css HTTP/1.1" 304 -
::1 - - [18/Jan/2020:04:32:02 +0000] "GET
/assets/vendor/jquery/jquery.min.js HTTP/1.1" 304 -
```

As expected, our log file has entries in Apache format. Feel free to add one or both of these environment variables to the script in `package.json` as well.

We've seen how to make a log of the HTTP requests and how to robustly record it in a file. Let's now discuss how to handle debugging messages.

Debugging messages

How many of us debug our programs by inserting `console.log` statements? Most of us do. Yes, we're supposed to use a debugger, and yes, it is a pain to manage the `console.log` statements and make sure they're all turned off before committing our changes. The `debug` package provides a better way to handle debug tracing, which is quite powerful.



For the documentation on the `debug` package, refer to <https://www.npmjs.com/package/debug>.

The Express team uses `DEBUG` internally, and we can generate quite a detailed trace of what Express does by running `Notes` this way:

```
$ DEBUG=express:* npm start
```

This is pretty useful if you want to debug Express. However, we can use this in our own code as well. This works similarly to inserting `console.log` statements, but without having to remember to comment out the debugging code.

To use this in our code, add the following declaration to the top of any module where you want the debugging output:

```
import { default as DBG } from 'debug';
const debug = DBG('notes:debug');
const dbgerror = DBG('notes:error');
```

This creates two functions—`debug` and `dbgerror`—which will generate debugging traces if enabled. The `Debug` package calls functions *debuggers*. The debugger named `debug` has a `notes:debug` specifier, while `dbgerror` has a `notes:error` specifier. We'll talk in more detail about specifiers shortly.

Using these functions is as simple as this:

```
debug('some message');
..
debug(`got file ${fileName}`);
```

When debugging is enabled for the current module, this causes a message to be printed. If debugging is not enabled for the current module, no messages are printed. Again, this is similar to using `console.log`, but you can dynamically turn it on and off without modifying your code, simply by setting the `DEBUG` variable appropriately.

The `DEBUG` environment variable contains a specifier describing which code will have debugging enabled. The simplest specifier is `*`, which is a wildcard that turns on every debugger. Otherwise, debug specifiers use the `identifier:identifier` format. When we said to use `DEBUG=express:*`, the specifier used `express` as the first identifier and used the `*` wildcard for the second identifier.

By convention, the first identifier should be the name of your application or library. So, we used `notes:debug` and `notes:error` earlier as specifiers. However, that's just a convention; you can use any specifier format you like.

To add debugging to Notes, let's add a little more code. Add the following to the bottom of `app.mjs`:

```
server.on('request', (req, res) => {
  debug(`${new Date().toISOString()} request ${req.method}
  ${req.url}`);
});
```

This is adapted from the `httpsniffer.mjs` example from Chapter 4, *HTTP Servers and Clients*, and for every HTTP request, a little bit of information will be printed.

Then, in `appsupport.mjs`, let's make two changes. Add the following to the top of the `onError` function:

```
export function onError(error) {
  dbgerror(error);
  ..
}
```

This will output an error trace on any errors captured by Express.

Then, change `onListening` to the following:

```
export function onListening() {
  const addr = server.address();
  const bind = typeof addr === 'string'
    ? 'pipe ' + addr
    : 'port ' + addr.port;
  debug(`Listening on ${bind}`);
}
```

This changes the `console.log` call to a `debug` call so that a `Listening on` message is printed only if debugging is enabled.

If we run the application with the `DEBUG` variable set appropriately, we get the following output:

```
$ REQUEST_LOG_FORMAT=common DEBUG=notes:* node ./app.mjs
notes:debug Listening on port 3000 +0ms
notes:debug 2020-01-18T05:48:27.960Z request GET /notes/add +0ms
::1 - - [18/Jan/2020:05:48:28 +0000] "GET /notes/add HTTP/1.1" 304 -
notes:debug 2020-01-18T05:48:28.143Z request GET
/assets/vendor/bootstrap/css/bootstrap.min.css +183ms
::1 - - [18/Jan/2020:05:48:28 +0000] "GET
/assets/vendor/bootstrap/css/bootstrap.min.css HTTP/1.1" 304 -
...
...
```

Look at this carefully and you'll see that the output is both the logging output from `morgan` and the debugging output from the `debug` module. The debugging output, in this case, starts with `notes:debug`. The logging output is, because of the `REQUEST_LOG_FORMAT` variable, in Apache format.

We now have a debug tracing system that's ready to be used. The next task to cover is seeing whether it's possible to capture this or other console output in a file.

Capturing `stdout` and `stderr`

Important messages can be printed to `process.stdout` or `process.stderr`, which can be lost if you don't capture the output. It is best practice to capture this output for future analysis because there can be useful debugging information contained in it. An even better practice is to use a system facility to capture these output streams.

A **system facility** can include a process manager application that launches applications while connecting the standard output and standard error streams to a file.

While it lacks this sort of facility, it turns out that JavaScript code running in Node.js can intercept the `process.stdout` and `process.stderr` streams. Among the available packages, let's look at `capture-console`. For a writable stream, this package will invoke a callback function that you provided for any output.



Refer to the `capture-console` package page for the relevant documentation at <https://www.npmjs.com/package/capture-console>.

The last administrative item to cover is ensuring we capture otherwise uncaught errors.

Capturing uncaught exceptions and unhandled rejected Promises

Uncaught exceptions and unhandled rejected Promises are other areas where important information can be lost. Since our code is supposed to capture all errors, anything that's uncaught is an error on our part. Important information might be missing from our failure analysis if we do not capture these errors.

Node.js indicates these conditions with events sent by the `process` object, `uncaughtException` and `unhandledRejection`. In the documentation for these events, the Node.js team sternly says that in either condition, the application is in an unknown state because something failed and that it may not be safe to keep the application running.

To implement these handlers, add the following to `appsupport.mjs`:

```
process.on('uncaughtException', function(err) {
  console.error(`I've crashed!!! - ${err.stack || err}`);
});

import * as util from 'util';
process.on('unhandledRejection', (reason, p) => {
  console.error(`Unhandled Rejection at: ${util.inspect(p)} reason:
${reason}`);
});
```

Because these are events that are emitted from the `process` object, the way to handle them is to attach an event listener to these events. That's what we've done here.

The names of these events describe their meaning well. An `uncaughtException` event means an error was thrown but was not caught by a `try/catch` construct. Similarly, an `UnhandledRejection` event means a Promise ended in a rejected state, but there was no `.catch` handler.

Our DevOps team will be happier now that we've handled these administrative chores. We've seen how to generate useful log files for HTTP requests, how to implement debug tracing, and even how to capture it to a file. We wrapped up this section by learning how to capture otherwise-uncaught errors.

We're now ready to move on to the real purpose of this chapter—storing notes in persistent storage, such as in a database. We'll implement support for several database systems, starting with a simple system using files on a disk.

Storing notes in a filesystem

Filesystems are an often-overlooked database engine. While filesystems don't have the sort of query features supported by database engines, they are still a reliable place to store files. The Notes schema is simple enough, so the filesystem can easily serve as its data storage layer.

Let's start by adding two functions to the `Note` class in `models/Notes.mjs`:

```
export default class Note {
  ...
  get JSON() {
    return JSON.stringify({
      key: this.key, title: this.title, body: this.body
    });
  }

  static fromJSON(json) {
    const data = JSON.parse(json);
    if (typeof data !== 'object'
      || !data.hasOwnProperty('key')
      || typeof data.key !== 'string'
      || !data.hasOwnProperty('title')
      || typeof data.title !== 'string'
      || !data.hasOwnProperty('body')
      || typeof data.body !== 'string') {
```

```
        throw new Error(`Not a Note: ${json}`);
    }
    const note = new Note(data.key, data.title, data.body);
    return note;
}
}
```

We'll use this to convert the `Note` objects into and from JSON-formatted text.

The `JSON` method is a getter, which means it retrieves the value of the object. In this case, the `note.JSON` attribute/getter (with no parentheses) will simply give us the JSON representation of the note. We'll use this later to write to JSON files.

`fromJSON` is a static function, or factory method, to aid in constructing the `Note` objects if we have a JSON string. Since we could be given anything, we need to test the input carefully. First, if the string is not in JSON format, `JSON.parse` will fail and throw an exception. Secondly, we have what the TypeScript community calls a **type guard**, or an `if` statement, to test whether the object matches what is required of a `Note` object. This checks whether it is an object with the `key`, `title`, and `body` fields, all of which must be strings. If the object passes these tests, we use the data to construct a `Note` instance.

These two functions can be used as follows:

```
const note = new Note("key", "title", "body");
const json = note.JSON;      // produces JSON text
const newnote = Note.fromJSON(json); // produces new Note instance
```

This example code snippet produces a simple `Note` instance and then generates the JSON version of the note. Then, a new note is instantiated from that JSON string using `from JSON()`.

Now, let's create a new module, `models/notes-fs.mjs`, to implement the filesystem datastore:

```
import fs from 'fs-extra';
import path from 'path';
import util from 'util';
import { approotdir } from '../approotdir.mjs';
import { Note, AbstractNotesStore } from './Notes.mjs';
import { default as DBG } from 'debug';
const debug = DBG('notes:notes-fs');
const error = DBG('notes:error-fs');
```

This imports the required modules; one addition is the use of the `fs-extra` module. This module is used because it implements the same API as the core `fs` module while adding a few useful additional functions. In our case, we are interested in `fs.ensureDir`, which verifies whether the named directory structure exists and if not, a directory path is created. If we did not need `fs.ensureDir`, we would simply use `fs.promises` since it, too, supplies filesystem functions that are useful in `async` functions.



For the documentation on `fs-extra`, refer to <https://www.npmjs.com/package/fs-extra>.

Now, add the following to `models/notes-fs.mjs`:

```
export default class FSNotesStore extends AbstractNotesStore {

    async close() { }
    async update(key, title, body) { return crupdate(key, title, body); }
    async create(key, title, body) { return crupdate(key, title, body); }

    async read(key) {
        const notesdir = await notesDir();
        const thenote = await readJSON(notesdir, key);
        return thenote;
    }

    async destroy(key) {
        const notesdir = await notesDir();
        await fs.unlink(filePath(notesdir, key));
    }

    async keylist() {
        const notesdir = await notesDir();
        let filez = await fs.readdir(notesdir);
        if (!filez || typeof filez === 'undefined') filez = [];
        const thenotes = filez.map(async fname => {
            const key = path.basename(fname, '.json');
            const thenote = await readJSON(notesdir, key);
            return thenote.key;
        });
        return Promise.all(thenotes);
    }
}
```

```
    async count() {
      const notesdir = await notesDir();
      const filez = await fs.readdir(notesdir);
      return filez.length;
    }
}
```

The `FSNotesStore` class is an implementation of `AbstractNotesStore`, with a focus on storing the `Note` instances as JSON in a directory. These methods implement the API that we defined in [Chapter 5, Your First Express Application](#). This implementation is incomplete since a couple of helper functions still need to be written, but you can see that it relies on files in the filesystem. For example, the `destroy` method simply uses `fs.unlink` to delete the note from the disk. In `keylist`, we use `fs.readdir` to read each `Note` object and construct an array of keys for the notes.

Let's add the helper functions:

```
async function notesDir() {
  const dir = process.env.NOTES_FS_DIR
    || path.join(approotdir, 'notes-fs-data');
  await fs.ensureDir(dir);
  return dir;
}

const filePath = (notesdir, key) => path.join(notesdir,
` ${key}.json`);

async function readJSON(notesdir, key) {
  const readFrom = filePath(notesdir, key);
  const data = await fs.readFile(readFrom, 'utf8');
  return Note.fromJSON(data);
}

async function crudate(key, title, body) {
  const notesdir = await notesDir();
  if (key.indexOf('/') >= 0) {
    throw new Error(`key ${key} cannot contain '/'`);
  }
  const note = new Note(key, title, body);
  const writeTo = filePath(notesdir, key);
  const writeJSON = note.JSON;
  await fs.writeFile(writeTo, writeJSON, 'utf8');
  return note;
}
```

The `crupdate` function is used to support both the `update` and `create` methods. For this `Notes` store, both of these methods are the same and write the content to the disk as a JSON file.

As the code is written, the notes are stored in a directory determined by the `notesDir` function. This directory is either specified in the `NOTES_FS_DIR` environment variable or in `notes-fs-data` within the `Notes` root directory (as learned from the `approotdir` variable). Either way, `fs.ensureDir` is used to make sure that the directory exists.

The pathname for `Notes` is calculated by the `filePath` function.

Because the pathname is `${notesDir} / ${key} .json`, the key cannot use characters that cannot be used in filenames. For that reason, `crupdate` throws an error if the key contains a `/` character.

The `readJSON` function does what its name suggests—it reads a `Note` object as a JSON file from the disk.

We're also adding another dependency:

```
$ npm install fs-extra --save
```

We're now almost ready to run the `Notes` application, but there's an issue that first needs to be resolved with the `import()` function.

Dynamically importing ES6 modules

Before we start modifying the router functions, we have to consider how to account for multiple `AbstractNotesStore` implementations. By the end of this chapter, we will have several of them, and we want an easy way to configure `Notes` to use any of them. For example, an environment variable, `NOTES_MODEL`, could be used to specify the `Notes` data model to use, and the `Notes` application would dynamically load the correct module.

In `Notes`, we refer to the `Notes` datastore module from several places. To change from one datastore to another requires changing the source in each of these places. It would be better to locate that selection in one place, and further, to make it dynamically configurable at runtime.

There are several possible ways to do this. For example, in a CommonJS module, it's possible to compute the pathname to the module for a `require` statement. It would consult the environment variable, `NOTES_MODEL`, to calculate the pathname for the datastore module, as follows:

```
const notesStore = require(`../models/notes-
${process.env.NOTES_MODEL}.js`);
```

However, our intent is to use ES6 modules, and so let's see how this works within that context. Because in the regular `import` statement the module name cannot be an expression like this, we need to load modules using `dynamic import`. The `dynamic import` feature—the `import()` function, in other words—does allow us to dynamically compute a module name to load.

To implement this idea, let's create a new file, `models/notes-store.mjs`, containing the following:

```
import { default as DBG } from 'debug';
const debug = DBG('notes:notes-store');
const error = DBG('notes:error-store');

var _NotesStore;

export async function useModel(model) {
    try {
        let NotesStoreModule = await import(`./notes-${model}.mjs`);
        let NotesStoreClass = NotesStoreModule.default;
        _NotesStore = new NotesStoreClass();
        return _NotesStore;
    } catch (err) {
        throw new Error(`No recognized NotesStore in ${model} because
            ${err}`);
    }
}

export { _NotesStore as NotesStore };
```

This is what we might call a factory function. It uses `import()` to load a module whose filename is calculated from the `model` parameter. We saw in `notes-fs.mjs` that the `FSTNotesStore` class is the default export. Therefore, the `NotesStoreClass` variable gets that class, then we call the constructor to create an instance, and then we stash that instance in a global scope variable. That global scope variable is then exported as `NotesStore`.

We need to make one small change in `models/notes-memory.mjs`:

```
export default class InMemoryNotesStore extends AbstractNotesStore {  
  ... }
```

Any module implementing `AbstractNotesStore` will export the defined class as the default export.

In `app.mjs`, we need to make another change to call this `useModel` function. In Chapter 5, *Your First Express Application*, we had `app.mjs` import `models/notes-memory.mjs` and then set up `NotesStore` to contain an instance of `InMemoryNotesStore`. Specifically, we had the following:

```
import { InMemoryNotesStore } from './models/notes-memory.mjs';  
export const NotesStore = new InMemoryNotesStore();
```

We need to remove these two lines of code from `app.mjs` and then add the following:

```
import { useModel as useNotesModel } from './models/notes-store.mjs';  
useNotesModel(process.env.NOTES_MODEL ? process.env.NOTES_MODEL :  
  "memory")  
.then(store => {})  
.catch(error => { onError({ code: 'ENOTESSTORE', error }); });
```

We are importing `useModel`, renaming it `useNotesModel`, and then calling it by passing in the `NOTES_MODEL` environment variable. In case the `NOTES_MODEL` variable is not set, we'll default to the "memory" `NotesStore`. Since `useNotesModel` is an `async` function, we need to handle the resulting Promise. `.then` handles the success case, but there is nothing to do, so we supply an empty function. What's important is that any errors will shut down the application, so we have added `.catch`, which calls `onError` to do so.

To support this error indicator, we need to add the following to the `onError` function in `appsupport.mjs`:

```
case 'ENOTESSTORE':  
  console.error(`Notes data store initialization failure because `,  
    error.error);  
  process.exit(1);  
  break;
```

This added error handler will also cause the application to exit.

These changes also require us to make another change. The `NotesStore` variable is no longer in `app.mjs`, but is instead in `models/notes-store.mjs`. This means we need to go to `routes/index.mjs` and `routes/notes.mjs`, where we make the following change to the imports:

```
import { default as express } from 'express';
import { NotesStore as notes } from '../models/notes-store.mjs';
export const router = express.Router();
```

We are importing the `NotesStore` export from `notes-store.mjs`, renaming it `notes`. Therefore, in both of the router modules, we will make calls such as `notes.keylist()` to access the dynamically selected `AbstractNotesStore` instance.

This layer of abstraction gives the desired result—setting an environment variable that lets us decide at runtime which datastore to use.

Now that we have all the pieces, let's run the `Notes` application and see how it behaves.

Running the Notes application with filesystem storage

In `package.json`, add the following to the `scripts` section:

```
"fs-start": "cross-env DEBUG=notes:* PORT=3000 NOTES_MODEL=fs node ./app.mjs",
"fs-server1": "cross-env NOTES_MODEL=fs PORT=3001 node ./app.mjs",
"fs-server2": "cross-env NOTES_MODEL=fs PORT=3002 node ./app.mjs",
```



When you add these entries to `package.json`, make sure you use the correct JSON syntax. In particular, if you leave a comma at the end of the `scripts` section, it will fail to parse and `npm` will throw an error message.

With this code in place, we can now run the `Notes` application, as follows:

```
$ DEBUG=notes:* npm run fs-start
> notes@0.0.0 fs-start /Users/david/chap07/notes
> cross-env DEBUG=notes:* PORT=3000 NOTES_MODEL=fs node ./app.mjs
notes:debug Listening on port 3000 +0ms
notes:notes-fs keylist dir /home/david/Chapter07/notes/notes-fs-data
files=[] +0ms
```

We can use the application at `http://localhost:3000` as before. Because we did not change any template or CSS files, the application will look exactly as you left it at the end of Chapter 6, *Implementing the Mobile-First Paradigm*.

Because debugging is turned on for `notes:*`, we'll see a log of whatever the Notes application is doing. It's easy to turn this off by simply not setting the `DEBUG` variable.

You can now kill and restart the Notes application and see the exact same notes. You can also edit the notes in the command line using regular text editors such as `vi`. You can now start multiple servers on different ports, using the `fs-server1` and `fs-server2` scripts, and see exactly the same notes.

As we did at the end of Chapter 5, *Your First Express Application*, we can start the two servers' separate command windows. This runs two instances of the application, each on different ports. Then, visit the two servers in separate browser windows, and you will see that both browser windows show the same notes.

Another thing to try is specifying `NOTES_FS_DIR` to define a different directory to store notes.

The final check is to create a note where the key has a / character. Remember that the key is used to generate the filename where we store the note, and so the key cannot contain a / character. With the browser open, click on **ADD Note** and enter a note, ensuring that you use a / character in the key field. On clicking the **Submit** button, you'll see an error saying that this isn't allowed.

We have now demonstrated adding persistent data storage to Notes. However, this storage mechanism isn't the best, and there are several other database types to explore. The next database service on our list is LevelDB.

Storing notes with the LevelDB datastore

To get started with actual databases, let's look at an extremely lightweight, small-footprint database engine: `level`. This is a Node.js-friendly wrapper that wraps around the LevelDB engine and was developed by Google. It is normally used in web browsers for local data persistence and is a non-indexed, NoSQL datastore originally designed for use in browsers. The Level Node.js module uses the LevelDB API and supports multiple backends, including `leveldown`, which integrates the C++ LevelDB database into Node.js.



Visit <https://www.npmjs.com/package/level> for information on this module.

To install the database engine, run the following command:

```
$ npm install level@6.x --save
```

This installs the version of `level` that the following code was written against.

Then, create the `models/notes-level.mjs` module, which will contain the `AbstractNotesStore` implementation:

```
import util from 'util';
import { Note, AbstractNotesStore } from './Notes.mjs';
import level from 'level';
import { default as DBG } from 'debug';
const debug = DBG('notes:notes-level');
const error = DBG('notes:error-level');

let db;

async function connectDB() {
  if (typeof db !== 'undefined' || db) return db;
  db = await level(
    process.env.LEVELDB_LOCATION || 'notes.level',
    {
      createIfMissing: true,
      valueEncoding: "json"
    });
  return db;
}
```

We start the module with the `import` statements and a couple of declarations. The `connectDB` function is used for what the name suggests—to connect with a database. The `createIfMissing` option also does what it suggests, which is creating a database if there isn't one already one with the name that is used. The import from the module, `level`, is a constructor function that creates a `level` instance connected to the database specified by the first argument. This first argument is a location in the filesystem—a directory, in other words—where the database will be stored.

The `level` constructor returns a `db` object through which to interact with the database. We're storing `db` as a global variable in the module for ease of use. In `connectDB`, if the `db` object is set, we just return it immediately; otherwise, we open the database using the constructor, as just described.

The location of the database defaults to `notes.level` in the current directory. The `LEVELDB_LOCATION` environment variable can be set, as the name implies, to specify the database location.

Now, let's add the rest of this module:

```
export default class LevelNotesStore extends AbstractNotesStore {  
  
    async close() {  
        const _db = db;  
        db = undefined;  
        return _db ? _db.close() : undefined;  
    }  
  
    async update(key, title, body) { return crupdate(key, title, body);  
    }  
    async create(key, title, body) { return crupdate(key, title, body);  
    }  
  
    async read(key) {  
        const db = await connectDB();  
        const note = Note.fromJSON(await db.get(key));  
        return note;  
    }  
  
    async destroy(key) {  
        const db = await connectDB();  
        await db.del(key);  
    }  
  
    async keylist() {  
        const db = await connectDB();  
        const keyz = [];  
        await new Promise((resolve, reject) => {  
            db.createKeyStream()  
            .on('data', data => keyz.push(data))  
            .on('error', err => reject(err))  
            .on('end', () => resolve(keyz));  
        });  
        return keyz;  
    }  
  
    async count() {  
        const db = await connectDB();  
        var total = 0;  
        await new Promise((resolve, reject) => {  
            db.createKeyStream()  
            .on('data', data => total++)  
        });  
        return total;  
    }  
}
```

```
        .on('error', err => reject(err))
        .on('end', () => resolve(total));
    });
    return total;
}
}

async function crudupdate(key, title, body) {
    const db = await connectDB();
    const note = new Note(key, title, body);
    await db.put(key, note.JSON);
    return note;
}
```

As expected, we're creating a `LevelNotesStore` class to hold the functions.

In this case, we have code in the `close` function that calls `db.close` to close down the connection. The `level` documentation suggests that it is important to close the connection, so we'll have to add something to `app.mjs` to ensure that the database closes when the server shuts down. The documentation also says that `level` does not support concurrent connections to the same database from multiple clients, meaning if we want multiple `Notes` instances to use the database, we should only have the connection open when necessary.

Once again, there is no difference between the `create` and `update` operations, and so we use a `crudupdate` function again. Notice that the pattern in all the functions is to first call `connectDB` to get `db`, and then to call a function on the `db` object. In this case, we use `db.put` to store the `Note` object in the database.

In the `read` function, `db.get` is used to read the note. Since the `Note` data was stored as JSON, we use `Note.fromJSON` to decode and instantiate the `Note` instance.

The `destroy` function deletes a record from the database using the `db.del` function.

Both `keylist` and `count` use the `createKeyStream` function. This function uses an event-oriented interface to stream through every database entry, emitting events as it goes. A `data` event is emitted for each key in the database, while the `end` event is emitted at the end of the database, and the `error` event is emitted on errors. Since there is no simple way to present this as a simple `async` function, we have wrapped it with a `Promise` so that we can use `await`. We then invoke `createKeyStream`, letting it run its course and collect data as it goes. For `keylist`, in the `data` events, we add the data (in this case, the key to a database entry) to an array.

For count, we use a similar process, and in this case, we simply increment a counter. Since we have this wrapped in a Promise, in an error event, we call reject, and in an end event, we call resolve.

Then, we add the following to package.json in the scripts section:

```
"level-start": "cross-env DEBUG=notes:* PORT=3000 NOTES_MODEL=level  
node ./app.mjs",
```

Finally, you can run the Notes application:

```
$ npm run level-start  
> notes@0.0.0 start /Users/david/chap07/notes  
> cross-env DEBUG=notes:* PORT=3000 NOTES_MODEL=level node ./app.mjs  
notes:server Listening on port 3000 +0ms
```

The printout in the console will be the same, and the application will also look the same. You can put it through its paces to check whether everything works correctly.

Since level does not support simultaneous access to a database from multiple instances, you won't be able to use the multiple Notes application scenario. You will, however, be able to stop and restart the application whenever you want to without losing any notes.

Before we move on to looking at the next database, let's deal with a issue mentioned earlier—closing the database connection when the process exits.

Closing database connections when closing the process

The level documentation says that we should close the database connection with db.close. Other database servers may well have the same requirement. Therefore, we should make sure we close the database connection before the process exits, and perhaps also on other conditions.

Node.js provides a mechanism to catch signals sent by the operating system. What we'll do is configure listeners for these events, then close NotesStore in response.

Add the following code to `appsupport.mjs`:

```
import { NotesStore } from './models/notes-store.mjs';

async function catchProcessDeath() {
  debug('urk...');
  await NotesStore.close();
  await server.close();
  process.exit(0);
}

process.on('SIGTERM', catchProcessDeath);
process.on('SIGINT', catchProcessDeath);
process.on('SIGHUP', catchProcessDeath);

process.on('exit', () => { debug('exiting...'); });
```

We import `NotesStore` so that we can call its methods, and `server` was already imported elsewhere.

The first three `process.on` calls listen to operating system signals. If you're familiar with Unix process signals, these terms will be familiar. In each case, the event calls the `catchProcessDeath` function, which then calls the `close` function on `NotesStore` and, for good measure, on `server`.

Then, to have a measure of confirmation, we attached an `exit` listener so that we can print a message when the process is exiting. The Node.js documentation says that the `exit` listeners are prohibited from doing anything that requires further event processing, so we cannot close database connections in this handler.

Let's try it out by running the `Notes` application and then immediately pressing `Ctrl + C`:

```
$ npm run level-start
> notes@0.0.0 level-start /home/david/Chapter07/notes
> cross-env DEBUG=notes:* PORT=3000 NOTES_MODEL=level node ./app.mjs
notes:debug Listening on port 3000 +0ms
^C  notes:debug urk... +1s
    notes:debug exiting... +3s
```

Sure enough, upon pressing `Ctrl + C`, the `exit` and `catchProcessDeath` listeners are called.

That covers the `level` database, and we also have the beginning of a handler to gracefully shut down the application. The next database to cover is an embedded SQL database that requires no server processes.

Storing notes in SQL with SQLite3

To get started with more normal databases, let's see how we can use SQL from Node.js. First, we'll use SQLite3, which is a lightweight, simple-to-set-up database engine eminently suitable for many applications.



To learn more about this database engine, visit

<http://www.sqlite.org/>.

To learn more about the Node.js module, visit

<https://github.com/mapbox/node-sqlite3/wiki/API> or

<https://www.npmjs.com/package/sqlite3>.

The primary advantage of SQLite3 is that it doesn't require a server; it is a self-contained, no-set-up-required SQL database. The SQLite3 team also claims that it is very fast and that large, high-throughput applications have been built with it. The downside to the SQLite3 package is that its API requires callbacks, so we'll have to use the Promise wrapper pattern.

The first step is to install the module:

```
$ npm install sqlite3@5.x --save
```

This, of course, installs the `sqlite3` package.

To manage a SQLite3 database, you'll also need to install the SQLite3 command-line tools. The project website has precompiled binaries for most operating systems. You'll also find the tools available in most package management systems.

One management task that we can use is setting up the database tables, as we will see in the next section.

The SQLite3 database schema

Next, we need to make sure that our database is configured with a database table suitable for the Notes application. This is an example database administrator task, as mentioned at the end of the previous section. To do this, we'll use the `sqlite3` command-line tool. The `sqlite3.org` website has precompiled binaries, or the tool can be installed through your operating system's package management system—for example, you can use `apt-get` on Ubuntu/Debian and MacPorts on macOS.

For Windows, make sure you have installed the Chocolatey package manager tool from <https://chocolatey.org>. Then start a PowerShell with Administrator privileges, and run "choco install sqlite". That installs the SQLite3 DLL's and its command-line tools, letting you run the following instructions.

We're going to use the following SQL table definition for the schema (save it as `models/schema-sqlite3.sql`):

```
CREATE TABLE IF NOT EXISTS notes (
    notekey VARCHAR(255),
    title VARCHAR(255),
    body TEXT
);
```

To initialize the database table, we run the following command:

```
$ sqlite3 chap07.sqlite3
SQLite version 3.30.1 2019-10-10 20:19:45
Enter ".help" for usage hints.
sqlite> CREATE TABLE IF NOT EXISTS notes (
...>     notekey VARCHAR(255),
...>     title VARCHAR(255),
...>     body TEXT
...> );
sqlite> .schema notes
CREATE TABLE notes (
    notekey VARCHAR(255),
    title VARCHAR(255),
    body TEXT
);
sqlite> ^D
$ ls -l chap07.sqlite3
-rwx----- 1 david staff 8192 Jan 14 20:40 chap07.sqlite3
```

While we can do this, however, the best practice is to automate all the administrative processes. To that end, we should instead write a little bit of script to initialize the database.

Fortunately, the `sqlite3` command offers us a way to do this. Add the following to the `scripts` section of `package.json`:

```
"sqlite3-setup": "sqlite3 chap07.sqlite3 --init models/schema-sqlite3.sql",
```

Run the setup script:

```
$ npm run sqlite3-setup

> notes@0.0.0 sqlite3-setup /home/david/Chapter07/notes
> sqlite3 chap07.sqlite3 --init models/schema-sqlite3.sql

-- Loading resources from models/schema-sqlite3.sql
SQLite version 3.30.1 2019-10-10 20:19:45
Enter ".help" for usage hints.
sqlite> .schema notes
CREATE TABLE notes (
    notekey VARCHAR(255),
    title   VARCHAR(255),
    body    TEXT
);
sqlite> ^D
```

This isn't fully automated since we have to press *Ctrl + D* at the `sqlite` prompt, but at least we don't have to use our precious brain cells to remember how to do this. We could have easily written a small `Node.js` script to do this; however, by using the tools provided by the package, we have less code to maintain in our own project.

With the database table set up, let's move on to the code to interface with SQLite3.

The SQLite3 model code

We are now ready to implement an `AbstractNotesStore` implementation for SQLite3.

Create the `models/notes-sqlite3.mjs` file:

```
import util from 'util';
import { Note, AbstractNotesStore } from './Notes.mjs';
import { default as sqlite3 } from 'sqlite3';
import { default as DBG } from 'debug';
const debug = DBG('notes:notes-sqlite3');
const error = DBG('notes:error-sqlite3');

let db;

async function connectDB() {
  if (db) return db;
  const dbfile = process.env.SQLITE_FILE || "notes.sqlite3";
  await new Promise((resolve, reject) => {
    db = new sqlite3.Database(dbfile,
```

```
        sqlite3.OPEN_READWRITE | sqlite3.OPEN_CREATE,
        err => {
            if (err) return reject(err);
            resolve(db);
        });
    );
    return db;
}
```

This imports the required packages and makes the required declarations.

The `connectDB` function has a similar purpose to the one in `notes-level.mjs`: to manage the database connection. If the database is not open, it'll go ahead and open it, and it will even make sure that the database file is created (if it doesn't exist). If the database is already open, it'll simply be returned.

Since the API used in the `sqlite3` package requires callbacks, we will have to wrap every function call in a Promise wrapper, as shown here.

Now, add the following to `models/notes-sqlite3.mjs`:

```
export default class SQLITE3NotesStore extends AbstractNotesStore {
    // See implementation below
}
```

Since there are many member functions, let's talk about them individually:

```
async close() {
    const _db = db;
    db = undefined;
    return _db ?
        new Promise((resolve, reject) => {
            _db.close(err => {
                if (err) reject(err);
                else resolve();
            });
        });
    ) : undefined;
}
```

In `close`, the task is to close the database. There's a little dance done here to make sure the global `db` variable is unset while making sure we can close the database by saving `db` as `_db`. The `sqlite3` package will report errors from `db.close`, so we're making sure we report any errors:

```
async update(key, title, body) {
    const db = await connectDB();
    const note = new Note(key, title, body);
    await new Promise((resolve, reject) => {
```

```
db.run("UPDATE notes "+
    "SET title = ?, body = ? WHERE notekey = ?",
    [ title, body, key ], err => {
        if (err) return reject(err);
        resolve(note);
    });
});
return note;
}

async create(key, title, body) {
    const db = await connectDB();
    const note = new Note(key, title, body);
    await new Promise((resolve, reject) => {
        db.run("INSERT INTO notes ( notekey, title, body ) "+
            "VALUES ( ?, ?, ? );", [ key, title, body ], err => {
                if (err) return reject(err);
                resolve(note);
            });
    });
    return note;
}
```

We are now justified in defining to have separate `create` and `update` operations for the `Notes` model because the SQL statement for each function is different. The `create` function, of course, requires an `INSERT INTO` statement, while the `update` function, of course, requires an `UPDATE` statement.

The `db.run` function, which is used several times here, executes a SQL query while giving us the opportunity to insert parameters in the query string.

This follows a parameter substitution paradigm that's common in SQL programming interfaces. The programmer puts the SQL query in a string and then places a question mark anywhere that the aim is to insert a value in the query string. Each question mark in the query string has to match a value in the array provided by the programmer. The module takes care of encoding the values correctly so that the query string is properly formatted, while also preventing SQL injection attacks.

The `db.run` function simply runs the SQL query it is given and does not retrieve any data:

```
async read(key) {
    const db = await connectDB();
    const note = await new Promise((resolve, reject) => {
        db.get("SELECT * FROM notes WHERE notekey = ?",
            [ key ], (err, row) => {
```

```
        if (err) return reject(err);
        const note = new Note(row.notekey, row.title, row.body);
        resolve(note);
    });
});
return note;
}
```

To retrieve data using the `sqlite3` module, you use the `db.get`, `db.all`, or `db.each` functions. Since our `read` method only returns one item, we use the `db.get` function to retrieve just the first row of the result set. By contrast, the `db.all` function returns all of the rows of the result set at once, and the `db.each` function retrieves one row at a time, while still allowing the entire result set to be processed.

By the way, this `read` function has a bug in it—see whether you can spot the error. We'll read more about this in Chapter 13, *Unit Testing and Functional Testing*, when our testing efforts uncover the bug:

```
async destroy(key) {
    const db = await connectDB();
    return await new Promise((resolve, reject) => {
        db.run("DELETE FROM notes WHERE notekey = ?;", [ key ], err =>
    {
        if (err) return reject(err);
        resolve();
    });
});
}
```

In our `destroy` method, we simply use `db.run` to execute the `DELETE FROM` statement to delete the database entry for the associated note:

```
async keylist() {
    const db = await connectDB();
    const keyz = await new Promise((resolve, reject) => {
        const keyz = [];
        db.all("SELECT notekey FROM notes", (err, rows) => {
            if (err) return reject(err);
            resolve(rows.map(row => {
                return row.notekey;
            }));
        });
    });
    return keyz;
}
```

In `keylist`, the task is to collect the keys for all of the `Note` instances. As we said, `db.get` returns only the first entry of the result set, while the `db.all` function retrieves all the rows of the result set. Therefore, we use `db.all`, although `db.each` would have been a good alternative.

The contract for this function is to return an array of note keys. The `rows` object from `db.all` is an array of results from the database that contains the data we are to return, but we use the `map` function to convert the array into the format required by this function:

```
async count() {
  const db = await connectDB();
  const count = await new Promise((resolve, reject) => {
    db.get("select count(notekey) as count from notes",
      (err, row) => {
        if (err) return reject(err);
        resolve(row.count);
      });
  });
  return count;
}
```

In `count`, the task is similar, but we simply need a count of the rows in the table. SQL provides a `count()` function for this purpose, which we've used, and then because this result only has one row, we can again use `db.get`.

This enables us to run `Notes` with `NOTES_MODEL` set to `sqlite3`. With our code now set up, we can now proceed to run `Notes` with this database.

Running Notes with SQLite3

We're now ready to run the `Notes` application with SQLite3. Add the following code to the `scripts` section of `package.json`:

```
"sqlite3-setup": "sqlite3 chap07.sqlite3 --init models/schema-
sqlite3.sql",
"sqlite3-start": "cross-env SQLITE_FILE=chap07.sqlite3 DEBUG=notes:*
NOTES_MODEL=sqlite3 node ./app.mjs",
"sqlite3-server1": "cross-env SQLITE_FILE=chap07.sqlite3
NOTES_MODEL=sqlite3 PORT=3001 node ./app.mjs",
"sqlite3-server2": "cross-env SQLITE_FILE=chap07.sqlite3
NOTES_MODEL=sqlite3 PORT=3002 node ./app.mjs",
```

This sets up the commands that we'll use to test Notes on SQLite3.

We can run the server as follows:

```
$ npm run sqlite3-start  
  
> notes@0.0.0 sqlite3-start /home/david/Chapter07/notes  
> cross-env SQLITE_FILE=chap07.sqlite3 DEBUG=notes:  
NOTES_MODEL=sqlite3 node ./app.mjs  
  
notes:debug Listening on port 3000 +0ms
```

You can now browse the application at `http://localhost:3000` and run it through its paces, as before.

Because we still haven't made any changes to the View templates or CSS files, the application will look the same as before.

Of course, you can use the `sqlite` command, or other SQLite3 client applications, to inspect the database:

```
$ sqlite3 chap07.sqlite3  
SQLite version 3.30.1 2019-10-10 20:19:45  
Enter ".help" for usage hints.  
sqlite> select * from notes;  
hithere|Hi There||ho there what there  
himom|Hi Mom||This is where we say thanks
```

The advantage of installing the SQLite3 command-line tools is that we can perform any database administration tasks without having to write any code.

We have seen how to use SQLite3 with Node.js. It is a worthy database for many sorts of applications, plus it lets us use a SQL database without having to set up a server.

The next package that we will cover is an **Object Relations Management (ORM)** system that can run on top of several SQL databases.

Storing notes the ORM way with Sequelize

There are several popular SQL database engines, such as PostgreSQL, MySQL, and MariaDB. Corresponding to each are Node.js client modules that are similar in nature to the `sqlite3` module that we just used. The programmer is close to SQL, which can be good in the same way that driving a stick shift car is fun. But what if we want a higher-level view of the database so that we can think in terms of objects, rather than rows of a database table? **ORM** systems provide a suitable higher-level interface, and even offer the ability to use the same data model with several databases. Just as driving an electric car provides lots of benefits at the expense of losing out on the fun of stick-shift driving, ORM produces lots of benefits, while also distancing ourselves from the SQL.

The **Sequelize** package (<http://www.sequelizejs.com/>) is Promise-based, offers strong, well-developed ORM features, and can connect to SQLite3, MySQL, PostgreSQL, MariaDB, and MSSQL databases. Because Sequelize is Promise-based, it will fit naturally with the Promise-based application code we're writing.

A prerequisite to most SQL database engines is having access to a database server. In the previous section, we skirted around this issue by using SQLite3, which requires no database server setup. While it's possible to install a database server on your laptop, right now, we want to avoid the complexity of doing so, and so we will use Sequelize to manage a SQLite3 database. We'll also see that it's simply a matter of using a configuration file to run the same Sequelize code against a hosted database such as MySQL. In *Chapter 11, Deploying Node.js Microservices with Docker*, we'll learn how to use Docker to easily set up a service, including database servers, on our laptop and deploy the exact same configuration to a live server. Most web-hosting providers offer MySQL or PostgreSQL as part of their service.

Before we start on the code, let's install two modules:

```
$ npm install sequelize@6.x --save
$ npm install js-yaml@3.13.x --save
```

The first obviously installs the Sequelize package. The second, `js-yaml`, is installed so that we can implement a YAML-formatted file to store the Sequelize connection configuration. YAML is a human-readable **data serialization language**, which simply means it is an easy-to-use text file format to describe data objects.



Perhaps the best place to learn about YAML is its Wikipedia page, which can be found at <https://en.wikipedia.org/wiki/YAML>.

Let's start this by learning how to configure Sequelize, then we will create an `AbstractNotesStore` instance for Sequelize, and finally, we will test `Notes` using Sequelize.

Configuring Sequelize and connecting to a database

We'll be organizing the code for Sequelize support a little differently from before. We foresee that the `Notes` table is not the only data model that the `Notes` application will use. We could support additional features, such as the ability to upload images for a note or to allow users to comment on notes. This means having additional database tables and setting up relationships between database entries. For example, we might have a class named `AbstractCommentStore` to store comments, which will have its own database table and its own modules to manage the commented data. Both the `Notes` and `Comments` storage areas should be in the same database, and so they should share a database connection.

With that in mind, let's create a file, `models/sequlz.mjs`, to hold the code to manage the Sequelize connection:

```
import { promises as fs } from 'fs';
import { default as jsyaml } from 'js-yaml';
import Sequelize from 'sequelize';

let sequlz;

export async function connectDB() {
  if (typeof sequlz === 'undefined') {
    const yamltext = await fs.readFile(process.env.SEQUELIZE_CONNECT,
      'utf8');
    const params = jsyaml.safeLoad(yamltext, 'utf8');

    if (typeof process.env.SEQUELIZE_DBNAME !== 'undefined'
        && process.env.SEQUELIZE_DBNAME !== '') {
      params.dbname = process.env.SEQUELIZE_DBNAME;
    }
    if (typeof process.env.SEQUELIZE_DBUSER !== 'undefined'
        && process.env.SEQUELIZE_DBUSER !== '') {
```

```
    params.username = process.env.SEQUELIZE_DBUSER;
}
if (typeof process.env.SEQUELIZE_DBPASSWD !== 'undefined'
    && process.env.SEQUELIZE_DBPASSWD !== '') {
    params.password = process.env.SEQUELIZE_DBPASSWD;
}
if (typeof process.env.SEQUELIZE_DBHOST !== 'undefined'
    && process.env.SEQUELIZE_DBHOST !== '') {
    params.params.host = process.env.SEQUELIZE_DBHOST;
}
if (typeof process.env.SEQUELIZE_DBPORT !== 'undefined'
    && process.env.SEQUELIZE_DBPORT !== '') {
    params.params.port = process.env.SEQUELIZE_DBPORT;
}
if (typeof process.env.SEQUELIZE_DBdialect !== 'undefined'
    && process.env.SEQUELIZE_DBdialect !== '') {
    params.params.dialect = process.env.SEQUELIZE_DBdialect;
}

sequlz = new Sequelize(params.dbname,
                      params.username, params.password,
                      params.params);
await sequlz.authenticate();
}
return sequlz;
}

export async function close() {
    if (sequlz) sequlz.close();
    sequlz = undefined;
}
```

As with the SQLite3 module, the `connectDB` function manages the connection through Sequelize to a database server. Since the configuration of the Sequelize connection is fairly complex and flexible, we're not using environment variables for the whole configuration, but instead we use a YAML-formatted configuration file that will be specified in an environment variable. Sequelize uses four items of data—the database name, the username, the password, and a parameters object.

When we read in a YAML file, its structure directly corresponds to the object structure that's created. Therefore, with a YAML configuration file, we don't need to use up any brain cells developing a configuration file format. The YAML structure is dictated by the Sequelize `params` object, and our configuration file simply has to use the same structure.

We also allow overriding any of the fields in this file using environment variables. This will be useful when we deploy Notes using Docker so that we can configure database connections without having to rebuild the Docker container.

For a simple SQLite3-based database, we can use the following YAML file for configuration and name it `models/sequelize-sqlite.yaml`:

```
dbname: notes
username:
password:
params:
  dialect: sqlite
  storage: notes-sequelize.sqlite3
```

The `params.dialect` value determines what type of database to use; in this case, we're using SQLite3. Depending on the dialect, the `params` object can take different forms, such as a connection URL to the database. In this case, we simply need a filename, which is given here.

The `authenticate` call is there to test whether the database connected correctly.

The `close` function does what you expect—it closes the database connection.

With this design, we can easily change the database to use other database servers, just by adding a runtime configuration file. For example, it is easy to set up a MySQL connection; we just create a new file, such as `models/sequelize-mysql.yaml`, containing something similar to the following code:

```
dbname: notes
username: .. user name
password: .. password
params:
  host: localhost
  port: 3306
  dialect: mysql
```

This is straightforward. The `username` and `password` fields must correspond to the database credentials, while `host` and `port` will specify where the database is hosted. Set the database's `dialect` parameter and other connection information and you're good to go.

To use MySQL, you will need to install the base MySQL driver so that Sequelize can use MySQL:

```
$ npm install mysql@2.x --save
```

Running Sequelize against the other databases it supports, such as PostgreSQL, is just as simple. Just create a configuration file, install the Node.js driver, and install/configure the database engine.

The object returned from `connectDB` is a database connection, and as we'll see that it is used by Sequelize. So, let's get going with the real goal of this section—to define the `SequelizeNotesStore` class.

Creating a Sequelize model for the Notes application

As with the other data storage engines we've used, we need to create a subclass of `AbstractNotesStore` for Sequelize. This class will manage a set of notes using a Sequelize Model class.

Let's create a new file, `models/notes-sequelize.mjs`:

```
import { Note, AbstractNotesStore } from './Notes.mjs';
import Sequelize from 'sequelize';
import {
  connectDB as connectSequlz,
  close as closeSequlz
} from './sequlz.mjs';
import DBG from 'debug';
const debug = DBG('notes:notes-sequelize');
const error = DBG('notes:error-sequelize');

let sequelize;
export class SQNote extends Sequelize.Model {}

async function connectDB() {
  if (sequelize) return;
  sequelize = await connectSequlz();
  SQNote.init({
    notekey: { type: Sequelize.DataTypes.STRING,
      primaryKey: true, unique: true },
    title: Sequelize.DataTypes.STRING,
    body: Sequelize.DataTypes.TEXT
  }, {
```

```
    sequelize,
    modelName: 'SQNote'
  );
  await SQNote.sync();
}
```

The database connection is stored in the `sequelize` object, which is established by the `connectDB` function that we just looked at (which we renamed `connectSequlz`) to instantiate a Sequelize instance. We immediately return if the database is already connected.

In Sequelize, the `Model` class is where we define the data model for a given object. Each `Model` class corresponds to a database table. The `Model` class is a normal ES6 class, and we start by subclassing it to define the `SQNote` class. Why do we call it `SQNote`? That's because we already defined a `Note` class, so we had to use a different name in order to use both classes.

By calling `SQNote.init`, we initialize the `SQNote` model with the fields—that is, the schema—that we want it to store. The first argument to this function is the schema description and the second argument is the administrative data required by Sequelize.

As you would expect, the schema has three fields: `notekey`, `title`, and `body`. Sequelize supports a long list of data types, so consult the documentation for more on that. We are using `STRING` as the type for `notekey` and `title` since both handle a short text string up to 255 bytes long. The `body` field is defined as `TEXT` since it does not need a length limit. In the `notekey` field, you see it is an object with other parameters; in this case, it is described as the primary key and the `notekey` values must be unique.



Online documentation can be found at the following locations:

Sequelize class: <http://docs.sequelizejs.com/en/latest/api/sequelize/>

Defining

models: <http://docs.sequelizejs.com/en/latest/api/model/>

That manages the database connection and sets up the schema. Now, let's add the `SequelizeNotesStore` class to `models/notes-sequelize.mjs`:

```
export default class SequelizeNotesStore extends AbstractNotesStore {

  async close() {
    closeSequlz();
    sequelize = undefined;
  }
}
```

```
}

async update(key, title, body) {
    await connectDB();
    const note = await SQNote.findOne({ where: { notekey: key } });
    if (!note) {
        throw new Error(`No note found for ${key}`);
    } else {
        await SQNote.update({ title, body },
            { where: { notekey: key } });
        return this.read(key);
    }
}

async create(key, title, body) {
    await connectDB();
    const sqnote = await SQNote.create({
        notekey: key, title, body
    });
    return new Note(sqnote.notekey, sqnote.title, sqnote.body);
}

async read(key) {
    await connectDB();
    const note = await SQNote.findOne({ where: { notekey: key } });
    if (!note) {
        throw new Error(`No note found for ${key}`);
    } else {
        return new Note(note.notekey, note.title, note.body);
    }
}

async destroy(key) {
    await connectDB();
    await SQNote.destroy({ where: { notekey: key } });
}

async keylist() {
    await connectDB();
    const notes = await SQNote.findAll({ attributes: [ 'notekey' ] });
    const notekeys = notes.map(note => note.notekey);
    return notekeys;
}

async count() {
```

```
        await connectDB();
        const count = await SQNote.count();
        return count;
    }
}
```

The first thing to note is that in each function, we call static methods defined in the `SQNote` class to perform database operations. Sequelize model classes work this way, and there is a comprehensive list of these static methods in its documentation.

When creating a new instance of a Sequelize model class—in this case, `SQNote`—there are two patterns to follow. One is to call the `build` method and then to create the object and the `save` method to save it to the database. Alternatively, we can, as is done here, use the `create` method, which does both of these steps. This function returns an `SQNote` instance, called `sqnote` here, and if you consult the Sequelize documentation, you will see that these instances have a long list of methods available. The contract for our `create` method is to return a note, so we construct a `Note` object to return.

In this, and some other methods, we do not want to return a Sequelize object to our caller. Therefore, we construct an instance of our own `Note` class in order to return a clean object.

Our `update` method starts by calling `SQNote.findOne`. This is done to ensure that there is an entry in the database corresponding to the key that we're given. This function looks for the first database entry where `notekey` matches the supplied key. Following the happy path, where there is a database entry, we then use `SQNote.update` to update the `title` and `body` values, and by using the same `where` clause, it ensures the update operation targets the same database entry.

The Sequelize `where` clause offers a comprehensive list of matching operators. If you ponder this, it's clear it roughly corresponds to SQL as follows:

```
SELECT SQNotes SET title = ?, body = ? WHERE notekey = ?
```

That's what Sequelize and other ORM libraries do—convert the high-level API into database operations such as SQL queries.

To read a note, we use the `findOne` operation again. There is the possibility of it returning an empty result, and so we have to throw an error to match. The contract for this function is to return a `Note` object, so we take the fields retrieved using Sequelize to create a clean `Note` instance.

To destroy a note, we use the `destroy` operation with the same `where` clause to specify which entry to delete. This means that, as in the equivalent SQL statement (`DELETE FROM SQNotes WHERE notekey = ?`), if there is no matching note, no error will be thrown.

Because the `keylist` function acts on all `Note` objects, we use the `findAll` operation. The difference between `findOne` and `findAll` is obvious from the names.

While `findOne` returns the first matching database entry, `findAll` returns all of them. The `attributes` specifier limits the result set to include the named field—namely, the `notekey` field. This gives us an array of objects with a field named `notekey`. We then use a `.map` function to convert this into an array of note keys.

For the `count` function, we can just use the `count()` method to calculate the required result.

This allows us to use Sequelize by setting `NOTES_MODEL` to `sequelize`.

Having set up the functions to manage the database connection and defined the `SequelizeNotesStore` class, we're now ready to test the `Notes` application.

Running the Notes application with Sequelize

Now, we can get ready to run the `Notes` application using Sequelize. We can run it against any database server, but let's start with SQLite3. Add the following declarations to the `scripts` entry in `package.json`:

```
"sequelize-start": "cross-env DEBUG=notes:*
  SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_MODEL=sequelize
  node ./app.mjs",
"sequelize-server1": "cross-env SEQUELIZE_CONNECT=models/sequelize-
  sqlite.yaml NOTES_MODEL=sequelize PORT=3001 node ./app.mjs",
"sequelize-server2": "cross-env SEQUELIZE_CONNECT=models/sequelize-
  sqlite.yaml NOTES_MODEL=sequelize PORT=3002 node ./app.mjs",
```

This sets up commands to run a single server instance (or two).

Then, run it as follows:

```
$ npm run sequelize-start  
  
> notes@0.0.0 sequelize-start /home/david/Chapter07/notes  
> cross-env DEBUG=notes:* SEQUELIZE_CONNECT=models/sequelize-  
sqlite.yaml NOTES_MODEL=sequelize node ./app.mjs  
  
notes:debug Listening on port 3000 +0ms
```

As before, the application looks exactly the same because we haven't changed the View templates or CSS files. Put it through its paces and everything should work.

You will be able to start two instances; use separate browser windows to visit both instances and see whether they show the same set of notes.

To reiterate, to use the Sequelize-based model on a given database server, do the following:

1. Install and provision the database server instance; otherwise, get the connection parameters for an already-provisioned database server.
2. Install the corresponding Node.js driver.
3. Write a YAML configuration file corresponding to the connection parameters.
4. Create new scripts entries in package.json to automate starting Notes against the database.

By using Sequelize, we have dipped our toes into a powerful library for managing data in a database. Sequelize is one of several ORM libraries available for Node.js. We've already used the word *comprehensive* several times in this section as it's definitely the best word to describe Sequelize.

An alternative that is worthy of exploration is not an ORM library but is what's called a query builder. knex supports several SQL databases, and its role is to simplify creating SQL queries by using a high-level API.

In the meantime, we have one last database to cover before wrapping up this chapter: MongoDB, the leading NoSQL database.

Storing notes in MongoDB

MongoDB is widely used with Node.js applications, a sign of which is the popular **MEAN** acronym: **MongoDB (or MySQL), Express, Angular, and Node.js**. MongoDB is one of the leading NoSQL databases, meaning it is a database engine that does not use SQL queries. It is described as a *scalable, high-performance, open source, document-oriented database*. It uses JSON-style documents with no predefined, rigid schema and a large number of advanced features. You can visit their website for more information and documentation at <http://www.mongodb.org>.



Documentation on the Node.js driver for MongoDB can be found at
<https://www.npmjs.com/package/mongodb> and
<http://mongodb.github.io/node-mongodb-native/>.

Mongoose is a popular ORM for MongoDB (<http://mongoosejs.com/>). In this section, we'll use the native MongoDB driver instead, but Mongoose is a worthy alternative.

First, you will need a running MongoDB instance. The Compose- (<https://www.compose.io/>) and ScaleGrid- (<https://scalegrid.io/>) hosted service providers offer hosted MongoDB services. Nowadays, it is straightforward to host MongoDB as a Docker container as part of a system built of other Docker containers. We'll do this in Chapter 13, *Unit Testing and Functional Testing*.

It's possible to set up a temporary MongoDB instance for testing on, say, your laptop. It is available in all the operating system package management systems, or you can download a compiled package from mongodb.com. The MongoDB website also has instructions (<https://docs.mongodb.org/manual/installation/>).

For Windows, it may be most expedient to use a cloud-hosted MongoDB instance.

Once installed, it's not necessary to set up MongoDB as a background service. Instead, you can run a couple of simple commands to get a MongoDB instance running in the foreground of a command window, which you can kill and restart any time you like.

In a command window, run the following:

```
$ mkdir data
$ mongod --dbpath data
```

This creates a data directory and then runs the MongoDB daemon against the directory.

In another command window, you can test it as follows:

```
$ mongo
MongoDB shell version v4.2.2
connecting to:
mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mong
odb
Implicit session: session { "id" : UUID("308e285e-5496-43c5-81ca-
b04784927734") }
MongoDB server version: 4.2.2
> db.foo.save({ a: 1 });
WriteResult({ "nInserted" : 1 })
> db.foo.find();
{ "_id" : ObjectId("5e261ffc7e36ca9ed76d9552"), "a" : 1 }
>
bye
```

This runs the Mongo client program with which you can run commands. The command language used here is JavaScript, which is comfortable for us.

This saves a *document* in the collection named `foo`. The second command finds all documents in `foo`, printing them out for you. There is only one document, the one we just inserted, so that's what gets printed. The `_id` field is added by MongoDB and serves as a document identifier.

This setup is useful for testing and debugging. For a real deployment, your MongoDB server must be properly installed on a server. See the MongoDB documentation for these instructions.

With a working MongoDB installation in our hands, let's get started with implementing the `MongoNotesStore` class.

A MongoDB model for the Notes application

The official Node.js MongoDB driver (<https://www.npmjs.com/package/mongodb>) is created by the MongoDB team. It is very easy to use, as we will see, and its installation is as simple as running the following command:

```
$ npm install mongodb@3.x --save
```

This sets us up with the driver package and adds it to `package.json`.

Now, create a new file, `models/notes-mongodb.mjs`:

```
import { Note, AbstractNotesStore } from './Notes.mjs';
import mongodb from 'mongodb';
const MongoClient = mongodb.MongoClient;
import DBG from 'debug';
const debug = DBG('notes:notes-mongodb');
const error = DBG('notes:error-mongodb');

let client;

const connectDB = async () => {
  if (!client) client = await
MongoClient.connect(process.env.MONGO_URL);
}
const db = () => { return client.db(process.env.MONGO_DBNAME); };
```

This sets up the required imports, as well as the functions to manage a connection with the MongoDB database.

The `MongoClient` class is used to connect with a MongoDB instance. The required URL, which will be specified through an environment variable, uses a straightforward format: `mongodb://localhost/`. The database name is specified via another environment variable.

The documentation for the MongoDB Node.js driver can be found at <http://mongodb.github.io/node-mongodb-native/>.



There are both reference and API documentation available. In the *API* section, the `MongoClient` and `Db` classes are the ones that most relate to the code we are writing (<http://mongodb.github.io/node-mongodb-native/>).

The `connectDB` function creates the database client object. This object is only created as needed. The connection URL is provided through the `MONGO_URL` environment variable.

The `db` function is a simple wrapper around the client object to access the database that is used for the `Notes` application, which we specify via the `MONGO_DBNAME` environment variable. Therefore, to access the database, the code will have to call `db().mongoDbFunction()`.

Now, we can implement the `MongoDBNotesStore` class:

```
export default class MongoDBNotesStore extends AbstractNotesStore {

    async close() {
        if (client) client.close();
        client = undefined;
    }

    async update(key, title, body) {
        await connectDB();
        const note = new Note(key, title, body);
        const collection = db().collection('notes');
        await collection.updateOne({ notekey: key },
            { $set: { title: title, body: body } });
        return note;
    }

    async create(key, title, body) {
        await connectDB();
        const note = new Note(key, title, body);
        const collection = db().collection('notes');
        await collection.insertOne({
            notekey: key, title: title, body: body
        });
        return note;
    }

    async read(key) {
        await connectDB();
        const collection = db().collection('notes');
        const doc = await collection.findOne({ notekey: key });
        const note = new Note(doc.notekey, doc.title, doc.body);
        return note;
    }

    async destroy(key) {
        await connectDB();
        const collection = db().collection('notes');
        const doc = await collection.findOne({ notekey: key });
        if (!doc) {
            throw new Error(`No note found for ${key}`);
        } else {
            await collection.findOneAndDelete({ notekey: key });
            this.emitDestroyed(key);
        }
    }
}
```

```
async keylist() {
    await connectDB();
    const collection = db().collection('notes');
    const keyz = await new Promise((resolve, reject) => {
        const keyz = [];
        collection.find({}).forEach(
            note => { keyz.push(note.notekey); },
            err => {
                if (err) reject(err);
                else resolve(keyz);
            }
        );
    });
    return keyz;
}

async count() {
    await connectDB();
    const collection = db().collection('notes');
    const count = await collection.count({});
    return count;
}
}
```

MongoDB stores all documents in collections. A *collection* is a group of related documents and is analogous to a table in a relational database. This means creating a new document or updating an existing one starts by constructing it as a JavaScript object and then asking MongoDB to save the object to the database. MongoDB automatically encodes the object into its internal representation.

The `db().collection` method gives us a `Collection` object with which we can manipulate the named collection. In this case, we access the `notes` collection with `db().collection('notes')`.



For the documentation of the `Collection` class, see the MongoDB Node.js driver documentation referenced earlier.

In the `create` method, we use `insertOne`; as the method name implies, it inserts one document into the collection. This document is used for the fields of the `Note` class. Likewise, in the `update` method, the `updateOne` method first finds a document (in this case, by looking up the document with the matching `notekey` field) and then changes fields in the document, as specified, before saving the modified document back to the database.

The `read` method uses `db().findOne` to search for the note.

The `findOne` method takes what is called a *query selector*. In this case, we are requesting a match against the `notekey` field. MongoDB supports a comprehensive set of operators for query selectors.

On the other hand, the `updateOne` method takes what is called a *query filter*. As an update operation, it searches the database for a record that matches the filter, updates its fields based on the update descriptor, and then saves it back to the database.

For an overview of the MongoDB CRUD operations, including inserting documents, updating documents, querying for documents, and deleting documents, refer to <https://docs.mongodb.com/manual/crud/>.



For the documentation on query selectors, refer to <https://docs.mongodb.com/manual/reference/operator/query/#query-selectors>.

For the documentation on query filters, refer to <https://docs.mongodb.com/manual/core/document/#query-filter-documents>.

For the documentation on update descriptors, refer to <https://docs.mongodb.com/manual/reference/operator/update/>.

MongoDB has many variations of base operations. For example, `findOne` is a variation on the basic `find` method.

In our `destroy` method, we see another `find` variant, `findOneAndDelete`. As the name implies, it finds a document that matches the query descriptor and then deletes the document.

In the `keylist` method, we need to process every document in the collection, and so the `find` query selector is empty. The `find` operation returns a `Cursor`, which is an object used to navigate query results. The `Cursor.forEach` method takes two callbacks and is not a Promise-friendly operation, so we have to use a Promise wrapper. The first callback is called for every document in the query result, and in this case, we simply push the `notekey` field into an array. The second callback is called when the operation is finished, and we notify the Promise whether it succeeded or failed. This gives us our array of keys, which is returned to the caller.



For the documentation on the `Cursor` class, refer to <http://mongodb.github.io/node-mongodb-native/3.1/api/Cursor.html>.

In our `count` method, we simply call the MongoDB `count` method. The `count` method takes a query descriptor and, as the name implies, counts the number of documents that match the query. Since we've given an empty query selector, it ends up counting the entire collection.

This allows us to run Notes with `NOTES_MODEL` set to `mongodb` to use a MongoDB database.

Now that we have everything coded for MongoDB, we can proceed with testing Notes.

Running the Notes application with MongoDB

We are ready to test Notes using a MongoDB database. By now, you know the drill; add the following to the `scripts` section of `package.json`:

```
"mongodb-start": "cross-env DEBUG=notes:*
MONGO_URL=mongodb://localhost/ MONGO_DBNAME=chap07 NOTES_MODEL=mongodb
node ./app.mjs",
"mongodb-server1": "cross-env DEBUG=notes:*
MONGO_URL=mongodb://localhost/ MONGO_DBNAME=chap07 NOTES_MODEL=mongodb
PORT=3001 node ./app.mjs",
"mongodb-server2": "cross-env DEBUG=notes:*
MONGO_URL=mongodb://localhost/ MONGO_DBNAME=chap07 NOTES_MODEL=mongodb
PORT=3002 node ./app.mjs",
```

The `MONGO_URL` environment variable is the URL to connect with your MongoDB database. This URL is the one that you need to use to run MongoDB on your laptop, as outlined at the top of this section. If you have a MongoDB server somewhere else, you'll be provided with the relevant URL to use.

You can start the Notes application as follows:

```
$ npm run mongodb-start
> notes@0.0.0 mongodb-start /home/david/Chapter07/notes
> cross-env DEBUG=notes:* MONGO_URL=mongodb://localhost/
MONGO_DBNAME=chap07 NOTES_MODEL=mongodb node ./app.mjs

notes:debug Listening on port 3000 +0ms
```

The `MONGO_URL` environment variable should contain the URL for connecting with your MongoDB database. The URL shown here is correct for a MongoDB server started on the local machine, as would be the case if you started MongoDB at the command line as shown at the beginning of this section. Otherwise, if you have a MongoDB server provisioned somewhere else, you will have been told what the access URL is, and your `MONGO_URL` variable should have that URL.

You can start two instances of the `Notes` application and see that both share the same set of notes.

We can verify that the MongoDB database ends up with the correct value. First, start the MongoDB client program as so:

```
$ mongo chap07
MongoDB shell version v3.6.8
connecting to: mongodb://127.0.0.1:27017/chap07
```

Again, this is assuming the MongoDB configuration presented so far, and if your configuration differs then add the URL on the command line. This starts the interactive MongoDB shell, connected to the database configured for use by `Notes`. To inspect the content of the database, simply enter the command: `db.notes.find()`. That will print out every database entry.

With that, we have completed support not only for MongoDB but also for several other databases in the `Notes` application, and so we are now ready to wrap up the chapter.

Summary

In this chapter, we went through a real whirlwind of different database technologies. While we looked at the same seven functions over and over, it's useful to be exposed to the various data storage models and ways of getting things done. Even so, we only touched on the surface of options for accessing databases and data storage engines in `Node.js`.

By abstracting the model implementations correctly, we were able to easily switch data storage engines without changing the rest of the application. This technique lets us explore how subclassing works in JavaScript and the concept of creating different implementations of the same API. Additionally, we got a practical introduction to the `import()` function and saw how it can be used to dynamically choose which module to load.

In real-life applications, we frequently create abstractions for a similar purpose. They help us hide details or allow us to change implementations while insulating the rest of the application from the change. A dynamic import, which we used for our app, is useful for dynamically stitching together an application; for example, to load each module in a given directory.

We avoided the complexity of setting up database servers. As promised, we'll get into that in [Chapter 10, Deploying Node.js Applications to Linux Servers](#), when we explore the production deployment of Node.js applications.

By focusing our model code for the purpose of storing data, both the model and the application should be easier to test. We'll look at this in more depth in [Chapter 13, Unit Testing and Functional Testing](#).

In the next chapter, we'll focus on supporting multiple users, allowing them to log in and out, and authenticating users using OAuth 2.

8

Authenticating Users with a Microservice

Now that our Notes application can save its data in a database, we can think about the next phase of making this a real application—namely, authenticating our users.

It's so natural to log in to a website to use its services. We do it every day, and we even trust banking and investment organizations to secure our financial information through login procedures on a website. The **HyperText Transfer Protocol (HTTP)** is a stateless protocol, and a web application cannot tell much about one HTTP request compared with another. Because HTTP is stateless, HTTP requests do not natively know the user's identity, whether the user driving the web browser is logged in, or even whether the HTTP request was initiated by a human being.

The typical method for user authentication is to send a cookie containing a token to the browser, to carry the user's identity, and indicate whether that browser is logged in.

With Express, the best way to do this is with the `express-session` middleware, which handles session management with a cookie. It is easy to configure but is not a complete solution for user authentication since it does not handle user login/logout.

The package that appears to be leading the pack in user authentication is Passport (<http://passportjs.org/>). In addition to authenticating users against local user information, it supports a long list of third-party services against which to authenticate. With this, a website can be developed that lets users sign up with credentials from another website—Twitter, for example.

We will use Passport to authenticate users against either a locally stored database or a Twitter account. We'll also take this as an opportunity to explore a **representational state transfer (REST)**-based microservice with Node.js.

The rationale is the greater opportunity to increase security by storing user information in a highly protected enclave. Many application teams store user information in a well-protected barricaded area with a strictly controlled **application programming interface (API)**, and even physical access to the user information database, implementing as many technological barriers as possible against unapproved access. We're not going to go quite that far, but by the end of the book, the user information service will be deployed in its own Docker container.

In this chapter, we'll discuss the following three aspects of this phase:

- Creating a microservice to store user profile/authentication data.
- Authenticating a user with a locally stored password.
- Using OAuth2 to support authentication via third-party services.
Specifically, we'll use Twitter as a third-party authentication service.

Let's get started!

The first thing to do is to duplicate the code used for the previous chapter. For example, if you kept that code in the `chap07/notes` directory, create a new directory, `chap08/notes`.

Creating a user information microservice

We could implement user authentication and accounts by simply adding a user model and a few routes and views to the existing *Notes* application. While that's easy, is this what is done in a real-world production application?

Consider the high value of user identity information and the super-strong need for robust and reliable user authentication. Website intrusions happen regularly, and it seems the item most frequently stolen is user identities. To that end, we declared earlier an intention to develop a user information microservice, but we must first discuss the technical rationale for doing so.

Microservices are not a panacea, of course, meaning we shouldn't try to force-fit every application into the microservice box. By analogy, microservices fit with the Unix philosophy of small tools, each doing one thing well, which we mix/match/combine into larger tools. Another word for this is composability. While we can build a lot of useful software tools with that philosophy, does it work for applications such as Photoshop or LibreOffice?

This is why microservices are popular today among application teams. Microservice architectures are more agile if used well. And, as we noted earlier, we're aiming for a highly secured microservice deployment.

With that decision out of the way, there are two other decisions to be made with regard to security implications. They are as follows:

- Do we create our own REST application framework?
- Do we create our own user login/authentication framework?

In many cases, it is better to use a well-regarded existing library where the maintainers have already stomped out lots of bugs, just as we used the Sequelize **ORM (Object-Relational Mapping)** library in the previous chapter, because of its maturity. We have identified two libraries for this phase of the Notes project.

We already mentioned using Passport for user login support, as well as authenticating Twitter users.

For REST support, we could have continued using Express, but instead will use Restify (<http://restify.com/>), which is a popular REST-centric application framework.

To test the service, we'll write a command-line tool for administering user information in the database. We won't be implementing an administrative user interface in the Notes application, and will instead rely on this tool to administer the users. As a side effect, we'll have a tool for testing the user service.

Once this service is functioning correctly, we'll set about modifying the Notes application to access user information from the service, while using Passport to handle authentication.

The first step is to create a new directory to hold the user information microservice. This should be a sibling directory to the Notes application. If you created a directory named `chap08/notes` to hold the Notes application, then create a directory named `chap08/users` to hold the microservice.

Then, in the `chap08/users` directory, run the following commands:

```
$ cd users
$ npm init
.. answer questions
.. name - user-auth-server
$ npm install debug@^4.1.x fs-extra@^9.x js-yaml@^3.14.x \
restify@^8.5.x restify-clients@^2.6.x sequelize@^6.x \
sqlite3@^5.x commander@^5.x cross-env@7.x --save
```

This gets us ready to start coding. We'll use the `debug` module for logging messages, `js-yaml` to read the Sequelize configuration file, `restify` for its REST framework, and `sequelize/sqlite3` for database access.

In the sections to come, we will develop a database model to store user information, and then create a REST service to manage that data. To test the service, we'll create a command-line tool that uses the REST API.

Developing the user information model

We'll be storing the user information using a Sequelize-based model in a SQL database. We went through that process in the previous chapter, but we'll do it a little differently this time. Rather than go for the ultimate flexibility of using any kind of database, we'll stick with Sequelize since the user information model is very simple and a SQL database is perfectly adequate.

The project will contain two modules. In this section, we'll create `users-sequelize.mjs`, which will define the `SQUser` schema and a couple of utility functions. In the next section, we'll start on `user-server.mjs`, which contains the REST server implementation.

First, let's ponder an architectural preference. Just how much should we separate between the data model code interfacing with the database from the REST server code? In the previous chapter, we went for a clean abstraction with several implementations of the database storage layer. For a simple server such as this, the REST request handler functions could contain all database calls, with no abstraction layer. Which is the best approach? We don't have a hard rule to follow. For this server, we will have database code more tightly integrated to the router functions, with a few shared functions.

Create a new file named `users-sequelize.mjs` in `users` containing the following code:

```
import Sequelize from "sequelize";
import { default as jsyaml } from 'js-yaml';
import { promises as fs } from 'fs';
import * as util from 'util';
import DBG from 'debug';
const log = DBG('users:model-users');
const error = DBG('users:error');

var sequlz;

export class SQUser extends Sequelize.Model {}

export async function connectDB() {
  if (sequlz) return sequlz;

  const yamltext = await fs.readFile(process.env.SEQUELIZE_CONNECT,
                                      'utf8');
  const params = await jsyaml.safeLoad(yamltext, 'utf8');

  if (typeof process.env.SEQUELIZE_DBNAME !== 'undefined'
      && process.env.SEQUELIZE_DBNAME !== '') {
    params.dbname = process.env.SEQUELIZE_DBNAME;
  }
  if (typeof process.env.SEQUELIZE_DBUSER !== 'undefined'
      && process.env.SEQUELIZE_DBUSER !== '') {
    params.username = process.env.SEQUELIZE_DBUSER;
  }
  if (typeof process.env.SEQUELIZE_DBPASSWD !== 'undefined'
      && process.env.SEQUELIZE_DBPASSWD !== '') {
    params.password = process.env.SEQUELIZE_DBPASSWD;
  }
  if (typeof process.env.SEQUELIZE_DBHOST !== 'undefined'
      && process.env.SEQUELIZE_DBHOST !== '') {
    params.params.host = process.env.SEQUELIZE_DBHOST;
  }
  if (typeof process.env.SEQUELIZE_DBPORT !== 'undefined'
      && process.env.SEQUELIZE_DBPORT !== '') {
    params.params.port = process.env.SEQUELIZE_DBPORT;
  }
  if (typeof process.env.SEQUELIZE_DBdialect !== 'undefined'
      && process.env.SEQUELIZE_DBdialect !== '') {
    params.params.dialect = process.env.SEQUELIZE_DBdialect;
  }

  log('Sequelize params ' + util.inspect(params));
}
```

```
sequulz = new Sequelize(params.dbname, params.username,
                       params.password, params.params);
SQUser.init({
  username: { type: Sequelize.STRING, unique: true },
  password: Sequelize.STRING,
  provider: Sequelize.STRING,
  familyName: Sequelize.STRING,
  givenName: Sequelize.STRING,
  middleName: Sequelize.STRING,
  emails: Sequelize.STRING(2048),
  photos: Sequelize.STRING(2048)
}, {
  sequelize: sequulz,
  modelName: 'SQUser'
});
await SQUser.sync();
}
```

As with our Sequelize-based model for Notes, we will use a **YAML Ain't Markup Language (YAML)** file to store connection configuration. We're even using the same environment variable, `SEQUELIZE_CONNECT`, and the same approach to overriding fields of the configuration. The approach is similar, with a `connectDB` function setting up the connection and initializing the `SQUsers` table.

With this approach, we can use a base configuration file in the `SEQUELIZE_CONNECT` variable and then use the other environment variables to override its fields. This will be useful when we start deploying Docker containers.



The user profile schema shown here is derived from the normalized profile provided by Passport—for more information, refer to <http://www.passportjs.org/docs/profile>.

The Passport project developed this object by harmonizing the user information given by several third-party services into a single object definition. To simplify our code, we're simply using the schema defined by Passport.

There are several functions to create that will be an API to manage user data. Let's add them to the bottom of `users-sequelize.mjs`, starting with the following code:

```
export function userParams(req) {
  return {
    username: req.params.username,
    password: req.params.password,
    provider: req.params.provider,
    familyName: req.params.familyName,
```

```
    givenName: req.params.givenName,
    middleName: req.params.middleName,
    emails: JSON.stringify(req.params.emails),
    photos: JSON.stringify(req.params.photos)
  };
}
```

In Restify, the route handler functions supply the same sort of `request` and `response` objects we've already seen. We'll go over the configuration of the REST server in the next section. Suffice to say that REST parameters arrive in the request handlers as the `req.params` object, as shown in the preceding code block. This function simplifies the gathering of those parameters into a simple object that happens to match the `SQUser` schema, as shown in the following code block:

```
export function sanitizedUser(user) {
  var ret = {
    id: user.username,
    username: user.username,
    provider: user.provider,
    familyName: user.familyName,
    givenName: user.givenName,
    middleName: user.middleName
  };
  try {
    ret.emails = JSON.parse(user.emails);
  } catch(e) { ret.emails = []; }
  try {
    ret.photos = JSON.parse(user.photos);
  } catch(e) { ret.photos = []; }
  return ret;
}
```

When we fetch an `SQUser` object from the database, Sequelize obviously gives us a Sequelize object that has many extra fields and functions used by Sequelize. We don't want to send that data to our callers. Furthermore, we think it will increase security to not provide the `password` data beyond the boundary of this server. This function produces a simple, sanitized, anonymous JavaScript object from the `SQUser` instance. We could have defined a full JavaScript class, but would that have served any purpose? This anonymous JavaScript class is sufficient for this simple server, as illustrated in the following code block:

```
export async function findOneUser(username) {
  let user = await SQUser.findOne({ where: { username: username } });
  user = user ? sanitizedUser(user) : undefined;
  return user;
}
```

```
export async function createUser(req) {
  let tocreate = userParams(req);
  await SQUser.create(tocreate);
  const result = await findOneUser(req.params.username);
  return result;
}
```

The pair of functions shown in the preceding code block provides some database operations that are used several times in the `user-server.mjs` module.

In `findOneUser`, we are looking up a single `SQUser`, and then returning a sanitized copy. In `createUser`, we gather the user parameters from the request object, create the `SQUser` object in the database, and then retrieve that newly created object to return it to the caller.

If you refer back to the `connectDB` function, there is a `SEQUELIZE_CONNECT` environment variable for the configuration file. Let's create one for SQLite3 that we can name `sequelize-sqlite.yaml`, as follows:

```
dbname: users
username:
password:
params:
  dialect: sqlite
  storage: users-sequelize.sqlite3
```

This is just like the configuration files we used in the previous chapter.

That's what we need for the database side of this service. Let's now move on to creating the REST service.

Creating a REST server for user information

The user information service is a REST server to handle user information data and authentication. Our goal is, of course, to integrate that with the Notes application, but in a real project, such a user information service could be integrated with several web applications. The REST service will provide functions we found useful while developing the user login/logout support in Notes, which we'll show later in the chapter.

In the package.json file, change the main tag to the following line of code:

```
"main": "user-server.mjs",
```

This declares that the module we're about to create, user-server.mjs, is the main package of this project.

Make sure the scripts section contains the following script:

```
"start": "cross-env DEBUG=users:* PORT=5858  
SEQUELIZE_CONNECT=sequelize-sqlite.yaml node ./user-server.mjs"
```

Clearly, this is how we'll start our server. It uses the configuration file from the previous section and specifies that we'll listen on port 5858.

Then, create a file named user-server.mjs containing the following code:

```
import restify from 'restify';
import * as util from 'util';
import { SQUser, connectDB, userParams, findOneUser,
        createUser, sanitizedUser } from './users-sequelize.mjs';

import DBG from 'debug';
const log = DBG('users:service');
const error = DBG('users:error');

/////////// Set up the REST server

var server = restify.createServer({
    name: "User-Auth-Service",
    version: "0.0.1"
});

server.use(restify.plugins.authorizationParser());
server.use(check);
server.use(restify.plugins.queryParser());
server.use(restify.plugins.bodyParser({
    mapParams: true
}));

server.listen(process.env.PORT, "localhost", function() {
    log(server.name + ' listening at ' + server.url);
});

process.on('uncaughtException', function(err) {
    console.error("UNCAUGHT EXCEPTION - " + (err.stack || err));
    process.exit(1);
});
```

```
process.on('unhandledRejection', (reason, p) => {
  console.error(`UNHANDLED PROMISE REJECTION: ${util.inspect(p)}`)
  reason: `${reason}`;
  process.exit(1);
});
```

We're using Restify, rather than Express, to develop this server. Obviously, the Restify API has similarities with Express, since both point to the Ruby framework Sinatra for inspiration. We'll see even more similarities when we talk about the route handler functions.

What we have here is the core setup of the REST server. We created the server object and added a few things that, in Express, were called *middleware*, but what Restify simply refers to as *handlers*. A Restify handler function serves the same purpose as an Express middleware function. Both frameworks let you define a function chain to implement the features of your service. One calls it a *middleware* function and the other calls it a *handler* function, but they're almost identical in form and function.

We also have a collection of listener functions that print a startup message and handle uncaught errors. You do remember that it's important to catch the uncaught errors?

An interesting thing is that, since REST services are often versioned, Restify has built-in support for handling version numbers. Restify supports **semantic versioning (SemVer)** version matching in the Accept-Version HTTP header.

In the *handlers* that were installed, they obviously have to do with authorization and parsing parameters from the **Uniform Resource Locator (URL)** query string and from the HTTP body. The handlers with names starting with `restify.plugins` are maintained by the Restify team, and documented on their website.

That leaves the handler simply named `check`. This handler is in `user-server.mjs` and provides a simple mechanism of token-based authentication for REST clients.

Add the following code to the bottom of `user-server.mjs`:

```
// Mimic API Key authentication.

var apiKey = [
  { user: 'them', key: 'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF' } ];

function check(req, res, next) {
  if (req.authorization && req.authorization.basic) {
    var found = false;
```

```
for (let auth of apiKeyes) {
  if (auth.key === req.authorization.basic.password
    && auth.user === req.authorization.basic.username) {
    found = true;
    break;
  }
}
if (found) next();
else {
  res.send(401, new Error("Not authenticated"));
  next(false);
}
} else {
  res.send(500, new Error('No Authorization Key'));
  next(false);
}
}
```

This handler executes for every request and immediately follows `restify.plugins.authorizationParser`. It looks for authorization data—specifically, HTTP basic authorization—to have been supplied in the HTTP request. It then loops through the list of keys in the `apiKeyes` array, and if the Basic Auth parameters supplied matched, then the caller is accepted.

This should not be taken as an example of a best practice since HTTP Basic Auth is widely known to be extremely insecure, among other issues. But it demonstrates the basic concept, and also shows that enforcing token-based authorization is easily done with a similar handler.

This also shows us the function signature of a Restify handler function—namely, that it is the same signature used for Express middleware, the `request` and `result` objects, and the `next` callback.

There is a big difference between Restify and Express as to how the `next` callback is used. In Express, remember that a middleware function calls `next` unless that middleware function is the last function on the processing chain—for example if the function has called `res.send` (or equivalent) to send a response to the caller. In Restify, every handler function calls `next`. If a handler function knows it should be the last function on the handler chain, then it uses `next(false)`; otherwise, it calls `next()`. If a handler function needs to indicate an error, it calls `next(err)`, where `err` is an object where `instanceof Error` is `true`.

Consider the following hypothetical handler function:

```
server.use((req, res, next) => {
    // ... processing
    if (foundErrorCondition) {
        next(new Error('Describe error condition'));
    } else if (successfulConclusion) {
        res.send(results);
        next(false);
    } else {
        // more processing must be required
        next();
    }
});
```

This shows the following three cases:

1. Errors are indicated with `next(new Error('Error description'))`.
2. Completion is indicated with `next(false)`.
3. The continuation of processing is indicated with `next()`.

We have created the starting point for a user information data model and the matching REST service. The next thing we need is a tool to test and administer the server.

What we want to do in the following sections is two things. First, we'll create the REST handler functions to implement the REST API. At the same time, we'll create a command-line tool that will use the REST API and let us both test the server and add or delete users.

Creating a command-line tool to test and administer the user authentication server

To give ourselves assurance that the user authentication server works, let's write a tool with which to exercise the server that can also be used for administration. In a typical project, we'd create not only a customer-facing web user interface, but also an administrator-facing web application to administer the service. Instead of doing that here, we'll create a command-line tool.

The tool will be built with Commander, a popular framework for developing command-line tools in Node.js. With Commander, we can easily build a **command-line interface (CLI)** tool supporting the program verb `--option optionValue` parameter pattern.



For documentation on Commander, see <https://www.npmjs.com/package/commander>.

Any command-line tool looks at the `process.argv` array to know what to do. This array contains strings parsed from what was given on the command line. The concept for all this goes way back to the earliest history of Unix and the C programming language.



For documentation on the `process.argv` array, refer to https://nodejs.org/api/process.html#process_process_argv.

By using Commander, we have a simpler path of dealing with the command line. It uses a declarative approach to handling command-line parameters. This means we use Commander functions to declare the options and sub-commands to be used by this program, and then we ask Commander to parse the command line the user supplies. Commander then calls the functions we declare based on the content of the command line.

Create a file named `cli.mjs` containing the following code:

```
import { default as program } from 'commander';
import { default as restify } from 'restify-clients';
import * as util from 'util';

var client_port;
var client_host;
var client_version = '*';
var client_protocol;
var authid = 'them';
var authcode = 'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF';

const client = (program) => {
  if (typeof process.env.PORT === 'string')
    client_port = Number.parseInt(process.env.PORT);
  if (typeof program.port === 'string')
    client_port = Number.parseInt(program.port);
  if (typeof program.host === 'string') client_host = program.host;
  if (typeof program.url === 'string') {
    let purl = new URL(program.url);
    if (purl.host && purl.host !== '') client_host = purl.host;
    if (purl.port && purl.port !== '') client_port = purl.port;
    if (purl.protocol && purl.protocol !== '') client_protocol =
      purl.protocol;
  }
}
```

```
    purl.protocol;
}
let connect_url = new URL('http://localhost:5858');
if (client_protocol) connect_url.protocol = client_protocol;
if (client_host) connect_url.host = client_host;
if (client_port) connect_url.port = client_port;
let client = restify.createJsonClient({
  url: connect_url.href,
  version: client_version
});
client.basicAuth(authid, authcode);
return client;
}

program
.option('-p, --port <port>',
  'Port number for user server, if using localhost')
.option('-h, --host <host>',
  'Port number for user server, if using localhost')
.option('-u, --url <url>',
  'Connection URL for user server, if using a remote server');
```

This is just the starting point of the command-line tool. For most of the REST handler functions, we'll also implement a sub-command in this tool. We'll take care of that code in the subsequent sections. For now, let's focus on how the command-line tool is set up.

The Commander project suggests we name the default import `program`, as shown in the preceding code block. As mentioned earlier, we declare the command-line options and sub-commands by calling methods on this object.

In order to properly parse the command line, the last line of code in `cli.mjs` must be as follows:

```
program.parse(process.argv);
```

The `process.argv` variable is, of course, the command-line arguments split out into an array. Commander, then, is processing those arguments based on the options' declarations.

For the REST client, we use the `restify-clients` package. As the name implies, this is a companion package to Restify and is maintained by the Restify team.

At the top of this script, we declare a few variables to hold connection parameters. The goal is to create a connection URL to access the REST service. The `connect_url` variable is initialized with the default value, which is port 5858 on the localhost.

The function named `client` looks at the information Commander parses from the command line, as well as a number of environment variables. From that data, it deduces any modification to the `connect_url` variable. The result is that we can connect to this service on any server from our laptop to a faraway cloud-hosted server.

We've also hardcoded the access token and the use of Basic Auth. Put on the backlog a high-priority task to change to a stricter form of authentication.

Where do the values of `program.port`, `program.host`, and `program.url` come from? We declared those variables—that's where they came from.

Consider the following line of code:

```
program.option('-p, --port <port>', 'Long Description of the option');
```

This declares an option, either `-p` or `--port`, that Commander will parse out of the command line. Notice that all we do is write a text string and, from that, Commander knows it must parse these options. Isn't this easy?

When it sees one of these options, the `<port>` declaration tells Commander that this option requires an argument. It will parse that argument out of the command line, and then assign it to `program.port`.

Therefore, `program.port`, `program.host`, and `program.url` were all declared in a similar way. When Commander sees those options, it will create the matching variables, and then our `client` function will take that data and modify `connect_url` appropriately.

One of the side effects of these declarations is that Commander can generate help text automatically. The result we'll achieve is being able to type the following code:

```
$ node cli.mjs --help
Usage: cli.mjs [options] [command]

Options:
  -p, --port <port> Port number for user server, if using localhost
  -h, --host <host> Port number for user server, if using localhost
  -u, --url <url> Connection URL for user server, if using a remote
    server
  -h, --help output usage information

Commands:
  add [options] <username> Add a user to the user server
  find-or-create [options] <username> Add a user to the user server
```

```
update [options] <username> Add a user to the user server
destroy <username> Destroy a user on the user server
find <username> Search for a user on the user server
list-users List all users on the user server
```

The text comes directly from the descriptive text we put in the declarations. Likewise, each of the sub-commands also takes a `--help` option to print out corresponding help text.

With all that out of the way, let's start creating these commands and REST functions.

Creating a user in the user information database

We have the starting point for the REST server, and the starting point for a command-line tool to administer the server. Let's start creating the functions—and, of course, the best place to start is to create an `SQUser` object.

In `user-server.mjs`, add the following route handler:

```
server.post('/create-user', async (req, res, next) => {
  try {
    await connectDB();
    let result = await createUser(req);
    res.contentType = 'json';
    res.send(result);
    next(false);
  } catch(err) {
    res.send(500, err);
    next(false);
  }
});
```

This handles a `POST` request on the `/create-user` URL. This should look very similar to an Express route handler function, apart from the use of the `next` callback. Refer back to the discussion on this. As we did with the Notes application, we declare the handler callback as an `async` function and then use a `try/catch` structure to catch all errors and report them as errors.

The handler starts with `connectDB` to ensure the database is set up. Then, if you refer back to the `createUser` function, you see it gathers up the user data from the request parameters and then uses `SQUser.create` to create an entry in the database. What we will receive here is the sanitized user object, and we simply return that to the caller.

Let's also add the following code to `user-server.mjs`:

```
server.post('/find-or-create', async (req, res, next) => {
  try {
    await connectDB();
    let user = await findOneUser(req.params.username);
    if (!user) {
      user = await createUser(req);
      if (!user) throw new Error('No user created');
    }
    res.contentType = 'json';
    res.send(user);
    return next(false);
  } catch(err) {
    res.send(500, err);
    next(false);
  }
});
```

This is a variation on creating an SQUser. While implementing login support in the Notes application, there was a scenario in which we had an authenticated user that may or may not already have an SQUser object in the database. In this case, we look to see whether the user already exists and, if not, then we create that user.

Let's turn now to `cli.mjs` and implement the sub-commands to handle these two REST functions, as follows:

```
program
  .command('add <username>')
  .description('Add a user to the user server')
  .option('--password <password>', 'Password for new user')
  .option('--family-name <familyName>',
    'Family name, or last name, of the user')
  .option('--given-name <givenName>', 'Given name, or first name,
    of the user')
  .option('--middle-name <middleName>', 'Middle name of the user')
  .option('--email <email>', 'Email address for the user')
  .action((username, cmdObj) => {
    const topost = {
      username, password: cmdObj.password, provider: "local",
      familyName: cmdObj.familyName,
      givenName: cmdObj.givenName,
      middleName: cmdObj.middleName,
      emails: [], photos: []
    };
    if (typeof cmdObj.email !== 'undefined')
      topost.emails.push(cmdObj.email);
```

```
client(program).post('/create-user', topost,
  (err, req, res, obj) => {
    if (err) console.error(err.stack);
    else console.log('Created ' + util.inspect(obj));
  });
});
```

By using `program.command`, we are declaring a sub-command—in this case, `add`. The `<username>` declaration says that this sub-command takes an argument. Commander will provide that argument value in the `username` parameter to the function passed in the `action` method.

The structure of a `program.command` declaration is to first declare the syntax of the sub-command. The `description` method provides user-friendly documentation. The `option` method calls are options for this sub-command, rather than global options. Finally, the `action` method is where we supply a callback function that will be invoked when Commander sees this sub-command in the command line.

Any arguments declared in the `program.command` string end up as parameters to that callback function.

Any values for the options for this sub-command will land in the `cmdObj` object. By contrast, the value for global options is attached to the `program` object.

With that understanding, we can see that this sub-command gathers information from the command line and then uses the `client` function to connect to the server. It invokes the `/create-user` URL, passing along the data gathered from the command line. Upon receiving the response, it will print either the error or the result object.

Let's now add the sub-command corresponding to the `/find-or-create` URL, as follows:

```
program
  .command('find-or-create <username>')
  .description('Add a user to the user server')
  .option('--password <password>', 'Password for new user')
  .option('--family-name <familyName>',
    'Family name, or last name, of the user')
  .option('--given-name <givenName>', 'Given name, or first name,
    of the user')
  .option('--middle-name <middleName>', 'Middle name of the user')
  .option('--email <email>', 'Email address for the user')
  .action((username, cmdObj) => {
    const topost = {
      username, password: cmdObj.password, provider: "local",
```

```
    familyName: cmdObj.familyName,
    givenName: cmdObj.givenName,
    middleName: cmdObj.middleName,
    emails: [], photos: []
  };
  if (typeof cmdObj.email !== 'undefined')
    topost.emails.push(cmdObj.email);
  client(program).post('/find-or-create', topost,
    (err, req, res, obj) => {
      if (err) console.error(err.stack);
      else console.log('Found or Created ' + util.inspect(obj));
    });
});
```

This is very similar, except for calling /find-or-create.

We have enough here to run the server and try the following two commands:

```
$ npm start

> user-auth-server@1.0.0 start /home/david/Chapter08/users
> DEBUG=users:* PORT=5858 SEQUELIZE_CONNECT=sequelize-sqlite.yaml node
./user-server.mjs

  users:service User-Auth-Service listening at http://127.0.0.1:5858
+0ms
```

We run this in one command window to start the server. In another command window, we can run the following command:

```
$ node cli.mjs add --password w0rd --family-name Einarrsdottir --
given-name Ashildr --email me@stolen.tardis me
Created {
  id: 'me',
  username: 'me',
  provider: 'local',
  familyName: 'Einarrsdottir',
  givenName: 'Ashildr',
  middleName: null,
  emails: [ 'me@stolen.tardis' ],
  photos: []
}
```

Over in the server window, it will print a trace of the actions taken in response to this. But it's what we expect: the values we gave on the command line are in the database, as shown in the following code block:

```
$ node cli.mjs find-or-create --password foooo --family-name Smith --  
given-name John --middle-name Snuffy --email snuffy@example.com  
snuffy-smith  
Found or Created {  
  id: 'snuffy-smith',  
  username: 'snuffy-smith',  
  provider: 'local',  
  familyName: 'Smith',  
  givenName: 'John',  
  middleName: 'Snuffy',  
  emails: [ 'snuffy@example.com' ],  
  photos: []  
}
```

Likewise, we have success with the `find-or-create` command.

That gives us the ability to create `SQUser` objects. Next, let's see how to read from the database.

Reading user data from the user information service

The next thing we want to support is to look for users in the user information service. Instead of a general search facility, the need is to retrieve an `SQUser` object for a given username. We already have the utility function for this purpose; it's just a matter of hooking up a REST endpoint.

In `user-server.mjs`, add the following function:

```
server.get('/find/:username', async (req, res, next) => {  
  try {  
    await connectDB();  
    const user = await findOneUser(req.params.username);  
    if (!user) {  
      res.send(404, new Error("Did not find " + req.params.username));  
    } else {  
      res.contentType = 'json';  
      res.send(user);  
    }  
    next(false);  
  } catch(err) {  
    res.send(500, err);  
    next(false);  
  }  
});
```

```
    }  
});
```

And, as expected, that was easy enough. For the /find URL, we need to supply the username in the URL. The code simply looks up the SQUser object using the existing utility function.

A related function retrieves the SQUser objects for all users. Add the following code to `user-server.mjs`:

```
server.get('/list', async (req, res, next) => {  
  try {  
    await connectDB();  
    let userlist = await SQUser.findAll({});  
    userlist = userlist.map(user => sanitizedUser(user));  
    if (!userlist) userlist = [];  
    res.contentType = 'json';  
    res.send(userlist);  
    next(false);  
  } catch(err) {  
    res.send(500, err);  
    next(false);  
  }  
});
```

We know from the previous chapter that the `findAll` operation retrieves all matching objects and that passing an empty query selector such as this causes `findAll` to match every SQUser object. Therefore, this performs the task we described, to retrieve information on all users.

Then, in `cli.mjs`, we add the following sub-command declarations:

```
program  
.command('find <username>')  
.description('Search for a user on the user server')  
.action((username, cmdObj) => {  
  client(program).get(`/find/${username}`),  
  (err, req, res, obj) => {  
    if (err) console.error(err.stack);  
    else console.log('Found ' + util.inspect(obj));  
  });  
});  
  
program  
.command('list-users')  
.description('List all users on the user server')  
.action((cmdObj) => {
```

```
        client(program).get('/list', (err, req, res, obj) => {
          if (err) console.error(err.stack);
          else console.log(obj);
        });
      );
    });
  );
});
```

This is similarly easy. We pass the username provided on our command line in the `/find` URL and then print out the result. Likewise, for the `list-users` sub-command, we simply call `/list` on the server and print out the result.

After restarting the server, we can test the commands, as follows:

```
$ node cli.mjs find me
Found {
  id: 'me',
  username: 'me',
  provider: 'local',
  familyName: 'Einarrsdottir',
  givenName: 'Ashildr',
  middleName: null,
  emails: [ 'me@stolen.tardis' ],
  photos: []
}
$ node cli.mjs list-users
[
  {
    id: 'snuffy-smith',
    username: 'snuffy-smith',
    provider: 'local',
    familyName: 'Smith',
    givenName: 'John',
    middleName: 'Snuffy',
    emails: [ 'snuffy2@gmail.com' ],
    photos: []
  },
  {
    id: 'me',
    username: 'me',
    provider: 'local',
    familyName: 'Einarrsdottir',
    givenName: 'Ashildr',
    middleName: null,
    emails: [ 'me@stolen.tardis' ],
    photos: []
  }
]
```

And, indeed, the results came in as we expected.

The next operation we need is to update an SQUser object.

Updating user information in the user information service

The next functionality to add is to update user information. For this, we can use the Sequelize `update` function, and simply expose it as a REST operation.

To that end, add the following code to `user-server.mjs`:

```
server.post('/update-user/:username', async (req, res, next) => {
  try {
    await connectDB();
    let toupdate = userParams(req);
    await SQUser.update(toupdate, { where: { username:
      req.params.username } });
    const result = await findOneUser(req.params.username);
    res.contentType = 'json';
    res.send(result);
    next(false);
  } catch(err) {
    res.send(500, err);
    next(false);
  }
});
```

The caller is to provide the same set of user information parameters, which will be picked up by the `userParams` function. We then use the `update` function, as expected, and then retrieve the modified SQUser object, sanitize it, and send it as the result.

To match that function, add the following code to `cli.mjs`:

```
program
  .command('update <username>')
  .description('Add a user to the user server')
  .option('--password <password>', 'Password for new user')
  .option('--family-name <familyName>',
    'Family name, or last name, of the user')
  .option('--given-name <givenName>', 'Given name, or first name,
    of the user')
  .option('--middle-name <middleName>', 'Middle name of the user')
  .option('--email <email>', 'Email address for the user')
```

```
.action((username, cmdObj) => {
  const topost = {
    username, password: cmdObj.password,
    familyName: cmdObj.familyName,
    givenName: cmdObj.givenName,
    middleName: cmdObj.middleName,
    emails: [], photos: []
  };
  if (typeof cmdObj.email !== 'undefined')
    topost.emails.push(cmdObj.email);
  client(program).post(`/update-user/${username}`, topost,
    (err, req, res, obj) => {
      if (err) console.error(err.stack);
      else console.log('Updated ' + util.inspect(obj));
    });
});
```

As expected, this sub-command must take the same set of user information parameters. It then bundles those parameters into an object, posting it to the /update-user endpoint on the REST server.

Then, to test the result, we run the command, like so:

```
$ node cli.mjs update --password fooooey --family-name Smith --given-
name John --middle-name Snuffy --email snuffy3@gmail.com snuffy-smith
Updated {
  id: 'snuffy-smith',
  username: 'snuffy-smith',
  provider: 'local',
  familyName: 'Smith',
  givenName: 'John',
  middleName: 'Snuffy',
  emails: [ 'snuffy3@gmail.com' ],
  photos: []
}
```

And, indeed, we managed to change Snuffy's email address.

The next operation is to delete an SQUser object.

Deleting a user record from the user information service

Our next operation will complete the **create, read, update, and delete (CRUD)** operations by letting us delete a user.

Add the following code to `user-server.mjs`:

```
server.del('/destroy/:username', async (req, res, next) => {
  try {
    await connectDB();
    const user = await SQUser.findOne({
      where: { username: req.params.username } });
    if (!user) {
      res.send(404,
        new Error(`Did not find requested ${req.params.username}
          to delete`));
    } else {
      user.destroy();
      res.contentType = 'json';
      res.send({});
```

```
    }
    next(false);
  } catch(err) {
    res.send(500, err);
    next(false);
  }
});
```

This is simple enough. We first look up the user to ensure it exists, and then call the `destroy` function on the `SQUser` object. There's no need for any result, so we send an empty object.

To exercise this function, add the following code to `cli.mjs`:

```
program
  .command('destroy <username>')
  .description('Destroy a user on the user server')
  .action((username, cmdObj) => {
    client(program).del(`/destroy/${username}`,
      (err, req, res, obj) => {
        if (err) console.error(err.stack);
        else console.log('Deleted - result= ' + util.inspect(obj));
      });
});
```

This is simply to send a `DELETE` request to the server on the `/destroy` URL.

And then, to test it, run the following command:

```
$ node cli.mjs destroy snuffy-smith
Deleted - result= {}
$ node cli.mjs find snuffy-smith
finding snuffy-smith
```

```
NotFoundError: {}
  at Object.createHttpErr
(/home/david/Chapter08/users/node_modules/restify-
clients/lib/helpers/errors.js:91:26)
  at ClientRequest.onResponse
(/home/david/Chapter08/users/node_modules/restify-
clients/lib/HttpClient.js:309:26)
  at Object.onceWrapper (events.js:428:26)
  at ClientRequest.emit (events.js:321:20)
  at HTTPParser.parserOnIncomingClient [as onIncoming]
(_http_client.js:602:27)
  at HTTPParser.parserOnHeadersComplete (_http_common.js:116:17)
  at Socket.socketOnData (_http_client.js:471:22)
  at Socket.emit (events.js:321:20)
  at addChunk (_stream_readable.js:305:12)
  at readableAddChunk (_stream_readable.js:280:11)
```

First, we deleted Snuffy's user record, and it gave us an empty response, as expected. Then, we tried to retrieve his record and, as expected, there was an error.

While we have completed the CRUD operations, we have one final task to cover.

Checking the user's password in the user information service

How can we have a user login/logout service without being able to check their password? The question is: Where should the password check occur? It seems, without examining it too deeply, that it's better to perform this operation inside the user information service. We earlier described the decision that it's probably safer to never expose the user password beyond the user information service. As a result, the password check should occur in that service so that the password does not stray beyond the service.

Let's start with the following function in `user-server.mjs`:

```
server.post('/password-check', async (req, res, next) => {
  try {
    await connectDB();
    const user = await SQUser.findOne({
      where: { username: req.params.username } });
    let checked;
    if (!user) {
      checked = {
        check: false, username: req.params.username,
        message: "Could not find user"
```

```
        };
    } else if (user.username === req.params.username
        && user.password === req.params.password) {
        checked = { check: true, username: user.username };
    } else {
        checked = {
            check: false, username: req.params.username,
            message: "Incorrect password"
        };
    }
    res.contentType = 'json';
    res.send(checked);
    next(false);
} catch(err) {
    res.send(500, err);
    next(false);
}
});
```

This lets us support the checking of user passwords. There are three conditions to check, as follows:

- Whether there is no such user
- Whether the passwords matched
- Whether the passwords did not match

The code neatly determines all three conditions and returns an object indicating, via the `check` field, whether the user is authenticated. The caller is to send `username` and `password` parameters that will be checked.

To check it out, let's add the following code to `cli.mjs`:

```
program
  .command('password-check <username> <password>')
  .description('Check whether the user password checks out')
  .action((username, password, cmdObj) => {
    client(program).post('/password-check', { username, password },
    (err, req, res, obj) => {
      if (err) console.error(err.stack);
      else console.log(obj);
    });
  });
});
```

And, as expected, the code to invoke this operation is simple. We take the `username` and `password` parameters from the command line, send them to the server, and then print the result.

To verify that it works, run the following command:

```
$ node cli.mjs password-check me w0rd
{ check: true, username: 'me' }
$ node cli.mjs password-check me w0rdy
{ check: false, username: 'me', message: 'Incorrect password' }
```

Indeed, the correct password gives us a `true` indicator, while the wrong password gives us `false`.

We've done a lot in this section by implementing a user information service. We successfully created a REST service while thinking about architectural choices around correctly handling sensitive user data. We were also able to verify that the REST service is functioning using an ad hoc testing tool. With this command-line tool, we can easily try any combination of parameters, and we can easily extend it if the need arises to add more REST operations.

Now, we need to start on the real goal of the chapter: changing the Notes user interface to support login/logout. We will see how to do this in the following sections.

Providing login support for the Notes application

Now that we have proved that the user authentication service is working, we can set up the Notes application to support user logins. We'll be using Passport to support login/logout, and the authentication server to store the required data.

Among the available packages, Passport stands out for simplicity and flexibility. It integrates directly with the Express middleware chain, and the Passport community has developed hundreds of so-called strategy modules to handle authentication against a long list of third-party services.



Refer to <http://www.passportjs.org/> for information and documentation.

Let's start this by adding a module for accessing the user information REST server we just created.

Accessing the user authentication REST API

The first step is to create a user data model for the Notes application. Rather than retrieving data from data files or a database, it will use REST to query the server we just created. Recall that we created this REST service in the theory of walling off the service since it contains sensitive user information.

Earlier, we suggested duplicating [Chapter 7, Data Storage and Retrieval](#), code for Notes in the `chap08/notes` directory and creating the user information server as `chap08/users`.

Earlier in this chapter, we used the `restify-clients` module to access the REST service. That package is a companion to the `Restify` library; the `restify` package supports the server side of the REST protocol and `restify-clients` supports the client side.

However nice the `restify-clients` library is, it doesn't support a Promise-oriented API, as is required to play well with `async` functions. Another library, `SuperAgent`, does support a Promise-oriented API and plays well in `async` functions, and there is a companion to that package, `SuperTest`, that's useful in unit testing. We'll use `SuperTest` in [Chapter 13, Unit Testing and Functional Testing](#) when we talk about unit testing.



For documentation, refer to <https://www.npmjs.com/package/superagent> and <http://visionmedia.github.io/superagent/>.

To install the package (again, in the Notes application directory), run the following command:

```
$ npm install superagent@^5.2.x --save
```

Then, create a new file, `models/users-superagent.mjs`, containing the following code:

```
import { default as request } from 'superagent';
import util from 'util';
import url from 'url';
const URL = url.URL;
import DBG from 'debug';
const debug = DBG('notes:users-superagent');
const error = DBG('notes:error-superagent');
```

```
var authid = 'them';
var authcode = 'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF';

function reqURL(path) {
  const requrl = new URL(process.env.USER_SERVICE_URL);
  requrl.pathname = path;
  return requrl.toString();
}
```

The `reqURL` function is similar in purpose to the `connectDB` functions that we wrote in earlier modules. Remember that we used `connectDB` in earlier modules to open a database connection that will be kept open for a long time. With SuperAgent, we don't leave a connection open to the service. Instead, we open a new server connection on each request. For every request, we will formulate the request URL. The base URL, such as `http://localhost:3333/`, is to be provided in the `USER_SERVICE_URL` environment variable. The `reqURL` function modifies that URL, using the new **Web Hypertext Application Technology Working Group (WHATWG)** URL support in Node.js, to use a given URL path.

We also added the authentication ID and code required for the server. Obviously, when the backlog task comes up to use a better token authentication system, this will have to change.

To handle creating and updating user records, run the following code:

```
export async function create(username, password,
    provider, familyName, givenName, middleName,
    emails, photos) {
  var res = await request
    .post(reqURL('/create-user'))
    .send({ username, password, provider,
        familyName, givenName, middleName, emails, photos
    })
    .set('Content-Type', 'application/json')
    .set('Accept', 'application/json')
    .auth(authid, authcode);
  return res.body;
}

export async function update(username, password,
    provider, familyName, givenName, middleName,
    emails, photos) {
  var res = await request
    .post(reqURL(`/update-user/${username}`))
    .send({ username, password, provider,
        familyName, givenName, middleName, emails, photos
    })
    .set('Content-Type', 'application/json')
    .set('Accept', 'application/json')
    .auth(authid, authcode);
  return res.body;
}
```

```
        })
      .set('Content-Type', 'application/json')
      .set('Accept', 'application/json')
      .auth(authid, authcode);
    return res.body;
}
```

These are our `create` and `update` functions. In each case, they take the data provided, construct an anonymous object, and `POST` it to the server. The function is to be provided with the values corresponding to the `SQUser` schema. It bundles the data provided in the `send` method, sets various parameters, and then sets up the Basic Auth token.

The SuperAgent library uses an API style called *method chaining*. The coder chains together method calls to construct a request. The chain of method calls can end in a `.then` or `.end` clause, either of which takes a callback function. But leave off both, and it will return a Promise, and, of course, Promises let us use this directly from an `async` function.

The `res.body` value at the end of each function contains the value returned by the REST server. All through this library, we'll use the `.auth` clause to set up the required authentication key.



These anonymous objects are a little different than normal. We're using a new **ECMAScript 2015 (ES-2015)** feature here that we haven't discussed so far. Rather than specifying the object fields using the `fieldName: fieldValue` notation, ES-2015 gives us the option to shorten this when the variable name used for `fieldValue` matches the desired `fieldName`. In other words, we can just list the variable names, and the field name will automatically match the variable name.

In this case, we've purposely chosen variable names for the parameters to match the field names of the object with parameter names used by the server. In doing so, we can use this shortened notation for anonymous objects, and our code is a little cleaner by using consistent variable names from beginning to end.

Now, add the following function to support the retrieval of user records:

```
export async function find(username) {
  var res = await request
    .get(reqURL(`/find/${username}`))
    .set('Content-Type', 'application/json')
    .set('Accept', 'application/json')
```

```
    .auth(authid, authcode);
    return res.body;
}
```

This is following the same pattern as before. The `set` methods are, of course, used for setting HTTP headers in the REST call. This means having at least a passing knowledge of the HTTP protocol.

The `Content-Type` header says the data sent to the server is in **JavaScript Object Notation (JSON)** format. The `Accept` header says that this REST client can handle JSON data. JSON is, of course, easiest for a JavaScript program—such as what we're writing—to utilize.

Let's now create the function for checking passwords, as follows:

```
export async function userPasswordCheck(username, password) {
    var res = await request
        .post(reqURL('/password-check'))
        .send({ username, password })
        .set('Content-Type', 'application/json')
        .set('Accept', 'application/json')
        .auth(authid, authcode);
    return res.body;
}
```

One point about this method is worth noting. It could have taken the parameters in the URL instead of the request body, as is done here. But since request URLs are routinely logged to files, putting the username and password parameters in the URL means user identity information would be logged to files and be part of activity reports. That would obviously be a very bad choice. Putting those parameters in the request body not only avoids that bad result but if an HTTPS connection to the service were used, the transaction would be encrypted.

Then, let's create our `find-or-create` function, as follows:

```
export async function findOrCreate(profile) {
    var res = await request
        .post(reqURL('/find-or-create'))
        .send({
            username: profile.id, password: profile.password,
            provider: profile.provider,
            familyName: profile.familyName,
            givenName: profile.givenName,
            middleName: profile.middleName,
            emails: profile.emails, photos: profile.photos
        })
}
```

```
        .set('Content-Type', 'application/json')
        .set('Accept', 'application/json')
        .auth(authid, authcode);
    return res.body;
}
```

The `/find-or-create` function either discovers the user in the database or creates a new user. The `profile` object will come from Passport, but take careful note of what we do with `profile.id`. The Passport documentation says it will provide the `username` in the `profile.id` field, but we want to store it as `username` instead.

Let's now create a function to retrieve the list of users, as follows:

```
export async function listUsers() {
    var res = await request
        .get(reqURL('/list'))
        .set('Content-Type', 'application/json')
        .set('Accept', 'application/json')
        .auth(authid, authcode);
    return res.body;
}
```

As before, this is very straightforward.

With this module, we can interface with the user information service, and we can now proceed with modifying the Notes user interface.

Incorporating login and logout routing functions in the Notes application

What we've built so far is a user data model, with a REST API wrapping that model to create our authentication information service. Then, within the Notes application, we have a module that requests user data from this server. As yet, nothing in the Notes application knows that this user model exists. The next step is to create a routing module for login/logout URLs and to change the rest of Notes to use user data.

The routing module is where we use `passport` to handle user authentication. The first task is to install the required modules, as follows:

```
$ npm install passport@^0.4.x passport-local@1.x --save
```

The `passport` module gives us the authentication algorithms. To support different authentication mechanisms, the `passport` authors have developed several *strategy* implementations—the authentication mechanisms, or strategies, corresponding to the various third-party services that support authentication, such as using OAuth to authenticate against services such as Facebook, Twitter, or GitHub.

Passport also requires that we install Express Session support. Use the following command to install the modules:

```
$ npm install express-session@1.17.x session-file-store@1.4.x --save
```



Express Session support, including all the various Session Store implementations, is documented on its GitHub project page at <https://github.com/expressjs/session>.

The strategy implemented in the `passport-local` package authenticates solely using data stored locally to the application—for example, our user authentication information service. Later, we'll add a strategy module to authenticate the use of OAuth with Twitter.

Let's start by creating the routing module, `routes/users.mjs`, as follows:

```
import path from 'path';
import util from 'util';
import { default as express } from 'express';
import { default as passport } from 'passport';
import { default as passportLocal } from 'passport-local';
const LocalStrategy = passportLocal.Strategy;
import * as usersModel from '../models/users-superagent.mjs';
import { sessionCookieName } from '../app.mjs';

export const router = express.Router();

import DBG from 'debug';
const debug = DBG('notes:router-users');
const error = DBG('notes:error-users');
```

This brings in the modules we need for the `/users` router. This includes the two `passport` modules and the REST-based user authentication model.

In `app.mjs`, we will be adding *session* support so our users can log in and log out. That relies on storing a cookie in the browser, and the cookie name is found in this variable exported from `app.mjs`. We'll be using that cookie in a moment.

Add the following functions to the end of `routes/users.mjs`:

```
export function initPassport(app) {
  app.use(passport.initialize());
  app.use(passport.session());
}

export function ensureAuthenticated(req, res, next) {
  try {
    // req.user is set by Passport in the deserialize function
    if (req.user) next();
    else res.redirect('/users/login');
  } catch (e) { next(e); }
}
```

The `initPassport` function will be called from `app.mjs`, and it installs the Passport middleware in the Express configuration. We'll discuss the implications of this later when we get to `app.mjs` changes, but Passport uses sessions to detect whether this HTTP request is authenticated. It looks at every request coming into the application, looks for clues about whether this browser is logged in, and attaches data to the request object as `req.user`.

The `ensureAuthenticated` function will be used by other routing modules and is to be inserted into any route definition that requires an authenticated logged-in user. For example, editing or deleting a note requires the user to be logged in and, therefore, the corresponding routes in `routes/notes.mjs` must use `ensureAuthenticated`. If the user is not logged in, this function redirects them to `/users/login` so that they can log in.

Add the following route handlers in `routes/users.mjs`:

```
router.get('/login', function(req, res, next) {
  try {
    res.render('login', { title: "Login to Notes", user: req.user, });
  } catch (e) { next(e); }
});

router.post('/login',
```

```
    passport.authenticate('local', {
      successRedirect: '/',
      failureRedirect: '/login',
    })
);
```

Because this router is mounted on `/users`, all these routes will have `/user` prepended. The `/users/login` route simply shows a form requesting a username and password. When this form is submitted, we land in the second route declaration, with a POST on `/users/login`. If passport deems this a successful login attempt using `LocalStrategy`, then the browser is redirected to the home page. Otherwise, it is redirected back to the `/users/login` page.

Add the following route for handling logout:

```
router.get('/logout', function(req, res, next) {
  try {
    req.session.destroy();
    req.logout();
    res.clearCookie(sessionCookieName);
    res.redirect('/');
  } catch (e) { next(e); }
});
```

When the user requests to log out of Notes, they are to be sent to `/users/logout`. We'll be adding a button to the header template for this purpose. The `req.logout` function instructs Passport to erase their login credentials, and they are then redirected to the home page.

This function deviates from what's in the Passport documentation. There, we are told to simply call `req.logout`, but calling only that function sometimes results in the user not being logged out. It's necessary to destroy the session object, and to clear the cookie, in order to ensure that the user is logged out. The cookie name is defined in `app.mjs`, and we imported `sessionCookieName` for this function.

Add the `LocalStrategy` to Passport, as follows:

```
passport.use(new LocalStrategy(
  async (username, password, done) => {
    try {
      var check = await usersModel.userPasswordCheck(username,
        password);
      if (check.check) {
        done(null, { id: check.username, username: check.username });
      } else {
        done(null, false, check.message);
      }
    } catch (e) {
      done(e);
    }
  }
));
```

```
        }
    } catch (e) { done(e); }
}
));
```

Here is where we define our implementation of `LocalStrategy`. In the callback function, we call `usersModel.userPasswordCheck`, which makes a REST call to the user authentication service. Remember that this performs the password check and then returns an object indicating whether the user is logged in.

A successful login is indicated when `check.check` is `true`. In this case, we tell Passport to use an object containing `username` in the session object. Otherwise, we have two ways to tell Passport that the login attempt was unsuccessful. In one case, we use `done(null, false)` to indicate an error logging in, and pass along the error message we were given. In the other case, we'll have captured an exception, and pass along that exception.

You'll notice that Passport uses a callback-style API. Passport provides a `done` function, and we are to call that function when we know what's what. While we use an `async` function to make a clean asynchronous call to the backend service, Passport doesn't know how to grok the Promise that would be returned. Therefore, we have to throw a `try/catch` around the function body to catch any thrown exception.

Add the following functions to manipulate data stored in the session cookie:

```
passport.serializeUser(function(user, done) {
    try {
        done(null, user.username);
    } catch (e) { done(e); }
});

passport.deserializeUser(async (username, done) => {
    try {
        var user = await usersModel.find(username);
        done(null, user);
    } catch(e) { done(e); }
});
```

The preceding functions take care of encoding and decoding authentication data for the session. All we need to attach to the session is the `username`, as we did in `serializeUser`. The `deserializeUser` object is called while processing an incoming HTTP request and is where we look up the user profile data. Passport will attach this to the `request` object.

Login/logout changes to app.mjs

A number of changes are necessary in `app.mjs`, some of which we've already touched on. We did carefully isolate the Passport module dependencies to `routes/users.mjs`. The changes required in `app.mjs` support the code in `routes/users.mjs`.

Add an import to bring in functions from the User router module, as follows:

```
import { router as indexRouter } from './routes/index.mjs';
import { router as notesRouter } from './routes/notes.mjs';
import { router as usersRouter, initPassport } from
  './routes/users.mjs';
```

The User router supports the `/login` and `/logout` URLs, as well as using Passport for authentication. We need to call `initPassport` for a little bit of initialization.

And now, let's import modules for session handling, as follows:

```
import session from 'express-session';
import sessionFileStore from 'session-file-store';
const FileStore = sessionFileStore(session);
export const sessionCookieName = 'notescookie.sid';
```

Because Passport uses sessions, we need to enable session support in Express, and these modules do so. The `session-file-store` module saves our session data to disk so that we can kill and restart the application without losing sessions. It's also possible to save sessions to databases with appropriate modules. A filesystem session store is suitable only when all Notes instances are running on the same server computer. For a distributed deployment situation, you'll need to use a session store that runs on a network-wide service, such as a database.

We're defining `sessionCookieName` here so that it can be used in multiple places. By default, `express-session` uses a cookie named `connect.sid` to store the session data. As a small measure of security, it's useful when there's a published default to use a different non-default value. Any time we use the default value, it's possible that an attacker might know a security flaw, depending on that default.

Add the following code to `app.mjs`:

```
app.use(session({
  store: new FileStore({ path: "sessions" }),
  secret: 'keyboard mouse',
  resave: true,
  saveUninitialized: true,
```

```
    name: sessionCookieName
  });
initPassport(app);
```

Here, we initialize the session support. The field named `secret` is used to sign the session ID cookie. The session cookie is an encoded string that is encrypted in part using this secret. In the Express Session documentation, they suggest the `keyboard cat` string for the secret. But, in theory, what if Express has a vulnerability, such that knowing this secret can make it easier to break the session logic on your site? Hence, we chose a different string for the secret, just to be a little different and—perhaps—a little more secure.

Similarly, the default cookie name used by `express-session` is `connect.sid`. Here's where we change the cookie name to a non-default name.

`FileStore` will store its session data records in a directory named `sessions`. This directory will be auto-created as needed.

In case you see errors on Windows that are related to the files used by `session-file-store`, there are several alternate session store packages that can be used. The attraction of the `session-file-store` is that it has no dependency on a service like a database server. Two other session stores have a similar advantage, `LokiStore`, and `MemoryStore`. Both are configured similarly to the `session-file-store` package. For example, to use `MemoryStore`, first use `npm` to install the `memorystore` package, then use these lines of code in `app.mjs`:

```
import sessionMemoryStore from 'memorystore';
const MemoryStore = sessionMemoryStore(session);
...
app.use(session({
  store: new MemoryStore({}),
  secret: 'keyboard mouse',
  resave: true,
  saveUninitialized: true,
  name: sessionCookieName
}));
```

This is the same initialization, but using `MemoryStore` instead of `FileStore`.



To learn more about session store implementations see: <http://expressjs.com/en/resources/middleware/session.html#compatible-session-stores>

Mount the User router, as follows:

```
app.use('/', indexRouter);
app.use('/notes', notesRouter);
app.use('/users', usersRouter);
```

These are the three routers that are used in the Notes application.

Login/logout changes in routes/index.mjs

This router module handles the home page. It does not require the user to be logged in, but we want to change the display a little if they are logged in. To do so, run the following code:

```
router.get('/', async (req, res, next) => {
  try {
    let keylist = await notes.keylist();
    let keyPromises = keylist.map(key => { return notes.read(key) });
    let notelist = await Promise.all(keyPromises);
    res.render('index', {
      title: 'Notes', notelist: notelist,
      user: req.user ? req.user : undefined
    });
  } catch (e) { next(e); }
});
```

Remember that we ensured that `req.user` has the user profile data, which we did in `deserializeUser`. We simply check for this and make sure to add that data when rendering the views template.

We'll be making similar changes to most of the other route definitions. After that, we'll go over the changes to the view templates, in which we use `req.user` to show the correct buttons on each page.

Login/logout changes required in routes/notes.mjs

The changes required here are more significant but still straightforward, as shown in the following code snippet:

```
import { ensureAuthenticated } from './users.mjs';
```

We need to use the `ensureAuthenticated` function to protect certain routes from being used by users who are not logged in. Notice how ES6 modules let us import just the function(s) we require. Since that function is in the User router module, we need to import it from there.

Modify the `/add` route handler, as shown in the following code block:

```
router.get('/add', ensureAuthenticated, (req, res, next) => {
  try {
    res.render('noteedit', {
      title: "Add a Note",
      docreate: true, notekey: "",
      user: req.user, note: undefined
    });
  } catch (e) { next(e); }
});
```

We'll be making similar changes throughout this module, adding calls to `ensureAuthenticated` and using `req.user` to check whether the user is logged in. The goal is for several routes to ensure that the route is only available to a logged-in user, and—in those and additional routes—to pass the `user` object to the template.

The first thing we added is to call `usersRouter.ensureAuthenticated` in the route definition. If the user is not logged in, they'll be redirected to `/users/login` thanks to that function.

Because we've ensured that the user is authenticated, we know that `req.user` will already have their profile information. We can then simply pass it to the view template.

For the other routes, we need to make similar changes.

Modify the `/save` route handler, as follows:

```
router.post('/save', ensureAuthenticated, (req, res, next) => {
  ..
});
```

The `/save` route only requires this change to call `ensureAuthenticated` in order to ensure that the user is logged in.

Modify the `/view` route handler, as follows:

```
router.get('/view', (req, res, next) => {
  try {
    var note = await notes.read(req.query.key);
    res.render('noteview', {
      title: note ? note.title : "",
      notekey: req.query.key,
      user: req.user ? req.user : undefined,
      note: note
    });
  } catch (e) { next(e); }
});
```

For this route, we don't require the user to be logged in. We do need the user's profile information, if any, sent to the view template.

Modify the `/edit` and `/destroy` route handlers, as follows:

```
router.get('/edit', ensureAuthenticated, (req, res, next) => {
  try {
    var note = await notes.read(req.query.key);
    res.render('noteedit', {
      title: note ? ("Edit " + note.title) : "Add a Note",
      docreate: false,
      notekey: req.query.key,
      user: req.user,
      note: note
    });
  } catch (e) { next(e); }
});
router.get('/destroy', ensureAuthenticated, (req, res, next) => {
  try {
    var note = await notes.read(req.query.key);
    res.render('notedestroy', {
      title: note ? `Delete ${note.title}` : "",
      notekey: req.query.key,
      user: req.user,
      note: note
    });
  } catch (e) { next(e); }
});
```

```
    });
    } catch (e) { next(e); }
});
router.post('/destroy/confirm', ensureAuthenticated, (req, res, next)
=> {
  ..
});
```

Remember that throughout this module, we have made the following two changes to router functions:

1. We protected some routes using `ensureAuthenticated` to ensure that the route is available only to logged-in users.
2. We passed the `user` object to the template.

For the routes using `ensureAuthenticated`, it is guaranteed that `req.user` will contain the `user` object. In other cases, such as with the `/view` router function, `req.user` may or may not have a value, and in case it does not, we make sure to pass `undefined`. In all such cases, the templates need to change in order to use the `user` object to detect whether the user is logged in, and whether to show HTML appropriate for a logged-in user.

Viewing template changes supporting login/logout

So far, we've created a backend user authentication service, a REST module to access that service, a router module to handle routes related to logging in and out of the website, and changes in `app.mjs` to use those modules. We're almost ready, but we've got a number of changes left that need to be made to the templates. We're passing the `req.user` object to every template because each one must be changed to accommodate whether the user is logged in.

This means that we can test whether the user is logged in simply by testing for the presence of a `user` variable.

In `partials/header.hbs`, make the following additions:

```
...
<nav class="navbar navbar-expand-md navbar-dark bg-dark">
  <a class="navbar-brand" href="/"><i data-feather="home"></i></a>
  <button class="navbar-toggler" type="button"
    data-toggle="collapse" data-target="#navbarLogIn"
    aria-controls="navbarLogIn"
    aria-expanded="false"
    aria-label="Toggle navigation">
```

```
<span class="navbar-toggler-icon"></span>
</button>
{{#if user}}
<div class="collapse navbar-collapse" id="navbarLogIn">
  <span class="navbar-text text-dark col">{{ title }}</span>
  <a class="btn btn-dark col-auto" href="/users/logout">
    Log Out <span class="badge badge-light">{{ user.username }}</span>
  </a>
  <a class="nav-item nav-link btn btn-dark col-auto" href='/notes/add'>ADD Note</a>
</div>
{{else}}
<div class="collapse navbar-collapse" id="navbarLogIn">
  <a class="btn btn-primary" href="/users/login">Log in</a>
</div>
{{/if}}
</nav>
...
```

What we're doing here is controlling which buttons to display at the top of the screen, depending on whether the user is logged in. The earlier changes ensure that the `user` variable will be `undefined` if the user is logged out; otherwise, it will have the user profile object. Therefore, it's sufficient to check the `user` variable, as shown in the preceding code block, to render different user interface elements.

A logged-out user doesn't get the **ADD Note** button and gets a **Log in** button. Otherwise, the user gets an **ADD Note** button and a **Log Out** button. The **Log in** button takes the user to `/users/login`, while the **Log Out** button takes them to `/users/logout`. Both of those buttons are handled in `routes/users.js` and perform the expected function.

The **Log Out** button has a Bootstrap badge component displaying the username. This adds a little visual splotch in which we'll put the username that's logged in. As we'll see later, it will serve as a visual clue to the user as to their identity.

Because `nav` is now supporting login/logout buttons, we have changed the `navbar-toggler` button so that it controls a `<div>` with `id="navbarLogIn"`.

We need to create `views/login.hbs`, as follows:

```
<div class="container-fluid">
  <div class="row">
    <div class="col-12 btn-group-vertical" role="group">
      <form method='POST' action='/users/login'>
```

```
<div class="form-group">
  <label for="username">User name:</label>
  <input class="form-control" type='text' id='username'
    name='username' value='' placeholder='User Name' />
</div>
<div class="form-group">
  <label for="password">Password:</label>
  <input class="form-control" type='password' id='password'
    name='password' value='' placeholder='Password' />
</div>
<button type="submit" class="btn btn-default">Submit</button>
</form>

</div>
</div>
</div>
```

This is a simple form decorated with Bootstrap goodness to ask for the username and password. When submitted, it creates a POST request to /users/login, which invokes the desired handler to verify the login request. The handler for that URL will start the Passport process to decide whether the user is authenticated.

In views/notedestroy.hbs, we want to display a message if the user is not logged in. Normally, the form to cause the note to be deleted is displayed, but if the user is not logged in, we want to explain the situation, as illustrated in the following code block:

```
<form method='POST' action='/notes/destroy/confirm'>
<div class="container-fluid">
  {{#if user}}
    <input type='hidden' name='notekey' value='{{#if
      note}}{{notekey}}{{/if}}'>
    <p class="form-text">Delete {{note.title}}?</p>

    <div class="btn-group">
      <button type="submit" value='DELETE'
        class="btn btn-outline-dark">DELETE</button>
      <a class="btn btn-outline-dark"
        href="/notes/view?key={{#if note}}{{notekey}}{{/if}}"
        role="button">Cancel</a>
    </div>
  {{else}}
    {{> not-logged-in }}
  {{/if}}
</div>
</form>
```

That's straightforward—if the user is logged in, display the form; otherwise, display the message in `partials/not-logged-in.hbs`. We determine which of these to display based on the `user` variable.

We could insert something such as the code shown in the following block in `partials/not-logged-in.hbs`:

```
<div class="jumbotron">
  <h1>Not Logged In</h1>
  <p>You are required to be logged in for this action, but you are not.
  You should not see this message. It's a bug if this message appears.
  </p>
  <p><a class="btn btn-primary" href="/users/login">Log in</a></p>
</div>
```

As the text says, this will probably never be shown to users. However, it is useful to put something such as this in place since it may show up during development, depending on the bugs you create.

In `views/noteedit.hbs`, we require a similar change, as follows:

```
...
<div class="container-fluid">
{{#if user}}
...
{{else}}
{{> not-logged-in }}
{{/if}}
</div>
...
```

That is, at the bottom we add a segment that, for non-logged-in users, pulls in the `not-logged-in` partial.

The **Bootstrap jumbotron** component makes a nice and large text display that stands out nicely and will catch the viewer's attention. However, the user should never see this because each of those templates is used only when we've pre-verified the fact that the user is logged in.

A message such as this is useful as a check against bugs in your code. Suppose that we slipped up and failed to properly ensure that these forms were displayed only to logged-in users. Suppose that we had other bugs that didn't check the form submission to ensure it's requested only by a logged-in user. Fixing the template in this way is another layer of prevention against displaying forms to users who are not allowed to use that functionality.

We have now made all the changes to the user interface and are ready to test the login/logout functionality.

Running the Notes application with user authentication

We have created the user information REST service, created a module to access that service from Notes, modified the router modules to correctly access the user information service, and changed other things required to support login/logout.

The final task that is necessary is to change the scripts section of package.json, as follows:

```
"scripts": {  
    "start": "cross-env DEBUG=notes:*  
        SEQUELIZE_CONNECT=models/sequelize-  
        sqlite.yaml NOTES_MODEL=sequelize  
        USER_SERVICE_URL=http://localhost:5858  
        node ./app.mjs",  
    "dl-minty": "mkdir -p minty && npm run dl-minty-css && npm run dl-  
        minty-min-css",  
    "dl-minty-css": "wget https://bootswatch.com/4/minty/bootstrap.css  
        -O minty/bootstrap.css",  
    "dl-minty-min-css": "wget  
        https://bootswatch.com/4/minty/bootstrap.min.css  
        -O minty/bootstrap.min.css"  
},
```

In the previous chapters, we built up quite a few combinations of models and databases for running the Notes application. Since we don't need those, we can strip most of them out from package.json. This leaves us with one, configured to use the Sequelize model for Notes, using the SQLite3 database, and to use the new user authentication service that we wrote earlier. All the other Notes data models are still available, just by setting the environment variables appropriately.

USER_SERVICE_URL needs to match the port number that we designated for that service.

In one window, start the user authentication service, as follows:

```
$ cd users  
$ npm start  
> user-auth-server@0.0.1 start /Users/david/chap08/users  
> DEBUG=users:* PORT=5858 SEQUELIZE_CONNECT=sequelize-sqlite.yaml node
```

```
user-server
users:server User-Auth-Service listening at http://127.0.0.1:5858 +0ms
```

Then, in another window, start the Notes application, as follows:

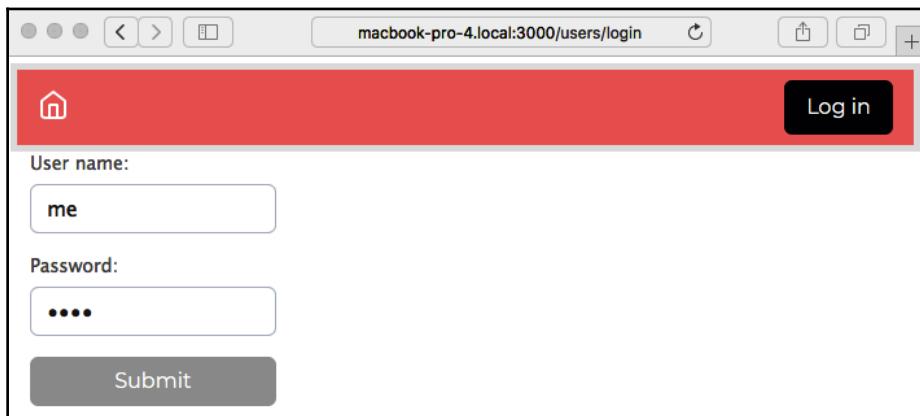
```
$ cd notes
$ DEBUG=notes:* npm start
> notes@0.0.0 start /Users/david/chap08/notes
> cross-env DEBUG=notes:* SEQUELIZE_CONNECT=models/sequelize-
  sqlite.yaml NOTES_MODEL=sequelize
    USER_SERVICE_URL=http://localhost:5858 node ./app.mjs
notes:server Listening on port 3000 +0ms
```

You'll be greeted with the following message:

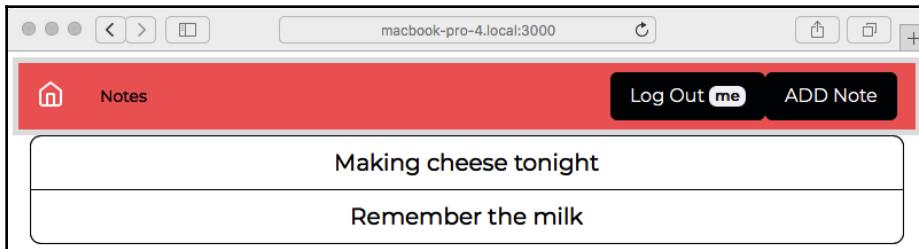


Notice the new button, **Log in**, and the lack of an **ADD Note** button. We're not logged in, and so `partials/header.hbs` is rigged to show only the **Log in** button.

Click on the **Log in** button, and you will see the login screen, as shown in the following screenshot:



This is our login form from `views/login.hbs`. You can now log in, create a note or three, and you might end up with the following messages on the home page:



You now have both **Log Out** and **ADD Note** buttons. You'll notice that the **Log Out** button has the username **(me)** shown. After some thought and consideration, this seemed the most compact way to show whether the user is logged in, and which user is logged in. This might drive the user experience team nuts, and you won't know whether this user interface design works until it's tested with users, but it's good enough for our purpose at the moment.

In this section, we've learned how to set up a basic login/logout functionality using locally stored user information. This is fairly good, but many web applications find it useful to allow folks to log in using their Twitter or other social media accounts for authentication. In the next section, we'll learn about that by setting up Twitter authentication.

Providing Twitter login support for the Notes application

If you want your application to hit the big time, it's a great idea to ease the registration process by using third-party authentication. Websites all over the internet allow you to log in using accounts from other services such as Facebook or Twitter. Doing so removes hurdles to prospective users signing up for your service. Passport makes it extremely easy to do this.

Authenticating users with Twitter requires installation of `TwitterStrategy` from the `passport-twitter` package, registering a new application with Twitter, adding a couple of routes to `routes/user.mjs`, and making a small change in `partials/header.hbs`. Integrating other third-party services requires similar steps.

Registering an application with Twitter

Twitter, as with every other third-party service, uses OAuth to handle authentication. OAuth is a standard protocol through which an application or a person can authenticate with one website by using credentials they have on another website. We use this all the time on the internet. For example, we might use an online graphics application such as `draw.io` or Canva by logging in with a Google account, and then the service can save files to our Google Drive.

Any application author must register with any sites you seek to use for authentication. Since we wish to allow Twitter users to log in to Notes using Twitter credentials, we have to register our Notes application with Twitter. Twitter then gives us a pair of authentication keys that will validate the Notes application with Twitter. Any application, whether it is a popular site such as Canva, or a new site such as Joe's Ascendant Horoscopes, must be registered with any desired OAuth authentication providers. The application author must then be diligent about keeping the registration active and properly storing the authentication keys.

The authentication keys are like a username/password pair. Anyone who gets a hold of those keys could use the service as if they were you, and potentially wreak havoc on your reputation or business.

Our task in this section is to register a new application with Twitter, fulfilling whatever requirements Twitter has.



To register a new application with Twitter, go to <https://developer.twitter.com/en/apps>.

As you go through this process, you may be shown the following message:

Please apply for a Twitter developer account X

You may continue managing your existing apps but if you would like to create new apps or use Twitter premium APIs, please apply for a developer account.

As a developer platform, our first responsibility is to our users: to provide a place that supports the health of conversation on Twitter. To continue to prevent misuse of our platform, we have introduced a few new requirements for developers.

Apply

Recall that in recent years, concerns began to arise regarding the misuse of third-party authentication, the potential to steal user information, and the negative results that have occurred thanks to user data being stolen from social networks. As a result, social networks have increased scrutiny over developers using their APIs. It is necessary to sign up for a Twitter developer account, which is an easy process that does not cost anything.

As we go through this, realize that the Notes application needs a minimal amount of data. The ethical approach to this is to request only the level of access required for your application, and nothing more.

Once you're registered, you can log in to `developer.twitter.com/apps` and see a dashboard listing the active applications you've registered. At this point, you probably do not have any registered applications. At the top is a button marked **Create an App**. Click on that button to start the process of submitting a request to register a new application.

Every service offering OAuth authentication has an administrative backend similar to `developer.twitter.com/apps`. The purpose is so that certified application developers can administer the registered applications and authorization tokens. Each such service has its own policies for validating that those requesting authorization tokens have a legitimate purpose and will not abuse the service. The authorization token is one of the mechanisms to verify that API requests come from approved applications. Another mechanism is the URL from which API requests are made.

In the normal case, an application will be deployed to a regular server, and is accessed through a domain name such as `MyNotes.xyz`. In our case, we are developing a test application on our laptop, and do not have a public IP address, nor is there a domain name associated with our laptop. Not all social networks allow interactions from an application on an untrusted computer—such as a developer's laptop—to make API requests; however, Twitter does.

At the time of writing, there are several pieces of information requested by the Twitter sign-up process, listed as follows:

- **Name:** This is the application name, and it can be anything you like. It would be a good form to use "Test" in the name, in case Twitter's staff decide to do some checking.
- **Description:** Descriptive phrase—and again, it can be anything you like. The description is shown to users during the login process. It's good form to describe this as a test application.

- **Website:** This would be your desired domain name. Here, the help text helpfully suggests *If you don't have a URL yet, just put a placeholder here but remember to change it later.*
- **Allow this application to be used to sign in with Twitter:** Check this, as it is what we want.
- **Callback URL:** This is the URL to return to following successful authentication. Since we don't have a public URL to supply, this is where we specify a value referring to your laptop. It's been found that `http://localhost:3000` works just fine. macOS users have another option because of the `.local` domain name that is automatically assigned to their laptop.
- **Tell us how this app will be used:** This statement will be used by Twitter to evaluate your request. For the purpose of this project, explain that it is a sample app from a book. It is best to be clear and honest about your intention.

The sign-up process is painless. However, at several points, Twitter reiterated the sensitivity of the information provided through the Twitter API. The last step before granting approval warned that Twitter prohibits the use of its API for various unethical purposes.

The last thing to notice is the extremely sensitive nature of the authentication keys. It's bad form to check these into a source code repository or otherwise put them in a place where anybody can access the key. We'll tackle this issue in Chapter 14, *Security in Node.js Applications*.



The Twitter developers' site has documentation describing best practices for storing authentication tokens. Visit <https://developer.twitter.com/en/docs/basics/authentication/guides/authentication-best-practices>.

Storing authentication tokens

The Twitter recommendation is to store configuration values in a `.env` file. The contents of this file are to somehow become environment variables, which we can then access using `process.env`, as we've done before. Fortunately, there is a third-party Node.js package to do just this, called `dotenv`.



Learn about the `dotenv` package at <https://www.npmjs.com/package/dotenv>.

First, install the package, as follows:

```
$ npm install dotenv@8.2.x --save
```

The documentation says we should load the `dotenv` package and then call `dotenv.config()` very early in the start up phase of our application, and that we must do this before accessing any environment variables. However, reading the documentation more closely, it seems best to add the following code to `app.mjs`:

```
import dotenv from 'dotenv/config.js';
```

With this approach, we do not have to explicitly call the `dotenv.config` function. The primary advantage is avoiding issues with referencing environment variables from multiple modules.

The next step is to create a file, `.env`, in the `notes` directory. The syntax of this file is very simple, as shown in the following code block:

```
VARIABLE1=value for variable 1  
VARIABLE2=value for variable 2
```

This is exactly the syntax we'd expect since it is the same as for shell scripts. In this file, we need two variables to be defined, `TWITTER_CONSUMER_KEY` and `TWITTER_CONSUMER_SECRET`. We will use these variables in the code we'll write in the next section. Since we are putting configuration values in the `scripts` section of `package.json`, feel free to add those environment variables to `.env` as well.

The next step is to avoid committing this file to a source code control system such as Git. To ensure that this does not happen, you should already have a `.gitignore` file in the `notes` directory, and make sure its contents are something like this:

```
notes-fs-data  
notes.level  
chap07.sqlite3  
notes-sequelize.sqlite3  
package-lock.json  
data  
node_modules  
.env
```

These values mostly refer to database files we generated in the previous chapter. In the end, we've added the `.env` file, and because of this, Git will not commit this file to the repository.

This means that when deploying the application to a server, you'll have to arrange to add this file to the deployment without it being committed to a source repository.

With an approved Twitter application, and with our authentication tokens recorded in a configuration file, we can move on to adding the required code to Notes.

Implementing TwitterStrategy

As with many web applications, we have decided to allow our users to log in using Twitter credentials. The OAuth protocol is widely used for this purpose and is the basis for authentication on one website using credentials maintained by another website.

The application registration process you just followed at `developer.twitter.com` generated for you a pair of API keys: a consumer key, and a consumer secret. These keys are part of the OAuth protocol and will be supplied by any OAuth service you register with, and the keys should be treated with the utmost care. Think of them as the username and password your service uses to access the OAuth-based service (Twitter et al.). The more people who can see these keys, the more likely it becomes that a miscreant can see them and then cause trouble. Anybody with those secrets can access the service API as if they are you.

Let's install the package required to use `TwitterStrategy`, as follows:

```
$ npm install passport-twitter@1.x --save
```

In `routes/users.mjs`, let's start making some changes, as follows:

```
import passportTwitter from 'passport-twitter';
const TwitterStrategy = passportTwitter.Strategy;
```

This imports the package, and then makes its `Strategy` variable available as `TwitterStrategy`.

Let's now install the `TwitterStrategy`, as follows:

```
const twittercallback = process.env.TWITTER_CALLBACK_HOST
  ? process.env.TWITTER_CALLBACK_HOST
  : "http://localhost:3000";
export var twitterLogin;
```

```
if (typeof process.env.TWITTER_CONSUMER_KEY !== 'undefined' && process.env.TWITTER_CONSUMER_KEY !== '' && typeof process.env.TWITTER_CONSUMER_SECRET !== 'undefined' && process.env.TWITTER_CONSUMER_SECRET !== '') {
  passport.use(new TwitterStrategy({
    consumerKey: process.env.TWITTER_CONSUMER_KEY,
    consumerSecret: process.env.TWITTER_CONSUMER_SECRET,
    callbackURL: `${twittercallback}/users/auth/twitter/callback`
  },
  async function(token, tokenSecret, profile, done) {
    try {
      done(null, await usersModel.findOrCreate({
        id: profile.username, username: profile.username, password: '',
        provider: profile.provider, familyName: profile.displayName,
        givenName: "", middleName: "",
        photos: profile.photos, emails: profile.emails
      }));
    } catch(err) { done(err); }
  }));
}

twitterLogin = true;
} else {
  twitterLogin = false;
}
```

This registers a TwitterStrategy instance with passport, arranging to call the user authentication service as users register with the Notes application. This callback function is called when users successfully authenticate using Twitter.

If the environment variables containing the Twitter tokens are not set, then this code does not execute. Clearly, it would be an error to set up Twitter authentication without the keys, so we avoid the error by not executing the code.

To help other code know whether Twitter support is enabled, we export a flag variable - `twitterLogin`.

We defined the `usersModel.findOrCreate` function specifically to handle user registration from third-party services such as Twitter. Its task is to look for the user described in the profile object and, if that user does not exist, to create that user account in Notes.



The `consumerKey` and `consumerSecret` values are supplied by Twitter, after you've registered your application. These secrets are used in the OAuth protocol as proof of identity to Twitter.

The `callbackURL` setting in the `TwitterStrategy` configuration is a holdover from Twitter's OAuth1-based API implementation. In OAuth1, the callback URL was passed as part of the OAuth request. Since `TwitterStrategy` uses Twitter's OAuth1 service, we have to supply the URL here. We'll see in a moment where that URL is implemented in Notes.

The `callbackURL`, `consumerKey`, and `consumerSecret` settings are all injected using environment variables. Earlier, we discussed how it is a best practice to not commit the values for `consumerKey` and `consumerSecret` to a source repository, and therefore we set up the `dotenv` package and a `.env` file to hold those configuration values. In Chapter 10, *Deploying Node.js Applications to Linux Servers*, we'll see that these keys can be declared as environment variables in a Dockerfile.

Add the following route declaration:

```
router.get('/auth/twitter', passport.authenticate('twitter'));
```

To start the user logging in with Twitter, we'll send them to this URL. Remember that this URL is really `/users/auth/twitter` and, in the templates, we'll have to use that URL. When this is called, the `passport` middleware starts the user authentication and registration process using `TwitterStrategy`.

Once the user's browser visits this URL, the OAuth dance begins. It's called a dance because the OAuth protocol involves carefully designed redirects between several websites. `Passport` sends the browser over to the correct URL at Twitter, where Twitter asks the user whether they agree to authenticate using Twitter, and then Twitter redirects the user back to your callback URL. Along the way, specific tokens are passed back and forth in a very carefully designed dance between websites.

Once the OAuth dance concludes, the browser lands at the URL designated in the following router declaration:

```
router.get('/auth/twitter/callback',
  passport.authenticate('twitter', { successRedirect: '/',
    failureRedirect: '/users/login' }));
```

This route handles the callback URL, and it corresponds to the `callbackURL` setting configured earlier. Depending on whether it indicates a successful registration, `Passport` will redirect the browser to either the home page or back to the `/users/login` page.

Because `router` is mounted on `/user`, this URL is actually `/user/auth/twitter/callback`. Therefore, the full URL to use in configuring the `TwitterStrategy`, and to supply to Twitter, is `http://localhost:3000/user/auth/twitter/callback`.

In the process of handling the callback URL, Passport will invoke the callback function shown earlier. Because our callback uses the `usersModel.findOrCreate` function, the user will be automatically registered if necessary.

We're almost ready, but we need to make a couple of small changes elsewhere in Notes.

In `partials/header.hbs`, make the following changes to the code:

```
...
{{else}}
<div class="collapse navbar-collapse" id="navbarLogIn">
  <span class="navbar-text text-dark col"></span>
  <a class="nav-item nav-link btn btn-dark col-auto"
    href="/users/login">
    Log in</a>
  {{#if twitterLogin}}
    <a class="nav-item nav-link btn btn-dark col-auto"
      href="/users/auth/twitter">
      
      Log in with Twitter</a>
  {{/if}}
</div>
{{/if}}
```

This adds a new button that, when clicked, takes the user to `/users/auth/twitter`, which—of course—kicks off the Twitter authentication process. The button is enabled only if Twitter support is enabled, as determined by the `twitterLogin` variable. This means that the router functions must be modified to pass in this variable.



This button includes a little image we downloaded from the official Twitter brand assets page at <https://about.twitter.com/company/brand-assets>. Twitter recommends using these branding assets for a consistent look across all services using Twitter. Download the whole set, and then pick the one you like.

For the URL shown here, the corresponding project directory is named `public/assets/vendor/twitter`. Notice that we force the size to be small enough for the navigation bar.

In `routes/index.mjs`, make the following change:

```
...
import { twitterLogin } from './users.mjs';
...
router.get('/', async (req, res, next) => {
    ...
        res.render('index', {
            title: 'Notes', notelist: notelist,
            user: req.user ? req.user : undefined,
            twitterLogin: twitterLogin
        });
    ...
});
```

This imports the variable, and then, in the data passed to `res.render`, we add this variable. This will ensure that the value reaches `partials/header.hbs`.

In `routes/notes.mjs`, we have a similar change to make in several router functions:

```
...
import { twitterLogin } from './users.mjs';
...
router.get('/add', ensureAuthenticated, (req, res, next) => {
    res.render('noteedit', {
        ... twitterLogin: twitterLogin, ...
    });
});

router.get('/view', (req, res, next) => {
    res.render('noteview', {
        ... twitterLogin: twitterLogin, ...
    });
});
```

```
router.get('/edit', ensureAuthenticated, (req, res, next) => {
  res.render('noteedit', {
    ... twitterLogin: twitterLogin,
  });
});

router.get('/destroy', ensureAuthenticated, (req, res, next) => {
  res.render('notedestroy', {
    ... twitterLogin: twitterLogin,
  });
});
```

This is the same change, importing the variable and passing it to `res.render`.

With these changes, we're ready to try logging in with Twitter.

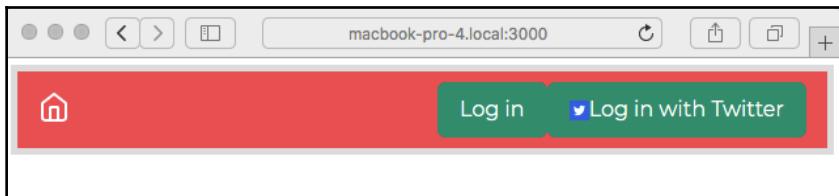
Start the user information server as shown previously, and then start the Notes application server, as shown in the following code block:

```
$ npm start

> notes@0.0.0 start /Users/David/chap08/notes
> DEBUG=notes:* SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml
NOTES_MODEL=sequelize USER_SERVICE_URL=http://localhost:5858 node --experimental-modules ./app.mjs

notes:server-debug Listening on port 3000 +0ms
```

Then, use a browser to visit `http://localhost:3000`, as follows:



Notice the new button. It looks about right, thanks to having used the official Twitter branding image. The button is a little large, so maybe you want to consult a designer. Obviously, a different design is required if you're going to support dozens of authentication services.

Run it while leaving out the Twitter token environment variables, and the Twitter login button should not appear.

Clicking on this button takes the browser to `/users/auth/twitter`, which is meant to start Passport running the OAuth protocol transactions necessary to authenticate. Instead, you may receive an error message that states **Callback URL not approved for this client application. Approved callback URLs can be adjusted in your application settings.** If this is the case, it is necessary to adjust the application configuration on `developer.twitter.com`. The error message is clearly saying that Twitter saw a URL being used that was not approved.

On the page for your application, on the **App Details** tab, click the **Edit** button. Then, scroll down to the **Callback URLs** section and add the following entries:

Callback URLs (required) ①

OAuth 1.0a applications should specify their oauth_callback URL on the request token step, which must match the URLs provided here. To restrict your application from using callbacks, leave these blank.

X

X

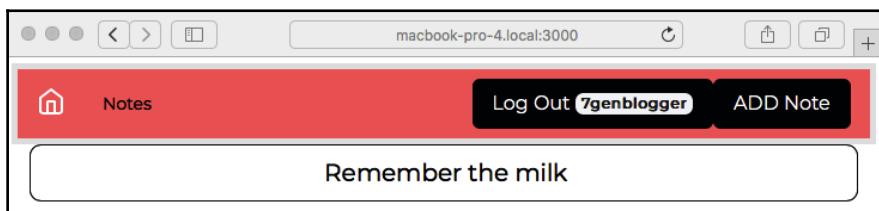
X

[+ Add another](#)

As it explains, this box lists the URLs that are allowed to be used for Twitter OAuth authentication. At the moment, we are hosting the application on our laptop using port 3000. If you are accessing it from other base URLs, such as `http://MacBook-Pro-4.local`, then that base URL should be used in addition.

Once you have the callback URLs correctly configured, clicking on the **Login with Twitter** button will take you to a normal Twitter OAuth authentication page. Simply click for approval, and you'll be redirected back to the Notes application.

And then, once you're logged in with Twitter, you'll see something like the following screenshot:



We're now logged in, and will notice that our Notes username is the same as our Twitter username. You can browse around the application and create, edit, or delete notes. In fact, you can do this to any note you like, even ones created by others. That's because we did not create any sort of access control or permissions system, and therefore every user has complete access to every note. That's a feature to put on the backlog.

By using multiple browsers or computers, you can simultaneously log in as different users, one user per browser.

You can run multiple instances of the Notes application by doing what we did earlier, as follows:

```
"scripts": {
  "start": "cross-env DEBUG=notes:*
SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml
NOTES_MODEL=models/notes-sequelize USERS_MODEL=models/users-rest
USER_SERVICE_URL=http://localhost:5858 node ./bin/www",
  "start-server1": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml
NOTES_MODEL=models/notes-sequelize USERS_MODEL=models/users-rest
USER_SERVICE_URL=http://localhost:5858 PORT=3000 node ./bin/www",
  "start-server2": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml
NOTES_MODEL=models/notes-sequelize USERS_MODEL=models/users-rest
USER_SERVICE_URL=http://localhost:5858 PORT=3002 node ./bin/www",
  "dl-minty": "mkdir -p minty && npm run dl-minty-css && npm run dl-
minty-min-css",
  "dl-minty-css": "wget https://bootswatch.com/4/minty/bootstrap.css
-O minty/bootstrap.css",
  "dl-minty-min-css": "wget
https://bootswatch.com/4/minty/bootstrap.min.css -O
minty/bootstrap.min.css"
},
}
```

Then, in one command window, run the following command:

```
$ npm run start-server1

> notes@0.0.0 start-server1 /Users/David/chap08/notes
> DEBUG=notes:* SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml
NOTES_MODEL=sequelize USER_SERVICE_URL=http://localhost:5858 PORT=3000
node --experimental-modules ./app.mjs

notes:server-debug Listening on port 3000 +0ms
```

In another command window, run the following command:

```
$ npm run start-server2

> notes@0.0.0 start-server2 /Users/David/chap08/notes
> DEBUG=notes:* SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml
NOTES_MODEL=sequelize USER_SERVICE_URL=http://localhost:5858 PORT=3002
node --experimental-modules ./app.mjs

notes:server-debug Listening on port 3002 +0ms
```

As previously, this starts two instances of the Notes server, each with a different value in the PORT environment variable. In this case, each instance will use the same user authentication service. As shown here, you'll be able to visit the two instances at `http://localhost:3000` and `http://localhost:3002`. As before, you'll be able to start and stop the servers as you wish, see the same notes in each, and see that the notes are retained after restarting the server.

Another thing to try is to fiddle with the **session store**. Our session data is being stored in the `sessions` directory. These are just files in the filesystem, and we can take a look with normal tools such as `ls`, as shown in the following code block:

```
$ ls -l sessions/
total 32
-rw-r--r-- 1 david wheel 139 Jan 25 19:28 -
QOS7eXZBAfmK9CCV8Xj8v-3DVETaLK.json
-rw-r--r-- 1 david wheel 139 Jan 25 21:30
T7VT4xt3_e9BiU49OMC6RjbJi6xB7VqG.json
-rw-r--r-- 1 david wheel 223 Jan 25 19:27
ermh-7ijiqY7XXMnA6zPzJvsVsWUghWm.json
-rw-r--r-- 1 david wheel 139 Jan 25 21:23
uKzkXKuJ8uMN_ROEfaRSmvPU7NmBc3md.json $ cat
sessions/T7VT4xt3_e9BiU49OMC6RjbJi6xB7VqG.json
{"cookie": {"originalMaxAge": null, "expires": null, "httpOnly": true, "path": "/"}, "__lastAccess": 1516944652270, "passport": {"user": "7genblogger"}}
```

This is after logging in using a Twitter account. You can see that the Twitter account name is stored here in the session data.

What if you want to clear a session? It's just a file in the filesystem. Deleting the session file erases the session, and the user's browser will be forcefully logged out.

The session will time out if the user leaves their browser idle for long enough. One of the `session-file-store` options, `ttl`, controls the timeout period, which defaults to 3,600 seconds (an hour). With a timed-out session, the application reverts to a logged-out state.

In this section, we've gone through the full process of setting up support for login using Twitter's authentication service. We created a Twitter developer account and created an application on Twitter's backend. Then, we implemented the required workflow to integrate with Twitter's OAuth support. To support this, we integrated the storage of user authorizations from Twitter in the user information service.

Our next task is extremely important: to keep user passwords encrypted.

Keeping secrets and passwords secure

We've cautioned several times about the importance of safely handling user identification information. The intention to handle that data safely is one thing, but it is important to follow through and actually do so. While we're using a few good practices so far, as it stands, the Notes application would not withstand any kind of security audit for the following reasons:

- User passwords are kept in clear text in the database.
- The authentication tokens for Twitter et al. are in clear text.
- The authentication service API key is not a cryptographically secure anything; it's just a clear text **universally unique identifier (UUID)**.

If you don't recognize the phrase *clear text*, it simply means unencrypted. Anyone could read the text of user passwords or the authentication tokens. It's best to keep both encrypted to avoid information leakage.

Keep this issue in the back of your mind because we'll revisit these—and other—security issues in Chapter 14, *Security in Node.js Applications*.

Before we leave this chapter, let's fix the first of those issues: storing passwords in plain text. We made the case earlier that user information security is extremely important. Therefore, we should take care of this from the beginning.

The `bcrypt` Node.js package makes it easy to securely store passwords. With it, we can easily encrypt the password right away, and never store an unencrypted password.



For `bcrypt` documentation, refer to <https://www.npmjs.com/package/bcrypt>.

To install `bcrypt` in both the `notes` and `users` directories, execute the following command:

```
$ npm install bcrypt@5.x --save
```

The `bcrypt` documentation says that the correct version of this package must be used precisely for the Node.js version in use. Therefore, you should adjust the version number appropriately to the Node.js version you are using.

The strategy of storing an encrypted password dates back to the earliest days of Unix. The creators of the Unix operating system devised a means for storing an encrypted value in `/etc/passwd`, which was thought sufficiently safe that the password file could be left readable to the entire world.

Let's start with the user information service.

Adding password encryption to the user information service

Because of our command-line tool, we can easily test end-to-end password encryption. After verifying that it works, we can implement encryption in the Notes application.

In `cli.mjs`, add the following code near the top:

```
import { default as bcrypt } from 'bcrypt';
const saltRounds = 10;
```

This brings in the `bcrypt` package, and then we configure a constant that governs the CPU time required to decrypt a password. The `bcrypt` documentation points to a blog post discussing why the algorithm of `bcrypt` is excellent for storing encrypted passwords. The argument boils down to the CPU time required for decryption. A brute-force attack against the password database is harder, and therefore less likely to succeed if the passwords are encrypted using strong encryption, because of the CPU time required to test all password combinations.

The value we assign to `saltRounds` determines the CPU time requirement. The documentation explains this further.

Next, add the following function:

```
async function hashpass(password) {
  let salt = await bcrypt.genSalt(saltRounds);
```

```
    let hashed = await bcrypt.hash(password, salt);
    return hashed;
}
```

This takes a plain text password and runs it through the encryption algorithm. What's returned is the hash for the password.

Next, in the commands for `add`, `find-or-create`, and `update`, we make this same change, as follows:

```
.action(async (username, cmdObj) => {
  const topost = {
    username,
    password: await hashpass(cmdObj.password),
    ...
  };
  ...
})
```

That is, in each, we make the callback function an `async` function so that we can use `await`. Then, we call the `hashpass` function to encrypt the password.

This way, we are encrypting the password right away, and the user information server will be storing an encrypted password.

Therefore, in `user-server.mjs`, the `password-check` handler must be rewritten to accommodate checking an encrypted password.

At the top of `user-server.mjs`, add the following import:

```
import { default as bcrypt } from 'bcrypt';
```

Of course, we need to bring in the module here to use its decryption function. This module will no longer store a plain text password, but instead, it will now store encrypted passwords. Therefore, it does not need to generate encrypted passwords, but the `bcrypt` package also has a function to compare a plain text password against the encrypted one in the database, which we will use.

Next, scroll down to the `password-check` handler and modify it, like so:

```
server.post('/password-check', async (req, res, next) => {
  try {
    const user = await SQUUser.findOne({
      where: { username: req.params.username } });
    let checked;
    if (!user) {
      checked = {
```

```
        check: false, username: req.params.username,
        message: "Could not find user"
    };
} else {
    let pwcheck = false;
    if (user.username === req.params.username) {
        pwcheck = await bcrypt.compare(req.params.password,
            user.password);
    }
    if (pwcheck) {
        checked = { check: true, username: user.username };
    } else {
        checked = {
            check: false, username: req.params.username,
            message: "Incorrect username or password"
        };
    }
}
...
} catch (e) { .. }
});
```

The `bcrypt.compare` function compares a plain text password, which will be arriving as `req.params.password`, against the encrypted password that we've stored. To handle encryption, we needed to refactor the checks, but we are testing for the same three conditions. And, more importantly, this returns the same objects for those conditions.

To test it, start the user information server as we've done before, like this:

```
$ npm start

> user-auth-server@1.0.0 start /home/david/Chapter08/users
> DEBUG=users:* PORT=5858 SEQUELIZE_CONNECT=sequelize-sqlite.yaml node
./user-server.mjs

  users:service User-Auth-Service listening at http://127.0.0.1:5858
+0ms
```

In another window, we can create a new user, as follows:

```
$ node cli.mjs add --password w0rd --family-name Einarsson --given-
name Ashildr --email me@stolen.tardis.me
Created {
  id: 'me',
  username: 'me',
  provider: 'local',
  familyName: 'Einarsson',
```

```
givenName: 'Ashildr',
middleName: null,
emails: [ 'me@stolen.tardis' ],
photos: []
}
```

We've done both these steps before. Where it differs is what we do next.

Let's check the database to see what was stored, as follows:

```
$ sqlite3 users-sequelize.sqlite3
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
sqlite> select * from SQUsers;
1|me|$2b$10$stjR1KjS1QVTigPkRmRfnOhN7uDnPA56db01UTgip8E6/n4PP7Jje|loc
1|Einarsdottir|Ashildr||["me@stolen.tardis"]||[]|2020-02-05
20:59:21.042 +00:00|2020-02-05 20:59:21.042 +00:00
sqlite> ^D
```

Indeed, the password field no longer has a plain text password, but what is—surely—encrypted text.

Next, we should check that the `password-check` command behaves as expected:

```
$ node cli.mjs password-check me w0rd
{ check: true, username: 'me' }
$ node cli.mjs password-check me w0rdy
{
  check: false,
  username: 'me',
  message: 'Incorrect username or password'
}
```

We performed this same test earlier, but this time, it is against the encrypted password.

We have verified that a REST call to check the password will work. Our next step is to implement the same changes in the Notes application.

Implementing encrypted password support in the Notes application

Since we've already proved how to implement encrypted password checking, all we need to do is duplicate some code in the Notes server.

In `users-superagent.mjs`, add the following code to the top:

```
import { default as bcrypt } from 'bcrypt';
const saltRounds = 10;

async function hashpass(password) {
  let salt = await bcrypt.genSalt(saltRounds);
  let hashed = await bcrypt.hash(password, salt);
  return hashed;
}
```

As before, this imports the `bcrypt` package and configures the complexity that will be used, and we have the same encryption function because we will use it from multiple places.

Next, we must change the functions that interface with the backend server, as follows:

```
export async function create(username, password,
    provider, familyName, givenName, middleName, emails, photos)
{
  var res = await request.post(reqURL('/create-user')).send({
    username, password: await hashpass(password), provider,
    familyName, givenName, middleName, emails, photos
  })
  ...
}

export async function update(username, password,
    provider, familyName, givenName, middleName, emails, photos)
{
  var res = await request.post(reqURL(`/update-user/${username}`))
  .send({
    username, password: await hashpass(password), provider,
    familyName, givenName, middleName, emails, photos
  })
  ...
}

export async function findOrCreate(profile) {
  var res = await request.post(reqURL('/find-or-create')).send({
```

```
username: profile.id,  
password: await hashpass(profile.password),  
...  
})  
...  
}
```

In those places where it is appropriate, we must encrypt the password. No other change is required.

Because the password-check backend performs the same checks, returning the same object, no change is required in the frontend code.

To test, start both the user information server and the Notes server. Then, use the application to check logging in and out with both a Twitter-based user and a local user.

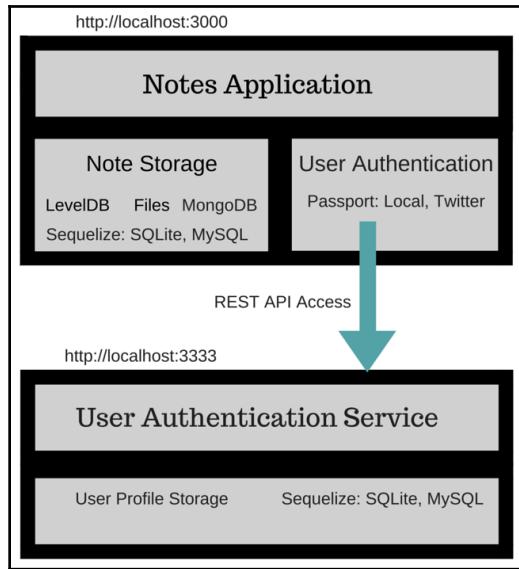
We've learned how to use encryption to safely store user passwords. If someone steals our user database, cracking the passwords will take longer thanks to the choices made here.

We're almost done with this chapter. The remaining task is simply to review the application architecture we've created.

Running the Notes application stack

Did you notice earlier when we said to run the Notes application stack? It's time to explain to the marketing team what's meant by that phrase. They may want to put an architecture diagram on marketing brochures or websites. It's also useful for developers such as us to take a step back and draw a picture of what we've created, or are planning to create.

Here's the sort of diagram that an engineer might draw to show the marketing team the system design (the marketing team will, of course, hire a graphics artist to clean it up):



The box labeled **Notes Application** in the preceding diagram is the public-facing code implemented by the templates and the router modules. As currently configured, it's visible from our laptop on port 3000. It can use one of several data storage services. It communicates with the **User Authentication Service** backend over port 5858 (or port 3333, as shown in the preceding diagram).

In Chapter 10, *Deploying Node.js Applications to Linux Servers*, we'll be expanding this picture a bit as we learn how to deploy on a real server.

Summary

You've covered a lot of ground in this chapter, looking at not only user authentication in Express applications, but also microservices development.

Specifically, you covered session management in Express, using Passport for user authentication—including Twitter/OAuth, using router middleware to limit access, creating a REST service with Restify, and when to create a microservice. We've even used an encryption algorithm to ensure that we only store encrypted passwords.

Knowing how to handle login/logout, especially OAuth login from third-party services, is an essential skill for web application developers. Now that you've learned this, you'll be able to do the same for your own applications.

In the next chapter, we'll take the Notes application to a new level with semi-real-time communication between application users. To do this, we'll write some browser-side JavaScript and explore how the Socket.io package can let us send messages between users.

9

Dynamic Client/Server Interaction with Socket.IO

The original design model of the web is similar to the way that mainframes worked in the 1970s. Both old-school dumb terminals, such as the IBM 3270, and web browsers follow a request-response paradigm. The user sends a request and the far-off computer sends a response. That request-response paradigm is evident in the Node.js HTTP Server API, as shown in the following code:

```
http.createServer(function (request, response) {  
  ... handle request  
}).listen();
```

The paradigm couldn't be more explicit than this. The `request` and the `response` are right there.

It wasn't until JavaScript improved that we had a quite different paradigm. The new paradigm is interactive communication driven by browser-side JavaScript. This change in the web application model is called, by some, the real-time web. In some cases, websites keep an open connection to the web browser, send notifications, or update the page as it changes.



For some deep background on this, read about the Comet application architecture introduced by Alex Russell in his blog in 2006 (<http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/>). That blog post called for a platform very similar to Node.js, years before Node.js existed.

In this chapter, we'll explore interactive dynamically updated content, as well as inter-user messaging, in the Notes application. To do this, we'll lean on the Socket.IO library (<http://socket.io/>). This library simplifies two-way communication between the browser and server and can support a variety of protocols with fallback to old-school web browsers. It keeps a connection open continuously between browser and server, and it follows the `EventEmitter` model, allowing us to send events back and forth.

We'll be covering the following topics:

- An introduction to the Socket.IO library
- Integrating Socket.IO with an Express application, and with Passport
- Real-time communications in modern web browsers
- Using Socket.IO events:
 - To update application content as it changes
 - To send messages between users
- User experience for real-time communication
- Using Modal windows to support a user experience that eliminates page reloads

These sorts of techniques are widely used in many kinds of websites. This includes online chat with support personnel, dynamically updated pricing on auction sites, and dynamically updated social network sites.

To get started, let's talk about what Socket.IO is and what it does.

Introducing Socket.IO

The aim of Socket.IO is to make real-time apps possible in every browser and mobile device. It supports several transport protocols, choosing the best one for the specific browser.

Look up the technical definition for the phrase *real-time* and you'll see the real-time web is not truly real-time. The actual meaning of *real-time* involves software with strict time boundaries that must respond to events within a specified time constraint. It is typically used in embedded systems to respond to button presses, for applications as diverse as junk food dispensers and medical devices in intensive care units. Eat too much junk food and you could end up in intensive care, and you'll be served by real-time software in both cases. Try and remember the distinction between different meanings for this phrase.

The proponents of the so-called real-time web should be calling it the pseudo-real-time-web, but that's not as catchy a phrase.

What does it mean that Socket.IO uses the best protocol for the specific browser? If you were to implement your application with WebSockets, it would be limited to the modern browsers supporting that protocol. Because Socket.IO falls back on so many alternative protocols (WebSockets, Flash, XHR, and JSONP), it supports a wider range of web browsers.

As the application author, you don't have to worry about the specific protocol Socket.IO uses with a given browser. Instead, you can implement the business logic and the library takes care of the details for you.

The Socket.IO package includes both a server-side package and a client library. After an easy configuration, the two will communicate back and forth over a socket. The API between the server side and client side is very similar. Because a Socket.IO application runs code in both browser and server, in this chapter we will be writing code for both.

The model that Socket.IO provides is similar to the `EventEmitter` object. The programmer uses the `.on` method to listen for events and the `.emit` method to send them. But with Socket.IO, an event is sent not just using its event name, but is targeted to a combination of two spaces maintained by Socket.IO – the *namespace* and the *room*. Further, the events are sent between the browser and the server rather than being limited to the Node.js process.

Information about Socket.IO is available at <https://socket.io/>.



On the server side, we wrap the HTTP Server object using the Socket.IO library, giving us the Socket.IO Server object. The Server object lets us create two kinds of communication spaces, *namespaces*, and *rooms*. With it we can send messages, using the `emit` method, either globally or into one of those spaces. We can also listen for messages, using the `on` method, either globally or from a namespace or room.

On the client side, we load the library from the Socket.IO server. Then, client code running in the browser opens one or more communication channels to the server, and the client can connect to namespaces or rooms.

This high-level overview should help to understand the following work. Our next step is to integrate Socket.IO into the initialization of the Notes application.

Initializing Socket.IO with Express

Socket.IO works by wrapping itself around an HTTP Server object. Think back to Chapter 4, *HTTP Servers and Clients*, where we wrote a module that hooked into HTTP Server methods so that we could spy on HTTP transactions. The HTTP Sniffer attaches a listener to every HTTP event to print out the events. But what if you used that idea to do real work? Socket.IO uses a similar concept, listening to HTTP requests and responding to specific ones by using the Socket.IO protocol to communicate with client code in the browser.

To get started, let's first make a duplicate of the code from the previous chapter. If you created a directory named `chap08` for that code, create a new directory named `chap09` and copy the source tree there.

We won't make changes to the user authentication microservice, but we will use it for user authentication, of course.

In the Notes source directory, install these new modules:

```
$ npm install socket.io@2.x passport.socketio@3.7.x --save
```

We will incorporate user authentication with the `passport` module, used in Chapter 8, *Authenticating Users with a Microservice*, into some of the real-time interactions we'll implement.

At the beginning of `app.mjs`, add this to the `import` statements:

```
import socketio from 'socket.io';
import passportSocketIO from 'passport.socketio';
```

This code brings in the required modules. The `socket.io` package supplies the core event-passing service. The `passport.socketio` module integrates Socket.IO with PassportJS-based user authentication. We will be reorganizing `app.mjs` so that session management will be shared between Socket.IO, Express, and Passport.

The first change is to move the declaration of some session-related values to the top of the module, as we've done here:

```
import session from 'express-session';
import sessionFileStore from 'session-file-store';
const FileStore = sessionFileStore(session);
export const sessionCookieName = 'notescookie.sid';
const sessionSecret = 'keyboard mouse';
const sessionStore = new FileStore({ path: "sessions" });
```

What this does is create a couple of global scope variables to hold objects related to the session configuration. We had been using these values as constants when setting up the Express session support. We now need to share those values with both the Socket.IO and the Express session managers. When we initialize both Express and Socket.IO session handlers, there is an initialization object taking initialization parameters. In each, we will pass in the same values for the `secret` and `sessionStore` fields, to ensure they are in agreement.

The next change is moving some code related to setting up the server object from the bottom of `app.mjs` closer to the top, as shown here:

```
export const app = express();

export const port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

export const server = http.createServer(app);

server.listen(port);
server.on('request', (req, res) => {
  debug(`\$new Date().toISOString() request \$${req.method}
\$${req.url}`);
});
server.on('error', onError);
server.on('listening', onListening);

export const io = socketio(server);

io.use(passportSocketIo.authorize({
  cookieParser: cookieParser,
  key: sessionCookieName,
  secret: sessionSecret,
  store: sessionStore
}));
```

In addition to moving some code from the bottom of `app.mjs`, we've added the initialization for Socket.IO. This is where the Socket.IO library wraps itself around the HTTP server object. Additionally, we're integrating it with the Passport library so that Socket.IO knows which sessions are authenticated.

The creation of the `app` and `server` objects is the same as before. All that's changed is the location in `app.mjs` where that occurred. What's new is the `io` object, which is our entry point into the Socket.IO API, and it is used for all Socket.IO operations. This precise object must be made available to other modules wishing to use Socket.IO operations since this object was created by wrapping the HTTP server object. Hence, the `io` object is exported so that other modules can import it.

By invoking `socketio(server)`, we have given Socket.IO access to the HTTP server. It listens for incoming requests on the URLs through which Socket.IO does its work. That's invisible to us, and we don't have to think about what's happening under the covers.



According to the Socket.IO internals, it looks like Socket.IO uses the `/socket.io` URL. That means our applications must avoid using this URL. See <https://socket.io/docs/internals/>.

The `io.use` function installs functions in Socket.IO that are similar to Express middleware, which the Socket.IO documentation even calls middleware. In this case, the middleware function is returned by calling `passportSocketIO.authorize`, and is how we integrate Passport authentication into Socket.IO.

Because we are sharing session management between Express and Socket.IO, we must make the following change:

```
app.use(session({
  store: sessionStore,
  secret: sessionSecret,
  resave: true,
  saveUninitialized: true,
  name: sessionCookieName
}));
initPassport(app);
```

This is the same configuration of Express session support that we added in Chapter 8, *Authenticating Users with a Microservice*, but modified to use the configuration variables we set up earlier. Done this way, both Express and Socket.IO session handling is managed from the same set of information.

We have accomplished the basic setup of Socket.IO in our Express application. First, we connected the Socket.IO library to the HTTP Server so that it can handle requests on the Socket.IO service. Then we integrated it with Passport session management.

Let's now learn how we can use Socket.IO to add real-time updating in Notes.

Real-time updates on the Notes homepage

The goal we're working toward is for the Notes home page to automatically update the list of notes as notes are edited or deleted. What we've done so far is to restructure the application startup so that Socket.IO is initialized in the Notes application. There's no change of behavior yet.

What we will do is send an event whenever a note is created, updated, or deleted. Any interested part of the Notes application can listen to those events and act appropriately. For example, the Notes home page router module can listen for events, and then send an update to the browser. The code in the web browser will listen for an event from the server, and in response, it would rewrite the home page. Likewise, when a Note is modified, a listener can send a message to the web browser with the new note content, or if the Note is deleted, a listener can send a message so that the web browser redirects to the home page.

These changes are required:

- Refactoring the Notes Store implementations to send create, update, and delete events
- Refactoring the templates to support both Bootstrap on every page and a custom Socket.IO client for each page
- Refactoring the home page and Notes' viewing router modules to listen for Socket.IO events and send updates to the browser

We'll handle this over the next few sections, so let's get started.

Refactoring the NotesStore classes to emit events

In order to automatically update the user interface when a Note is changed or deleted or created, the `NotesStore` must send events to notify interested parties of those changes. We will employ our old friend, the `EventEmitter` class, to manage the listeners to the events we must send.

Recall that we created a class, `AbstractNotesStore`, and that every storage module contains a subclass of `AbstractNotesStore`. Hence we can add listener support in `AbstractNotesStore`, making it automatically available to the implementations.

In `models/Notes.mjs`, make this change:

```
import EventEmitter from 'events';

export class AbstractNotesStore extends EventEmitter {
  static store() { }
  async close() { }
  async update(key, title, body) { }
  async create(key, title, body) { }
  async read(key) { }
  async destroy(key) { }
  async keylist() { }
  async count() { }

  emitCreated(note) { this.emit('notecreated', note); }
  emitUpdated(note) { this.emit('noteupdated', note); }
  emitDestroyed(key) { this.emit('notedestroyed', key); }
}
```

We imported the `EventEmitter` class, made `AbstractNotesStore` a subclass of `EventEmitter`, and then added some methods to emit events. As a result, every `NotesStore` implementation now has an `on` and `emit` method, plus these three helper methods.

This is only the first step since nothing is emitting any events. We have to rewrite the `create`, `update`, and `destroy` methods in `NotesStore` implementations to call these methods so the events are emitted.



In the interest of space, we'll show the modifications to one of the `NotesStore` implementations, and leave the rest as an exercise for you.

Modify these functions in `models/notes-sequelize.mjs` as shown in the following code:

```
async update(key, title, body) {
  ...
  const note = await this.read(key);
  this.emitUpdated(note);
  return note;
}
async create(key, title, body) {
  ...
  const note = new Note(sqnote.notekey, sqnote.title, sqnote.body);
```

```
    this.emitCreated(note);
    return note;
}

async destroy(key) {
  ...
  this.emitDestroyed(key);
}
```

The changes do not change the original contract of these methods, since they still create, update, and destroy notes. The other `NotesStore` implementations require similar changes. What's new is that now those methods emit the appropriate events for any code that may be interested.

Another task to take care of is initialization, which must happen after `NotesStore` is initialized. Recall that setting up `NotesStore` is asynchronous. Therefore, calling the `.on` function to register an event listener must happen after `NotesStore` is initialized.

In both `routes/index.mjs` and `routes/notes.mjs`, add the following function:

```
export function init() {
```

This function is meant to be in place of such initialization.

Then, in `app.mjs`, make this change:

```
import {
  router as indexRouter, init as homeInit
} from './routes/index.mjs';
import {
  router as notesRouter, init as notesInit
} from './routes/notes.mjs';

...
import { useModel as useNotesModel } from './models/notes-store.mjs';
useNotesModel(process.env.NOTES_MODEL)
.then(store => {
  homeInit();
  notesInit();
})
.catch(error => { onError({ code: 'ENOTESSTORE', error }); });
```

This imports the two `init` functions, giving them unique names, then calling them once `NotesStore` is set up. At the moment, both functions do nothing, but that will change shortly. The important thing is these two `init` functions will be called after `NotesStore` is completely initialized.

We have our `NotesStore` sending events when a Note is created, updated, or destroyed. Let's now use those events to update the user interface appropriately.

Real-time changes in the Notes home page

The Notes model now sends events as Notes are created, updated, or destroyed. For this to be useful, the events must be displayed to our users. Making the events visible to our users means the controller and view portions of the application must consume those events.

At the top of `routes/index.mjs`, add this to the list of imports:

```
import { io } from '../app.mjs';
```

Remember that this is the initialized Socket.IO object we use to send messages to and from connected browsers. We will use it to send messages to the Notes home page.

Then refactor the `router` function:

```
router.get('/', async (req, res, next) => {
  try {
    const notelist = await getKeyTitlesList();
    res.render('index', {
      title: 'Notes', notelist: notelist,
      user: req.user ? req.user : undefined
    });
  } catch (e) { next(e); }
});

async function getKeyTitlesList() {
  const keylist = await notes.keylist();
  const keyPromises = keylist.map(key => notes.read(key));
  const notelist = await Promise.all(keyPromises);
  return notelist.map(note => {
    return { key: note.key, title: note.title };
  });
};
```

This extracts what had been the body of the `router` function into a separate function. We need to use this function not only in the home page `router` function but also when we emit Socket.IO messages for the home page.

We did change the return value. Originally, it contained an array of Note objects, and now it contains an array of anonymous objects containing `key` and `title` data. We did this because providing the array of Note objects to Socket.IO resulted in an array of empty objects being sent to the browser while sending the anonymous objects worked correctly.

Then, add this at the bottom:

```
const emitNoteTitles = async () => {
  const notelist = await getKeyTitlesList();
  io.of('/home').emit('notetitles', { notelist });
};

export function init() {
  io.of('/home').on('connect', socket => {
    debug('socketio connection on /home');
  });
  notes.on('notecreated', emitNoteTitles);
  notes.on('noteupdate', emitNoteTitles);
  notes.on('notedestroy', emitNoteTitles);
}
```

The primary purpose of this section is to listen to the create/update/destroy events, so we can update the browser. For each, the current list of Notes is gathered, then sent to the browser.

As we said, the Socket.IO package uses a model similar to the `EventEmitter` class. The `emit` method sends an event, and the policy of event names and event data is the same as with `EventEmitter`.

Calling `io.of('/namespace')` creates a `Namespace` object for the named namespace. Namespaces are named in a pattern that looks like a pathname in Unix-like filesystems.

Calling `io.of('/namespace').on('connect'...)` has the effect of letting server-side code know when a browser connects to the named namespace. In this case, we are using the `/home` namespace for the Notes home page. This has the side-effect of keeping the namespace active after it is created. Remember that `init` is called during the initialization of the server. Therefore, we will have created the `/home` namespace long before any web browser tries to access that namespace by visiting the Notes application home page.

Calling `io.emit(...)` sends a broadcast message. Broadcast messages are sent to every browser connected to the application server. That can be useful in some situations, but in most situations, we want to avoid sending too many messages. To limit network data consumption, it's best to target each event to the browsers that need the event.

Calling `io.of('/namespace').emit(...)` targets the event to browsers connected to the named namespace. When the client-side code connects to the server, it connects with one or more namespaces. Hence, in this case, we target the `notetitles` event to browsers attached to the `/home` namespace, which we'll see later is the Notes home page.

Calling `io.of('/namespace').to('room')` accesses what Socket.IO calls a `room`. Before a browser receives events in a room, it must *join* the room. Rooms and namespaces are similar, but different, things. We'll use rooms later.

The next task accomplished in the `init` function is to create the event listeners for the `notecreated`, `noteupdate`, and `notedestroy` events. The handler function for each emits a Socket.IO event, `notetitles`, containing the list of note keys and titles.

As Notes are created, updated, and destroyed, we are now sending an event to the home page that is intended to refresh the page to match the change. The home page template, `views/index.hbs`, must be refactored to receive that event and rewrite the page to match.

Changing the home page and layout templates

Socket.IO runs on both the client and the server, with the two communicating back and forth over the HTTP connection. So far, we've seen the server side of using Socket.IO to send events. The next step is to install a Socket.IO client on the Notes home page.

Generally speaking, every application page is likely to need a different Socket.IO client, since each page has different requirements. This means we must change how JavaScript code is loaded in Notes pages.

Initially, we simply put JavaScript code required by Bootstrap and FeatherJS at the bottom of `layout.hbs`. That worked because every page required the same set of JavaScript modules, but now we've identified the need for different JavaScript code on each page. Because the custom Socket.IO clients for each page use jQuery for DOM manipulation, they must be loaded after jQuery is loaded. Therefore, we need to change `layout.hbs` to not load the JavaScript. Instead, every template will now be required to load the JavaScript code it needs. We'll supply a shared code snippet for loading the Bootstrap, Popper, jQuery, and FeatherJS libraries but beyond that, each template is responsible for loading any additional required JavaScript.

Create a file, `partials/footerjs.hbs`, containing the following code:

```
<!-- jquery first, then Popper.js, then Bootstrap JS -->
<script src="/assets/vendor/jquery/jquery.min.js"></script>
<script src="/assets/vendor/popper.js/popper.min.js"></script>
<script src="/assets/vendor/bootstrap/js/bootstrap.min.js"></script>
<script src="/assets/vendor/feather-icons/feather.js"></script>
<script>
  feather.replace();
</script>
```

This code had been at the bottom of `views/layout.hbs`, and it is the shared code snippet we just mentioned. This is meant to be used on every page template, and to be followed by custom JavaScript.

We now need to modify `views/layout.hbs` as follows:

```
<html>
<head>...</head>
<body>
  {{> header }}
  {{> body}}
</body>
</html>
```

That is, we'll leave `layout.hbs` pretty much as it was, except for removing the JavaScript tags from the bottom. Those tags are now in `footerjs.hbs`.

We'll now need to modify every template (`error.hbs`, `index.hbs`, `login.hbs`, `notedestroy.hbs`, `noteedit.hbs`, and `noteview.hbs`) to, at the minimum, load the `footerjs` partial.

```
{> footerjs}
```

With this, every one of the templates explicitly loads the JavaScript code for Bootstrap and FeatherJS at the bottom of the page. They were previously loaded at the bottom of the page in `layout.hbs`. What this bought us is the freedom to load Socket.IO client code after Bootstrap and jQuery are loaded.

We have changed every template to use a new policy for loading the JavaScript. Let's now take care of the Socket.IO client on the home page.

Adding a Socket.IO client to the Notes home page

Remember that our task is to add a Socket.IO client to the home page so that the home page receives notifications about created, updated, or deleted Notes.

In `views/index.hbs`, add this at the bottom, after the `footerjs` partial:

```
{> footerjs}

<script src="/socket.io/socket.io.js"></script>
<script>
$(document).ready(function () {
    var socket = io('/home');
    socket.on('connect', socket => {
        console.log('socketio connection on /home');
    });
    socket.on('notetitles', function(data) {
        var notelist = data.notelist;
        $('#notetitles').empty();
        for (var i = 0; i < notelist.length; i++) {
            notedata = notelist[i];
            $('#notetitles')
                .append('<a class="btn btn-lg btn-block btn-outline-dark"'
                    + href="/notes/view?key=' + notedata.key + '">' +
                    notedata.title + '</a>');
        }
    });
});
</script>
```

This is what we meant when we said that each page will have its own Socket.IO client implementation. This is the client for the home page, but the client for the Notes view page will be different. This Socket.IO client connects to the /home namespace, then for `notetitles` events, it redraws the list of Notes on the home page.

The first `<script>` tag is where we load the Socket.IO client library, from `/socket.io/socket.io.js`. You'll notice that we never set up any Express route to handle the `/socket.io` URL. Instead, the Socket.IO library did that for us. Remember that the Socket.IO library handles every request starting with `/socket.io`, and this is one of such request it handles. The second `<script>` tag is where the page-specific client code lives.

Having client code within a `$(document).ready(function() { .. })` block is typical when using jQuery. This, as the code implies, waits until the web page is fully loaded, and then calls the supplied function. That way, our client code is not only held within a private namespace; it executes only when the page is fully set up.

On the client side, calling `io()` or `io('/namespace')` creates a `socket` object. This object is what's used to send messages to the server or to receive messages from the server.

In this case, the client connects a `socket` object to the `/home` namespace, which is the only namespace defined so far. We then listen for the `notetitles` events, which is what's being sent from the server. Upon receiving that event, some jQuery DOM manipulation erases the current list of Notes and renders a new list on the screen. The same markup is used in both places.

Additionally, for this script to function, this change is required elsewhere in the template:

```
<div class="col-12 btn-group-vertical" id="notetitles" role="group">
  ...
</div>
```

You'll notice in the script that it references `$("#notetitles")` to clear the existing list of note titles, then to add a new list. Obviously, that requires an `id="notetitles"` attribute on this `<div>`.

Our code in `routes/index.mjs` listened to various events from the Notes model and, in response, sent a `notetitles` event to the browser. The browser code takes that list of note information and redraws the screen.



You might notice that our browser-side JavaScript is not using ES-2015/2016/2017 features. This code would, of course, be cleaner if we were to do so. How can we know whether our visitors use a browser modern enough for those language features? We could use Babel to transpile ES-2015/2016/2017 code into ES5 code capable of running on any browser. However, it is a pragmatic trade-off to still write ES5 code in the browser.

Running Notes with real-time home page updates

We now have enough implemented to run the application and see some real-time action.

As you did earlier, start the user information microservice in one window:

```
$ npm start

> user-auth-server@0.0.1 start /Users/david/chap09/users
> DEBUG=users:* PORT=5858 SEQUELIZE_CONNECT=sequelize-sqlite.yaml node
   ./user-server.mjs

  users:service User-Auth-Service listening at http://127.0.0.1:5858
  +0ms
```

Then, in another window, start the Notes application:

```
$ npm start

> notes@0.0.0 start /Users/david/chap09/notes
> DEBUG=notes:* SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml
  NOTES_MODEL=sequelize USER_SERVICE_URL=http://localhost:5858 node --
  experimental-modules ./app

(node:11998) ExperimentalWarning: The ESM module loader is
experimental.
  notes:debug-INDEX Listening on port 3000 +0ms
```

Then, in a browser window, go to `http://localhost:3000` and log in to the Notes application. To see the real-time effects, open multiple browser windows. If you can use Notes from multiple computers, then do that as well.

In one browser window, start creating and deleting notes, while leaving the other browser windows viewing the home page. Create a note, and it should show up immediately on the home page in the other browser windows. Delete a note and it should disappear immediately as well.

One scenario you might try requires three browser windows. In one window, create a new note, and then leave that browser window showing the newly created note. In another window, show the Notes home page. And in the third window, show the newly created note. Now, delete this newly created note. Of those windows, two are correctly updated and are now showing the home page. The third, where we were simply viewing the note, is still showing that note even though it no longer exists.

We'll get to that shortly, but first, we need to talk about how to debug your Socket.IO client code.

A word on enabling debug tracing in Socket.IO code

It is useful to inspect what Socket.IO is doing in case you're having trouble. Fortunately, the Socket.IO package uses the same Debug package that Express uses, and we can turn on debug tracing just by setting the `DEBUG` environment variable. It even uses a variable, `localStorage.debug`, with the same syntax on the client side, and we can enable debug tracing in the browser as well.

On the server side, this is a useful `DEBUG` environment variable setting:

```
DEBUG=notes:*,socket.io:*
```

This enables debug tracing for the Notes application and the Socket.IO package.

Enabling this in a browser is a little different since there are no environment variables. Simply open up the JavaScript console in your browser and enter this command:

```
localStorage.debug = 'socket.io-client:*,socket.io-parser';
```

Immediately, you will start seeing a constant chatter of messages from Socket.IO. One thing you'll learn is that even when the application is idle, Socket.IO is communicating back and forth.

There are several other `DEBUG` strings to use. For example, Socket.IO relies on the Engine.IO package for its transport layer. If you want debug tracing of that package, add `engine:*` to the `DEBUG` string. The strings shown were most helpful during the testing of this chapter.

Now that we've learned about debug tracing, we can take care of changing the `/notes/view` pages to react so they changes to the Note being viewed.

Real-time action while viewing notes

It's cool how we can now see real-time changes in a part of the Notes application. Let's turn to the /notes/view page to see what we can do. What comes to mind is this functionality:

- Update the note if someone else edits it.
- Redirect the viewer to the home page if someone else deletes the note.
- Allow users to leave comments on the note.

For the first two features, we can rely on the existing events coming from the Notes model. Therefore, we can implement those two features in this section. The third feature will require a messaging subsystem, so we'll get to that later in this chapter.

To implement this, we could create one Socket.IO namespace for each Note, such as /notes/\${notekey}. Then, when the browser is viewing a Note, the client code added to the noteview.hbs template would connect to that namespace. However, that raises the question of how to create those namespaces. Instead, the implementation selected was to have one namespace, /notes, and to create one room per Note.

In routes/notes.mjs, make sure to import the `io` object as shown here:

```
import { emitNoteTitles } from './index.mjs';
import { io } from '../app.mjs';
```

This, of course, makes the `io` object available to code in this module. We're also importing a function from `index.mjs` that is not currently exported. We will need to cause the home page to be updated, and therefore in `index.mjs`, make this change:

```
export const emitNoteTitles = async () => { ... };
```

This simply adds the `export` keyword so we can access the function from elsewhere.

Then, change the `init` function to this:

```
export function init() {
  io.of('/notes').on('connect', socket => {
    if (socket.handshake.query.key) {
      socket.join(socket.handshake.query.key);
    }
  });
  notes.on('noteupdated', note => {
    const toemit = {
      key: note.key, title: note.title, body: note.body
    }
    io.of('/notes').emit('noteupdated', toemit);
  });
}
```

```
    };
    io.of('/notes').to(note.key).emit('noteupdated', toemit);
    emitNoteTitles();
});
notes.on('notedestroyed', key => {
    io.of('/notes').to(key).emit('notedestroyed', key);
    emitNoteTitles();
});
}
```

First, we handle `connect` events on the `/notes` namespace. In the handler, we're looking for a `query` object containing the `key` for a Note. Therefore, in the client code, when calling `io('/notes')` to connect with the server, we'll have to arrange to send that `key` value. It's easy to do, and we'll learn how in a little while.

Calling `socket.join(roomName)` does what is suggested—it causes this connection to join the named room. Therefore, this connection will be addressed as being in the `/notes` namespace, and in a room whose name is the `key` for a given Note.

The next thing is to add listeners for the `noteupdated` and `notedestroyed` messages. In both, we are using this pattern:

```
io.of('/namespace').to(roomName).emit(..);
```

This is how we use Socket.IO to send a message to any browser connected to the given namespace and room.

For `noteupdated`, we simply send the new Note data. We again had to convert the Note object into an anonymous JavaScript object, because otherwise, an empty object arrived in the browser. The client code will have to use, as we will see shortly, jQuery operations to update the page.

For `notedestroyed`, we simply send the `key`. Since the client code will respond by redirecting the browser to the home page, we don't have to send anything at all.

In both, we also call `emitNoteTitles` to ensure the home page is updated if it is being viewed.

Changing the note view template for real-time action

As we did in the home page template, the data contained in these events must be made visible to the user. We must not only add client code to the template, `views/noteview.hbs`; we need a couple of small changes to the template:

```
<div class="container-fluid">
    <div class="row"><div class="col-xs-12">
        {{#if note}}<h3 id="notetitle">{{ note.title }}</h3>{{/if}}
        {{#if note}}<div id="notebody">{{ note.body }}</div>{{/if}}
        <p>Key: {{ notekey }}</p>
    </div></div>
    {{#if user }}{{#if notekey }}
        <div class="row"><div class="col-xs-12">
            <div class="btn-group">
                <a class="btn btn-outline-dark"
                    href="/notes/destroy?key={{notekey}}"
                    role="button">Delete</a>
                <a class="btn btn-outline-dark"
                    href="/notes/edit?key={{notekey}}"
                    role="button">Edit</a>
            </div></div></div>
        {{/if}}/{{/if}}
    </div>
```

In this section of the template, we add a pair of IDs to two elements. This enables the JavaScript code to target the correct elements.

Add this client code to `noteview.hbs`:

```
{{> footerjs}}
```

```
 {{#if notekey }}
<script src="/socket.io/socket.io.js"></script>
<script>
$(document).ready(function () {
    let socket = io('/notes', {
        query: { key: '{{ notekey }}' }
    });
    socket.on('noteupdated', note => {
        $('h3#notetitle').empty();
        $('h3#notetitle').text(note.title);
        $('#navbartitle').empty();
        $('#navbartitle').text(note.title);
        $('#notebody').empty();
        $('#notebody').text(note.body);
    });
    socket.on('notedestroyed', key => {
```

```
        window.location.href = "/";
    });
});
</script>
{{/if}}
```

In this script, we first connect to the `/notes` namespace and then create listeners for the `noteupdated` and `notedestroyed` events.

When connecting to the `/notes` namespace, we are passing an extra parameter. The optional second parameter to this function is an options object, and in this case, we are passing the `query` option. The `query` object is identical in form to the `query` object of the `URL` class. This means the namespace is as if it were a URL such as `/notes?key=${notekey}`. Indeed, according to the Socket.IO documentation, we can pass a full URL, and it also works if the connection is created like this:

```
let socket = io('/notes?key={{ notekey }}');
```

While we could set up the URL query string this way, it's cleaner to do it the other way.

We need to call out a technique being used. These code snippets are written in a Handlebars template, and therefore the syntax `{{ expression }}` is executed on the server, with the result of that expression to be substituted into the template. Therefore, the `{{ expression }}` construct accesses server-side data. Specifically, `query: { key: '{{ notekey }}' }` is a data structure on the client side, but the `'{{ notekey }}'` portion is evaluated on the server. The client side does not see `{{ notekey }}` , it sees the value `notekey` had on the server.

For the `noteupdated` event, we take the new note content and display it on the screen. For this to work, we had to add `id=` attributes to certain HTML elements so we could use jQuery selectors to manipulate the correct elements.

Additionally in `partials/header.hbs`, we needed to make this change as well:

```
<span id="navbartitle" class="navbar-text text-dark col">{{ title }}</span id="navbartitle">
```

We needed to update the title at the top of the page as well, and this `id` attribute helps to target the correct element.

For the `notedestroyed` event, we simply redirect the browser window back to the home page. The note being viewed has been deleted, and there's no point the user continuing to look at a note that no longer exists.

Running Notes with pseudo-real-time updates while viewing a note

At this point, you can now rerun the Notes application and try the new real-time updates feature.

By now you have put Notes through its paces many times, and know what to do. Start by launching the user authentication server and the Notes application. Make sure there is at least one note in the database; add one if needed. Then, open multiple browser windows with one viewing the home page and two viewing the same note. In a window viewing the note, edit the note to make a change, making sure to change the title. The text change should change on both the home page and the page viewing the note.

Then delete the note and watch it disappear from the home page, and further, the browser window that had viewed the note is now on the home page.

We took care of a lot of things in this section, and the Notes application now has dynamic updates happening. To do this, we created an event-based notification system, then used Socket.IO in both browser and server to communicate data back and forth.

We have implemented most of what we've set out to do. By refactoring the Notes Store implementations to send events, we are able to send events to Socket.IO clients in the browser. That in turn is used to automatically update the Notes home page, and the /notes/view page.

The remaining feature is for users to be able to write comments on Notes. In the next section, we will take care of that by adding a whole new database table to handle messages.

Inter-user chat and commenting for Notes

This is cool! We now have real-time updates in Notes as we edit delete or create notes. Let's now take it to the next level and implement something akin to inter-user chatting.

Earlier, we named three things we could do with Socket.IO on /notes/view pages. We've already implemented live updating when a Note is changed and a redirect to the home page if a Note is deleted; the remaining task is to allow users to make comments on Notes.

It's possible to pivot our Notes application concept and take it in the direction of a social network. In the majority of such networks, users post things (notes, pictures, videos, and so on), and other users comment on those things. Done well, these basic elements can develop a large community of people sharing notes with each other. While the Notes application is kind of a toy, it's not too terribly far from being a basic social network. Commenting the way we will do now is a tiny step in that direction.

On each note page, we'll have an area to display messages from Notes users. Each message will show the username, a timestamp, and their message. We'll also need a method for users to post a message, and we'll also allow users to delete messages.

Each of those operations will be performed without refreshing the screen. Instead, code running inside the web page will send commands to/from the server and take action dynamically. By doing this, we'll learn about Bootstrap modal dialogs, as well as more about sending and receiving Socket.IO messages. Let's get started.

Data model for storing messages

We need to start by implementing a data model for storing messages. The basic fields required are a unique ID, the username of the person sending the message, the namespace and the room associated with the message, the message, and finally a timestamp for when the message was sent. As messages are received or deleted, events must be emitted from the data model so we can do the right thing on the web page. We associate messages with a room and namespace combination because in Socket.IO that combination has proved to be a good way to address a specific page in the Notes application.

This data model implementation will be written for Sequelize. If you prefer a different storage solution, you can, by all means, re-implement the same API on other data storage systems.

Create a new file, `models/messages-sequelize.mjs`, containing the following:

```
import Sequelize from 'sequelize';
import {
  connectDB as connectSequlz,
  close as closeSequlz
} from './sequlz.mjs';

import EventEmitter from 'events';
class MessagesEmitter extends EventEmitter {}
export const emitter = new MessagesEmitter();
```

```
import DBG from 'debug';
const debug = DBG('notes:model-messages');
const error = DBG('notes:error-messages');
```

This sets up the modules being used and also initializes the `EventEmitter` interface. We're also exporting the `EventEmitter` as `emitter` so other modules can be notified about messages as they're created or deleted.

Now add this code for handling the database connection:

```
let sequelize;
export class SQMessage extends Sequelize.Model {}

async function connectDB() {
    if (sequelize) return;
    sequelize = await connectSequelize();

    SQMessage.init({
        id: { type: Sequelize.INTEGER, autoIncrement: true,
              primaryKey: true },
        from: Sequelize.STRING,
        namespace: Sequelize.STRING,
        room: Sequelize.STRING,
        message: Sequelize.STRING(1024),
        timestamp: Sequelize.DATE
    }, {
        hooks: {
            afterCreate: (message, options) => {
                const toEmit = sanitizedMessage(message);
                emitter.emit('newmessage', toEmit);
            },
            afterDestroy: (message, options) => {
                emitter.emit('destroymessage', {
                    id: message.id,
                    namespace: message.namespace,
                    room: message.room
                });
            }
        },
        sequelize,
        modelName: 'SQMessage'
    });
    await SQMessage.sync();
}
```

The structure of `connectDB` is similar to what we did in `notes-sequelize.mjs`. We use the same `connectSequlz` function to connect with the same database, and we return immediately if the database is already connected.

With `SQMessage.init`, we define our message schema in the database. We have a simple database schema that is fairly self-explanatory. To emit events about messages, we're using a Sequelize feature to be called at certain times.

The `id` field won't be supplied by the caller; instead, it will be autogenerated. Because it is an `autoIncrement` field, each message that's added will be assigned a new `id` number by the database. The equivalent in MySQL is the `AUTO_INCREMENT` attribute on a column definition.

The `namespace` and `room` fields together define which page in Notes each message belongs to. Remember that when emitting an event with Socket.IO we can target the event to one or both of those spaces, and therefore we will use these values to target each message to a specific page.

So far we defined one namespace, `/home`, for the Notes home page, and another namespace, `/notes`, for viewing an individual note. In theory, the Notes application could be expanded to have messages displayable in other areas. For example, a `/private-message` namespace could be used for private messages. Therefore, the schema is defined with both a `namespace` and `room` field so that, in due course, we could use messages in any future part of the Notes application that may be developed.

For our current purposes, messages will be stored with `namespace` equal to `/home`, and `room` equal to the `key` of a given Note.

We will use the `timestamp` to present messages in the order of when they were sent. The `from` field is the username of the sender.

To send notifications about created and destroyed messages, let's try something different. If we follow the pattern we used earlier, the functions we're about to create will have `emitter.emit` calls with corresponding messages. But Sequelize offers a different approach.

With Sequelize, we can create what are called hook methods. Hooks can also be called **life cycle events**, and they are a series of functions we can declare. Hook methods are invoked when certain trigger states exist for the objects managed by Sequelize. In this case, our code needs to know when a message is created, and when a message is deleted.

Hooks are declared as shown in the options object. A field named `hooks` in the schema options object defines hook functions. For each hook we want to use, add an appropriately named field containing the hook function. For our needs, we need to declare `hooks.afterCreate` and `hooks.afterDestroy`. For each, we've declared a function that takes the instance of the `SQMessage` object that has just been created or destroyed. And, with that object, we call `emitter.emit` with either the `newmessage` or `destroymessage` event name.

Continue by adding this function:

```
function sanitizedMessage(msg) {
  return {
    id: msg.id,
    from: msg.from,
    namespace: msg.namespace,
    room: msg.room,
    message: msg.message,
    timestamp: msg.timestamp
  };
}
```

The `sanitizedMessage` function performs the same function as `sanitizedUser`. In both cases, we are receiving a Sequelize object from the database, and we want to return a simple object to the caller. These functions produce that simplified object.

Next, we have several functions to store new messages, retrieve messages, and delete messages.

The first is this function:

```
export async function postMessage(from, namespace, room, message) {
  await connectDB();
  const newmsg = await SQMessage.create({
    from, namespace, room, message, timestamp: new Date()
  });
}
```

This is to be called when a user posts a new comment/message. We store it in the database, and the hook emits an event saying the message was created.

Remember that the `id` field is auto-created as the new message is stored. Therefore, it is not supplied when calling `SQMessage.create`.

This function, and the next, could have contained the `emitter.emit` call to send the `newmessage` or `destroymessage` events. Instead, those events are sent in the hook functions we created earlier. The question is whether it is correct to place `emitter.emit` in a hook function, or to place it here.

The rationale used here is that by using hooks we are assured of always emitting the messages.

Then, add this function:

```
export async function destroyMessage(id) {
    await connectDB();
    const msg = await SQMessage.findOne({ where: { id } });
    if (msg) {
        msg.destroy();
    }
}
```

This is to be called when a user requests that a message should be deleted. With Sequelize, we must first find the message and then delete it by calling its `destroy` method.

Add this function:

```
export async function recentMessages(namespace, room) {
    await connectDB();
    const messages = await SQMessage.findAll({
        where: { namespace, room },
        order: [ ['timestamp', 'DESC'] ],
        limit: 20
    });
    const msgs = messages.map(message => {
        return sanitizedMessage(message);
    });
    return (msgs && msgs.length >= 1) ? msgs : undefined;
}
```

This function retrieves recent messages, and the immediate use case is for this to be used while rendering `/notes/view` pages.

While our current implementation is for viewing a Note, it is generalized to work for any Socket.IO namespace and room. This is for possible future expansion, as we explained earlier. It finds the most recent 20 messages associated with the given namespace and room combination, then returns a cleaned-up list to the caller.

In `findAll`, we specify an `order` attribute. This is similar to the `ORDER BY` phrase in SQL. The `order` attribute takes an array of one or more descriptors declaring how Sequelize should sort the results. In this case, there is one descriptor, saying to sort by the `timestamp` field in descending order. This will cause the most recent message to be displayed first.

We have created a simple module to store messages. We didn't implement the full set of **create, read, update, and delete** (CRUD) operations because they weren't necessary for this task. The user interfaces we're about to create only let folks add new messages, delete existing messages, and view the current messages.

Let's get on with creating the user interface.

Adding support for messages to the Notes router

Now that we can store messages in the database, let's integrate this into the Notes router module.

Integrating messages to the `/notes/view` page will require some new HTML and JavaScript in the `notesview.hbs` template, and some new Socket.IO communications endpoints in the `init` function in `routes/notes.mjs`. In this section, let's take care of those communications endpoints, then in the next section let's talk about how to set it up in the user interface.

In `routes/notes.mjs`, add this to the import statements:

```
import {
  postMessage, destroyMessage, recentMessages,
  emitter as msgEvents
} from '../models/messages-sequelize.mjs';

import DBG from 'debug';
const debug = DBG('notes:home');
const error = DBG('notes:error-home');
```

This imports the functions we just created so we can use them. And we also set up `debug` and `error` functions for tracing.

Add these event handlers to the `init` function in `routes/notes.mjs`:

```
msgEvents.on('newmessage', newmsg => {
  io.of(newmsg.namespace).to(newmsg.room).emit('newmessage',
```

```
    newmsg);
  });
msgEvents.on('destroymessage', data => {
  io.of(data.namespace).to(data.room).emit('destroymessage', data);
});
```

These receive notifications of new messages, or destroyed messages, from `models/messages-sequelize.mjs`, then forwards the notification to the browser. Remember that the message object contains the namespace and room, therefore this lets us address this notification to any Socket.IO communication channel.

Why didn't we just make the Socket.IO call in `models/messages-sequelize.mjs`? Clearly, it would have been slightly more efficient, require fewer lines of code, and therefore fewer opportunities for a bug to creep in, to have put the Socket.IO call in `messages-sequelize.mjs`. But we are maintaining the separation between model, view, and controller, which we talked of earlier in [Chapter 5, Your First Express Application](#). Further, can we predict confidently that there will be no other use for messages in the future? This architecture allows us to connect multiple listener methods to those message events, for multiple purposes.

In the user interface, we'll have to implement corresponding listeners to receive these messages, then take appropriate user interface actions.

In the `connect` listener in the `init` function, add these two new event listeners:

```
io.of('/notes').on('connect', async (socket) => {
  let notekey = socket.handshake.query.key;
  if (notekey) {
    socket.join(notekey);

    socket.on('create-message', async (newmsg, fn) => {
      try {
        await postMessage(
          newmsg.from, newmsg.namespace, newmsg.room,
          newmsg.message);
        fn('ok');
      } catch (err) {
        error(`FAIL to create message ${err.stack}`);
      }
    });

    socket.on('delete-message', async (data) => {
      try {
        await destroyMessage(data.id);
      } catch (err) {
        error(`FAIL to delete message ${err.stack}`);
      }
    });
  }
});
```

```
        }
    });
});
```

This is the existing function to listen for connections from /notes/view pages, but with two new Socket.IO event handler functions. Remember that in the existing client code in notesview.hbs, it connects to the /notes namespace and supplies the note key as the room to join. In this section, we build on that by also setting up listeners for create-message and delete-message events when a note key has been supplied.

As the event names imply, the create-message event is sent by the client side when there is a new message, and the delete-message event is sent to delete a given message. The corresponding data model functions are called to perform those functions.

For the create-message event, there is an additional feature being used. This uses what Socket.IO calls an acknowledgment function.

So far, we've used the Socket.IO `emit` method with an event name and a data object. We can also include a `callback` function as an optional third parameter. The receiver of the message will receive the function and can call the function, and any data passed to the function is sent to the `callback` function. The interesting thing is this works across the browser-server boundary.

This means our client code will do this:

```
io.of('/notes').to(note.key).emit('create-message', {
  ... message data
},
function (result) {
  ... acknowledgement action
});
```

That function in the third parameter becomes the `fn` parameter in the `create-message` event handler function. Then, anything supplied to a call to `fn` will arrive in this function as the `result` parameter. It doesn't matter that it's a browser supplying that function across a connection to the server and that the call to the function happens on the server, Socket.IO takes care of transporting the response data back to the browser code and invoking the acknowledgment function there. The last thing to note is that we're being lazy with error reporting. So, put a task on the backlog to improve error reporting to the users.

The next task is to implement code in the browser to make all this visible to the user.

Changing the note view template for messages

We need to dive back into `views/noterview.hbs` with more changes so that we can view, create, and delete messages. This time, we will add a lot of code, including using a Bootstrap modal popup to get the message, the Socket.IO messages we just discussed, and the jQuery manipulations to make everything appear on the screen.

We want the `/notes/view` page to not cause unneeded page reloads. Instead, we want the user to add a comment by having a pop-up window collect the message text, and then the new message is added to the page, without causing the page to reload. Likewise, if another user adds a message to a Note, we want the message to show up without the page reloading. Likewise, we want to delete messages without causing the page to reload, and for messages to be deleted for others viewing the note without the page reloading.

Of course, this will involve several Socket.IO messages going back and forth between browser and server, along with some jQuery DOM manipulations. We can do both without reloading the page, which generally improves the user experience.

Let's start by implementing the user interface to create a new message.

Composing messages on the Note view page

The next task for the `/notes/view` page is to let the user add a message. They'll click a button, a pop-up window lets them enter the text, they'll click a button in the popup, the popup will be dismissed, and the message will show up. Further, the message will be shown to other viewers of the Note.

The Bootstrap framework includes support for Modal windows. They serve a similar purpose to Modal dialogs in desktop applications. Modal windows appear above existing windows of an application, while preventing interaction with other parts of the web page or application. They are used for purposes such as asking a question of the user. The typical interaction is to click a button, then the application pops up a Modal window containing some UI elements, the user interacts with the Modal, then dismisses it. You will certainly have interacted with many thousands of Modal windows while using computers.

Let's first add a button with which the user will request to add a comment. In the current design, there is a row of two buttons below the Note text. In `views/noteview.hbs`, let's add a third button:

```
<div class="row"><div class="col-xs-12">
<div class="btn-group">
  <a class="btn btn-outline-dark"
    href="/notes/destroy?key={{notekey}}"
    role="button">Delete</a>
  <a class="btn btn-outline-dark"
    href="/notes/edit?key={{notekey}}"
    role="button">Edit</a>
  <button type="button" class="btn btn-outline-dark"
    data-toggle="modal"
    data-target="#notes-comment-modal">Comment</button>
</div>
</div></div>
```

This is directly out of the documentation for the Bootstrap Modal component. The `btn-outline-dark` style matches the other buttons in this row, and between the `data-toggle` and the `data-target` attributes, Bootstrap knows which Modal window to pop up.

Let's insert the definition for the matching Modal window in `views/noteview.hbs`:

```
{{#if notekey}}{{#if user}}
<div class="modal fade" id="notes-comment-modal" tabindex="-1"
  role="dialog" aria-labelledby="noteCommentModalLabel" aria-
  hidden="true">
<div class="modal-dialog modal-dialog-centered" role="document">
  <div class="modal-content"><div class="modal-header">
    <h5 class="modal-title" id="noteCommentModalLabel">Leave
      a Comment</h5>
    <button type="button" class="close" data-dismiss="modal"
      aria-label="Close"><span aria-hidden="true">&times;</span>
    </button>
  </div>
  <div class="modal-body">
    <form id="submit-comment">
      <input id="comment-from" type="hidden"
        name="from" value="{{ user.id }}"/>
      <input id="comment-namespace" type="hidden"
        name="namespace" value="/notes"/>
      <input id="comment-room" type="hidden"
        name="room" value="{{notekey}}"/>
      <input id="comment-key" type="hidden"
        name="key" value="{{notekey}}"/>
    </form>
  </div>
</div>
```

```
<fieldset>
  <div class="form-group">
    <label for="noteCommentTextArea">Your Excellent
      Thoughts</label>
    <textarea id="noteCommentTextArea" name="message"
              class="form-control" rows="3"></textarea>
  </div>
  <div class="form-group">
    <button id="submitNewComment" type="submit"
            class="btn btn-primary col-sm-offset-2 col-sm-10">
      Make Comment</button>
  </div>
</fieldset>
</form>
</div></div>
</div>
{{/if}}{{/if}}
```

Again, this comes directly from the Bootstrap documentation for the Modal component, along with a simple form to collect the message.

Notice there is `<div class="modal-dialog">`, and within that, `<div class="model-content">`. Together, these form what is shown within the dialog window. The content is split between a `<div class="modal-header">` for the top row of the dialog, and a `<div class="modal-body">` for the main content.

The `id` value on the outermost element, `id="notes-comment-modal"`, matches the target declared in the button, `data-target="#notes-comment-modal"`. Another connection to make is `aria-labelledby`, which matches the `id` of the `<h5 class="modal-title">` element.

`<form id="submit-comment">` is minimal because we will not use it to submit anything over an HTTP connection to a regular URL. Therefore, it does not have `action` and `method` attributes. Otherwise, this is a normal everyday Bootstrap `form`, with a `fieldset` and various form elements.

The next step is to add the client-side JavaScript code to make this functional. When clicking the button, we want some client code to run, which will send a `create-message` event matching the code we added to `routes/notes.mjs`.

In `views/noteview.hbs`, we have a section using `$(document).ready` that contains the client code. In that function, add a section that exists only if the `user` object exists, as follows:

```
$(document).ready(function () {
  ...
  {{#if user}}
  ...
  {{/if}}
});
```

That is, we want a section of jQuery code that's active only when there is a `user` object, meaning that this Note is being shown to a logged-in user.

Within that section, add this event handler:

```
$('#submitNewComment').on('click', function(event) {
  socket.emit('create-message', {
    from: $('#comment-from').val(),
    namespace: $('#comment-namespace').val(),
    room: $('#comment-room').val(),
    key: $('#comment-key').val(),
    message: $('#noteCommentTextArea').val()
  },
  response => {
    $('#notes-comment-modal').modal('hide');
    $('#noteCommentTextArea').empty();
  });
});
```

This matches the button in the form we just created. Normally in the event handler for a `type="submit"` button, we would use `event.preventDefault` to prevent the normal result, which is to reload the page. But that's not required in this case.

The function gathers various values from the form elements and sends the `create-message` event. If we refer back to the server-side code, `create-message` calls `postMessage`, which saves the message to the database, which then sends a `newmessage` event, which makes its way to the browser.

Therefore, we will need a `newmessage` event handler, which we'll get to in the next section. In the meantime, you should be able to run the Notes application, add some messages, and see they are added to the database.

Notice that this has a third parameter, a function that when called causes the Modal to be dismissed, and clears any message that was entered. This is the acknowledgment function we mentioned earlier, which is invoked on the server, and Socket.IO arranges to then invoke it here in the client.

Showing any existing messages on the Note view page

Now that we can add messages, let's learn how to display messages. Remember that we've defined an SQMessage schema and that we've defined a function, recentMessages, to retrieve the recent messages.

We have two possible methods to display existing messages when rendering Note pages. One option is for the page, when it initially displays, to send an event requesting the recent messages, and rendering those messages on the client once they're received. The other option is to render the messages on the server, instead. We've chosen the second option, server-side rendering.

In `routes/notes.mjs`, modify the `/view` router function like so:

```
router.get('/view', async (req, res, next) => {
  try {
    const note = await notes.read(req.query.key);
    const messages = await recentMessages('/notes', req.query.key);
    res.render('noteview', {
      title: note ? note.title : "",
      notekey: req.query.key,
      user: req.user ? req.user : undefined,
      note, messages
    });
  } catch (err) { error(err); next(err); }
});
```

That's simple enough: we retrieve the recent messages, then supply them to the `noteview.hbs` template. When we retrieve the messages, we supply the `/notes` namespace and a room name of the note key. It is now up to the template to render the messages.

In the `noteview.hbs` template, just below the **delete**, **edit**, and **comment** buttons, add this code:

```
<div id="noteMessages">
{{#if messages}}
{{#each messages}}
```

```
<div id="note-message-{{ id }}" class="card">
  <div class="card-body">
    <h5 class="card-title">{{ from }}</h5>
    <div class="card-text">{{ message }}<br/>
      <small style="display: block">{{ timestamp }}</small>
    </div>
    <button type="button" class="btn btn-primary message-del">
      Delete
    </button>
  </div>
</div>
{{/each}}
{{/if}}
</div>
```

If there is a `messages` object, these steps through the array, and for each entry, it sets up a Bootstrap `card` component to display the message. The messages are displayed within `<div id="noteMessages">`, which we'll target in DOM manipulations later. The markup for each message comes directly from the Bootstrap documentation, with a few modifications.

In each case, the `card` component has an `id` attribute we can use to associate with a given message in the database. The `button` component will be used to cause a message to be deleted, and it carries data attributes to identify which message would be deleted.

With this, we can view a Note, and see any messages that have been attached. We did not select the ordering of the messages but remember that in `models/messages-sequelize.mjs` the database query orders the messages in reverse chronological order.

In any case, our goal was for messages to automatically be added without having to reload the page. For that purpose, we need a handler for the `newmessage` event, which is a task left over from the previous section.

Below the handler for the `submitNewComment` button, add this:

```
socket.on('newmessage', newmsg => {
  var msgtxt = [
    '<div id="note-message-%id%" class="card">',
    '<div class="card-body">',
    '<h5 class="card-title">%from%</h5>',
    '<div class="card-text">%message%<br/>
```

```
'<small style="display: block">%timestamp%</small>',
'</div>',
'<button type="button" class="btn btn-primary message-del
    -button" ,
    'data-id="%id%" data-namespace="%namespace%" ' ,
    'data-room="%room%">',
    'Delete',
'</button>',
'</div>',
'</div>
].join('\n')
.replace(/%id%/g, newmsg.id)
.replace(/%from%/g, newmsg.from)
.replace(/%namespace%/g, newmsg.namespace)
.replace(/%room%/g, newmsg.room)
.replace(/%message%/g, newmsg.message)
.replace(/%timestamp%/g, newmsg.timestamp);
$('#noteMessages').prepend(msgtxt);
});
```

This is a handler for the Socket.IO newmessage event. What we have done is taken the same markup as is in the template, substituted values into it, and used jQuery to prepend the text to the top of the noteMessages area.

Remember that we decided against using any ES6 goodness because a template string would sure be handy in this case. Therefore, we have fallen back on an older technique, the JavaScript `String.replace` method.

There is a common question, how do we replace multiple occurrences of a target string in JavaScript? You'll notice that the target `%id%` appears twice. The best answer is to use `replace(/pattern/g, newText)`; in other words, you pass a regular expression and specify the `g` modifier to make it a global action. To those of us who grew up using `/bin/ed` and for whom `/usr/bin/vi` was a major advance, we're nodding in recognition that this is the JavaScript equivalent to `s/pattern/newText/g`.

With this event handler, the message will now appear automatically when it is added by the user. Further, for another window simply viewing the Note the new message will appear automatically.

Because we use the jQuery `prepend` method, the message appears at the top. If you want it to appear at the bottom, then use `append`. And in `models/messages-sequelize.mjs`, you can remove the `DESC` attribute in `recentMessages` to change the ordering.

The last thing to notice is the markup includes a button with the `id="message-del-button"`. This button is meant to be used to delete a message, and in the next section, we'll implement that feature.

Deleting messages on the Notes view page

To make the `message-del-button` button active, we need to listen to click events on the button.

Below the `newmessage` event handler, add this button click handler:

```
$('button.message-del-button').on('click', function(event) {  
    socket.emit('delete-message', {  
        id: $(event.target).data('id'),  
        namespace: $(event.target).data('namespace'),  
        room: $(event.target).data('room')  
    })  
});
```

The `socket` object already exists and is the Socket.IO connection to the room for this Note. We send to the room a `delete-message` event giving the values stored in `data` attributes on the button.

As we've already seen, on the server the `delete-message` event invokes the `destroyMessage` function. That function deletes the message from the database and also emits a `destroymessage` event. That event is received in `routes/notes.mjs`, which forwards the message to the browser. Therefore, we need an event listener in the browser to receive the `destroymessage` event:

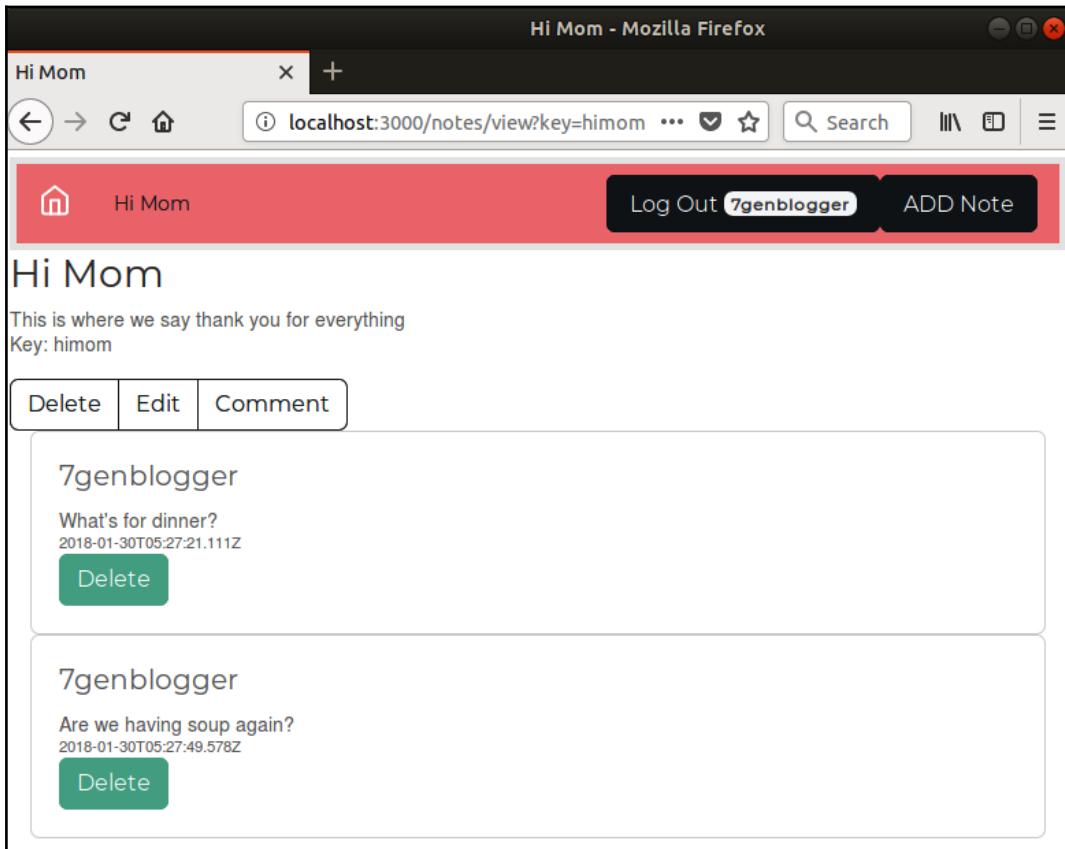
```
socket.on('destroymessage', data => {  
    $('#note-message-' + data.id).remove();  
});
```

Refer back and see that every message display card has an `id` parameter fitting the pattern shown here. Therefore, the jQuery `remove` function takes care of removing the message from the display.

Running Notes and passing messages

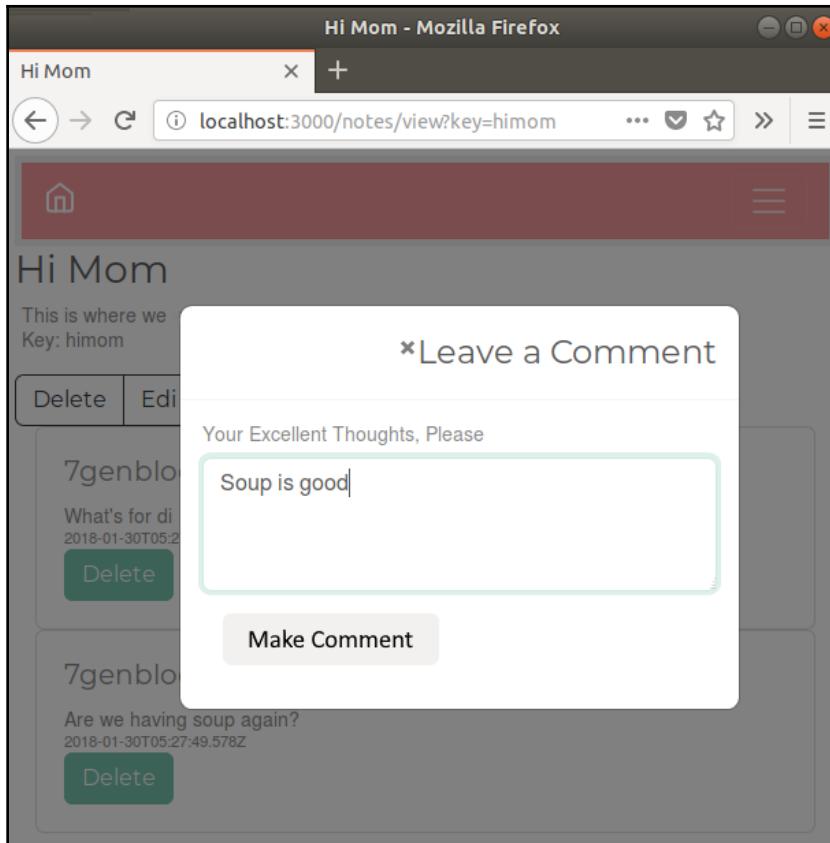
That was a lot of code, but we now have the ability to compose messages, display them on the screen, and delete them, all with no page reloads.

You can run the application as we did earlier, first starting the user authentication server in one command-line window and the Notes application in another:



It shows us any existing messages on a Note.

While entering a message, the Modal looks like this:



Try this with multiple browser windows viewing the same note or different notes. This way, you can verify that notes show up only on the corresponding note window.

Summary

We came a long way in this chapter, but maybe Facebook doesn't have anything to fear from the baby steps we took toward converting the Notes application into a social network. Still, we added interesting new features to the application, which gave us the opportunity to explore some really cool technology for pseudo-real-time communication between browser sessions.

We learned about using Socket.IO for pseudo-real-time web experiences. As we learned, it is a framework for dynamic interaction between server-side code and client code running in the browser. It follows an event-driven model for sending events between the two. Our code used this both for notifications to the browser of events occurring on the server and for users who wish to write comments.

We learned about the value of events being sent from one part of the server-side code to another. This lets us have client-side updates based on changes occurring in the server. This used the `EventEmitter` class with listener methods that directed events and data to the browser.

In the browser, we used jQuery DOM manipulation to change the user interface in response to these dynamically sent messages. By using Socket.IO and normal DOM manipulation, we were able to refresh the page content while avoiding page reloads.

We also learned about Modal windows, using that technique to create comments. Of course, there is much more that could be done, such as a different experience of creating, deleting, or editing notes.

To support all this, we added another kind of data, the *message*, and an accompanying database table, managed by a new Sequelize schema. It is used for representing the comments our users can make on notes, but is general enough to be used in other ways.

Socket.IO, as we've seen, gives us a rich foundation of events passing between server and client that can build multiuser, multichannel communication experiences for your users.

In the next chapter, we will look into Node.js application deployment on real servers. Running code on our laptop is cool, but to hit the big time, the application needs to be properly deployed.

3

Section 3: Deployment

In addition to the traditional method of deploying Node.js applications, using `systemd`, the new best practice is to use Kubernetes or similar systems.

This section comprises the following chapters:

- Chapter 10, *Deploying Node.js Applications to Linux Servers*
- Chapter 11, *Deploying Node.js Microservices with Docker*
- Chapter 12, *Deploying a Docker Swarm to AWS EC2 with Terraform*
- Chapter 13, *Unit Testing and Functional Testing*
- Chapter 14, *Security in Node.js Applications*

10

Deploying Node.js Applications to Linux Servers

Now that the Notes application is fairly complete, it's time to think about how to deploy it to a real server. We've created a minimal implementation of the collaborative note concept that works fairly well. To grow, *Notes* must escape our laptop and live on a real server.

The user story to fulfill is access to a hosted application that's available even when your laptop is turned off, for evaluation. The developer stories are to identify one of several deployment solutions, to have enough reliability so that the system restarts when it crashes, and for the users can access the app without taking too much of the developers time.

In this chapter, we will cover the following topics:

- A discussion of the application architecture, and thoughts on how to implement the deployment
- A traditional LSB-compliant Node.js deployment on a Linux server
- Configuring Ubuntu to manage background tasks
- Adjusting Twitter settings for application authentication
- Using PM2 to reliably manage background tasks
- Deployment to a virtual Ubuntu instance, which could be a **Virtual Machine (VM)** on our laptop or a **Virtual Private Server (VPS)** provider

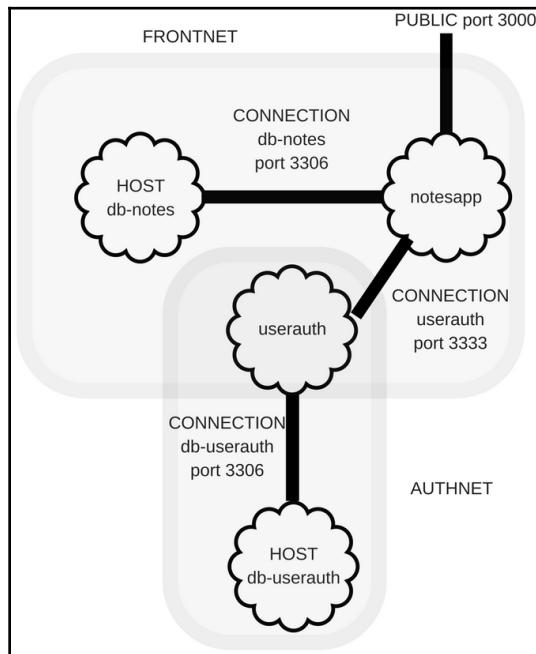
There are two services making up the *Notes* application: *Notes* itself, and the user authentication service, along with the corresponding database instances. For them to be reliably available to the users, these services must be deployed on servers visible on the public internet, along with system management tools to keep the services running, handle service failures, and scale the service up to handle large traffic loads. One common way to do this is the traditional method of relying on scripts executing during server boot-up to start the required background processes.

Even though our end goal is deployment on a cloud-based platform with auto-scaling and all the buzzwords, you must still start from the basics of how to get an application to run in the background on a Unix-like system.

Let's start the chapter by again reviewing the architecture, and think about how to best deploy on a server.

Notes application architecture and deployment considerations

Before we get into deploying the *Notes* application, we need to review its architecture and understand what we're planning to do. We have segmented the services into two groups, as shown in the following diagram:



The user-facing portion is the Notes service along with its database. The backend, the user authentication service, and its database require more security. On our laptop, we weren't able to create the envisioned protective wall around that service, but we're about to implement one form of such protection.

One strategy to enhance security is to expose as few ports as possible. That reduces the so-called attack surface, simplifying our work in hardening the application against security bugs. With the *Notes* application, we have exactly one port to expose: the HTTP service through which users access the application. The other ports – two for the MySQL servers, and one for the user authentication service port – should not be visible to the public internet since they are for internal use only. Therefore, in the final system, we should arrange to expose that one HTTP port and keep everything else walled off from the public internet.

Internally, the *Notes* application needs to access both the Notes database and the user authentication service. That service, in turn, needs to access the user authentication database. The Notes service does not need to access the user authentication database, and the user authentication service does not need to access the Notes database. As currently envisaged, no external access to either database or the authentication service is required.

This gives us a sense of what will be implemented. To get started, let's learn the traditional way to deploy applications on Linux.

Traditional Linux deployment for Node.js services

In this section, we will explore the traditional Linux/Unix service deployment. We'll do this with a virtual Ubuntu instance running on our laptop. The goal is to create background processes that automatically start during boot-up, restart if the process crashes, and allow us to monitor log files and system state.

Traditional Linux/Unix server application deployment uses an **init script** to manage background processes. They are to start every time the system boots, and cleanly shut down when the system is halted. The name "init script" comes from the name of the first process launched in the system, whose traditional name is `/etc/init`. The init scripts are usually stored in `/etc/init.d`, and are typically simple shell scripts. Some operating systems use other process managers, such as `upstart`, `systemd`, or `launchd`, while following the same model. While it's a simple model, the specifics of this vary widely from one **operating system (OS)** to another.

The Node.js project itself does not include any scripts to manage server processes on any OS. Implementing a complete web service based on Node.js means that we must create the scripting to integrate with process management on your OS.

Having a web service on the internet requires having background processes running on a server, and those processes have to be the following:

- **Reliable:** For example, they should auto-restart when the server process crashes.
- **Manageable:** They should integrate well with system management practices.
- **Observable:** The administrator must be able to get status and activity information from the service.

To demonstrate what's involved, we'll use PM2 to implement background server process management for *Notes*. PM2 bills itself as a *process manager*, meaning it tracks the state of processes it is managing and makes sure the processes execute reliably and are observable. PM2 detects the system type and can automatically integrate itself with the native process management system. It will create an LSB-style init script (<http://wiki.debian.org/LSBInitScripts>), or other scripts as required for your server.

Our goal in this chapter is exploring how to do this, and there are several routes to achieving this goal:

- Traditional VM management applications including VirtualBox, Parallels, and VMware let us install Ubuntu or any other OS within a virtual environment. On Windows, Hyper-V comes with Windows 10 Pro and offers a similar capability. In these cases, you download an ISO image of the boot CD-ROM, boot the VM from that ISO image, and run the full OS installation as if it was a regular computer.
- You can rent inexpensive VPSes from one of hundreds of web hosting providers around the world. Often the choice is limited to Ubuntu servers. In these cases, you're handed a pre-baked server system ready to go for installing server software to run websites.
- A new product, Multipass, is a lightweight VM management tool, based on lightweight hypervisor technology, and is available for every desktop computer OS. It gives you the exact same starting point as you'd get by renting a VPS or using VM software like VirtualBox, with a much lower system impact than traditional VM applications such as VirtualBox. It is like getting a VPS from a hosting provider, but it's on your laptop.

There is no practical difference between these choices from the standpoint of the tools and commands required to launch background processes. The Ubuntu instance installed in VirtualBox is the same as the Ubuntu on the VPS rented from a web-hosting provider, and is the same as the Ubuntu launched in a Multipass instance. It's the same OS, the same command-line tools, and the same system management practices. The difference is in the performance impact on your laptop. With Multipass, we can set up a virtual Ubuntu instance in a few seconds, and it is easy to have multiple instances running on a laptop with little or no performance impact. The experience of using VirtualBox, Hyper-V, or other VM solutions is that using the laptop feels quickly like walking through molasses, especially when running multiple VMs at once.

Therefore, in this chapter, we will run this exercise on Multipass. Everything shown in this chapter is easily transferrable to Ubuntu on VirtualBox/VMware/and so on or to a VPS rented from a web hosting provider.

For this deployment, we will create two Ubuntu instances with Multipass: one instance for the Notes service and the other for the user service. In each instance, there will be a MySQL instance for the corresponding database. Then we'll use PM2 to configure these systems to start our services in the background when launched.

Because of apparent incompatibilities between Multipass and WSL2, there might be difficulties using Multipass on Windows. If you run into problems, we have a section describing what to do.

The first task is to duplicate the source code from the previous chapter. It's suggested you create a new directory, `chap10`, as a sibling of the `chap09` directory, and copy everything from `chap09` to `chap10`.

To get started, let's install Multipass, and after that we'll start by deploying and testing the user authentication service, followed by deploying and testing Notes. We'll also cover setup issues on Windows.

Installing Multipass

Multipass is an open source tool developed by Canonical. It is an extremely lightweight tool for managing VMs, specifically Ubuntu-based VMs. It is light enough to enable running a mini-sized cloud hosting system on your laptop.



To install Multipass, get an installer from <https://multipass.run/>. It may also be available through package management systems.

With Multipass installed, you can run some of the following commands to try it out:

```
$ multipass launch
Launched: pleased-mustang
$ multipass list
Name          State    IPv4           Image
pleased-mustang Running 192.168.64.5 Ubuntu 18.04 LTS
$ multipass exec pleased-mustang -- lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description: Ubuntu 18.04.3 LTS
Release: 18.04
Codename: bionic
$ multipass shell pleased-mustang
Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 4.15.0-76-generic x86_64)

...
ubuntu@pleased-mustang:~$
```

Because we did not supply a name for the machine, Multipass created a random name. It isn't shown in the preceding snippet, but the first command included the download and setup of a VM image. The `shell` command starts a login shell inside the newly created VM, where you can use tools like `ps` or `htop` to see that there is indeed a full complement of processes running already.

Since one of the first things you do with a new Ubuntu install is to update the system, let's do so the Multipass way:

```
$ multipass exec pleased-mustang -- sudo apt-get update
... much output
$ multipass exec pleased-mustang -- sudo apt-get upgrade
... much output
```

This works as expected, in that you see `apt-get` first update its list of available packages, and then ask you to approve downloading and installing the packages to update, after which it does so. Anyone who is familiar with Ubuntu will find this normal. The difference is doing this from the command-line environment of the host computer.

That was fun, but we have some work to do, and we're not pleased with this mustang-based machine name Multipass saddled us with. Let's learn how to delete Multipass instances:

```
$ multipass list
Name          State    IPv4           Image
pleased-mustang  Running 192.168.64.5 Ubuntu 18.04 LTS
$ multipass delete pleased-mustang
$ multipass list
Name          State    IPv4 Image
pleased-mustang Deleted -- Not Available
$ multipass purge
$ multipass list
Name State IPv4 Image
```

We can easily delete a VM image with the `delete` command; it is then marked as `Deleted`. To truly remove the VM, we must use the `purge` command.

We've learned how to create, manage, and delete VMs using Multipass. This was a lot faster than some of the alternative technologies. With VirtualBox, for example, we would have had to find and download an ISO, then boot a VirtualBox VM instance and run the Ubuntu installer, taking a lot more time.

There might be difficulties using Multipass on Windows, so let's talk about that and how to rectify it.

Handling a failure to launch Multipass instances on Windows

The Multipass team makes their application available to on run Windows systems, but issues like the following can crop up:

```
PS C:\Users\david> multipass launch
launch failed: The following errors occurred:
unassuming-tailorbird: timed out waiting for response
```

It goes through all the steps of setting up an instance, but in the last step, we get this message instead of success. Running `multipass list` might show the instance in a `Running` state, but no IP address has been assigned, and running `multipass shell` also results in a timeout.

This timeout is observed if WSL2 is installed on the computer along with Multipass. WSL2 is a lightweight Linux subsystem for Windows, that is billed as an excellent environment for running Linux commands on Windows. Running WSL2 and Multipass at the same time may result in unwanted behavior.

For the purposes of this chapter, WSL2 is not useful. This is because WSL2 does not, at this time, support installing a background service that persists after a reboot, because it does not support `systemd`. Remember that our goal is to learn about setting up persistent background services.

It may be necessary to disable WSL2. To do so, use the **Search** box in the Windows taskbar to look for the **Turn Windows Features On or Off** control panel. Because WSL2 is a feature rather than an application that is installed or uninstalled, it is turned off or on using this control panel. Simply scroll down to find the feature, untick the checkbox, and then reboot the computer.



The Multipass online documentation has a troubleshooting page for Windows that has some useful hints, at <https://multipass.run/docs/troubleshooting-networking-on-windows>.

Both WSL2 and Multipass use Hyper-V. This is a virtualization engine for Windows, and it also supports installing VMs in a mode similar to VirtualBox or VMware. It is easy to download an ISO for Ubuntu or any other OS and install it on Hyper-V. This results in a full OS in which to experiment with background process deployment. You may prefer to run these examples inside Hyper-V instead.

Once the virtual machine is installed most of the instructions in the rest of this chapter will work. Specifically, the `install-packages.sh` script will be useful for installing the Ubuntu packages required to complete the instructions, and the two `configure-svc` scripts are useful for "deploying" the services into `/opt/notes` and `/opt/userauth`. It is recommended to use Git inside the virtual machine to clone the repository associated with this book. Finally, the scripts in the `pm2-single` directory are useful for running the Notes and Users services under PM2.

Our purpose is to learn how to deploy Node.js services on a Linux system, without having to leave our laptop. For that purpose, we've familiarized ourselves with Multipass since it is an excellent tool for managing Ubuntu instances. We've also learned about alternatives such as Hyper-V or VirtualBox that also can be used for managing Linux instances.

Let's start exploring deployment with the user authentication service.

Provisioning a server for the user authentication service

Since we want to have a segmented infrastructure, with the user authentication service in a walled-off area, let's make the first attempt at building that architecture. Using Multipass we will create two server instances, `svc-userauth` and `svc-notes`. Each will contain its own MySQL instance and the corresponding Node.js-based service. In this section, we'll set up `svc-userauth`, then in another section, we'll replicate the process to set up `svc-notes`.

Feeling kindly to our DevOps team, who've requested automation for all administrative tasks, we'll create some shell scripts to manage the server setup and configuration.

The scripts shown here handle deployment to two servers, with one server holding the authentication service and the other holding the *Notes* application. In the GitHub repository accompanying this book, you'll find other scripts to handle deployment to a single server. The single server scenario might be required if you're using a heavier-weight virtualization tool such as VirtualBox rather than Multipass.

In this section, we will create the user authentication backend server, `svc-userauth`, and in a later section, we will create the server for the *Notes* frontend, `svc-notes`. Since the two server instances will be set up similarly, we might question why we're setting up two servers. It's because of the security model we decided on.

There are several steps involved, including a few scripts for automating Multipass operations, as follows:

1. Create a directory named `chap10/multipass` for scripts related to managing Multipass instances.
2. Then, in that directory, create a file named `create-svc-userauth.sh`, containing the following:

```
multipass launch --name svc-userauth
multipass mount ../users svc-userauth:/build-users
multipass mount `pwd` svc-userauth:/build
```

On Windows, instead create a file named `create-svc-userauth.ps1` containing the following:

```
multipass launch --name svc-userauth
multipass mount ../users svc-userauth:/build-users
multipass mount (get-location) svc-userauth:/build
```

The two are nearly the same, except for the method to compute the current directory.

The `mount` command in Multipass attaches a host directory into the instance at the given location. Therefore, we attach the `multipass` directory as `/build` and `users` as `/build-users`.

The ``pwd`` notation is a feature of the Unix/Linux shell environment. It means to run the `pwd` process and capture its output, supplying it as a command-line argument to the `multipass` command. For Windows, we use `(get-location)` for the same purpose in PowerShell.

3. Create the instance by running the script:

```
$ sh ./create-svc-userauth.sh
```

Or, on Windows, run this:

```
PS C:\Path> .\create-svc-userauth.ps1
```

Either one runs the commands in the scripts that will launch the instance and mount directories from the host filesystem.

4. Create a file named `install-packages.sh` containing the following:

```
curl -sL https://deb.nodesource.com/setup_14.x | sudo -E bash  
-  
sudo apt-get update  
sudo apt-get upgrade -y  
sudo apt-get install -y nodejs build-essential mysql-server  
mysql-client
```

This installs Node.js 14.x and sets up other packages required to run the authentication service. This includes a MySQL server instance and the MySQL client.



The Node.js documentation (<https://nodejs.org/en/download/package-manager/>) has documentation on installing Node.js from package managers for several OSes. This script uses the recommended installation for Debian and Ubuntu systems because that's the OS used in the Multipass instance.

A side effect of installing the `mysql-server` package is that it launches a running MySQL service with a default configuration. Customizing that configuration is up to you, but for our purposes here and now, the default configuration will work.

5. Execute this script inside the instance like so:

```
$ multipass exec svc-userauth -- sh -x /build/install-packages.sh
```

The `exec` command, as we discussed earlier, causes a command to execute inside the container by running this command on the host system.

6. In the `users` directory, edit `user-server.mjs` and change the following:

```
server.listen(process.env.PORT,  
  process.env.REST_LISTEN ? process.env.REST_LISTEN :  
  'localhost',  
  function() {  
    log(`\$${server.name} listening at ${server.url}`);  
  });
```

Previously, we had specified a hardcoded '`localhost`' here. The effect of this was that the user authentication service only accepted connections from the same computer. To implement our vision of *Notes* and the user authentication services running on different computers, this service must support connections from elsewhere.

This change introduces a new environment variable, `REST_LISTEN`, where we will declare where the server should listen for connections.

As you edit the source files, notice that the changes are immediately reflected inside the Multipass machine in the `/build-users` directory.

7. Create a file called `users/sequelize-mysql.yaml` containing the following:

```
dbname: userauth  
username: userauth  
password: userauth  
params:  
  host: localhost  
  port: 3306  
  dialect: mysql
```

This is our configuration for allowing the user service to connect with a local MySQL instance. The `dbname`, `username`, and `password` parameters must match the values in the configuration script shown earlier.

8. Then, in the `users/package.json` file, add these entries to the `scripts` section:

```
"scripts": {  
    "start": "cross-env DEBUG=users:* PORT=5858  
SEQUELIZE_CONNECT=sequelize-sqlite.yaml node ./user-  
server.mjs",
```

The `on-server` script contains the runtime configuration we'll use on the server.

9. Next, in the `users` directory, run this command:

```
$ npm install mysql2 --save
```

Since we're now using MySQL, we must have the driver package installed.

10. Now create a file named `configure-svc-userauth.sh` containing the following:

```
### Create the database for the UserAuthentication service  
  
sudo mysql --user=root <<EOF  
CREATE DATABASE userauth;  
CREATE USER 'userauth'@'localhost' IDENTIFIED BY 'userauth';  
GRANT ALL PRIVILEGES ON userauth.* TO 'userauth'@'localhost'  
WITH GRANT OPTION;  
EOF  
  
### Set up the UserAuthentication service code  
  
sudo mkdir -p /opt/userauth  
sudo chmod 777 /opt/userauth  
(cd /build-users; tar cf - .) | (cd /opt/userauth; tar xf -)  
(  
    cd /opt/userauth  
    rm -rf node_modules package-lock.json users  
    -sequelize.sqlite3  
    npm install  
)
```

This script is meant to execute inside the Ubuntu system managed by Multipass. The first section sets a user identity in the database. The second section copies the user authentication service code, from `/build-users` to `/userauth`, into the instance, followed by installing the required packages.

Since the MySQL server is already running, the `mysql` command will access the running server to create the database, and create the `userauth` user. We will use this user ID to connect with the database from the user authentication service.

But, why are some files removed before copying them into the instance? The primary goal is to delete the `node_modules` directory; the other files are simply unneeded. The `node_modules` directory contains modules that were installed on your laptop, and surely your laptop has a different OS than the Ubuntu instance running on the server? Therefore, rerunning `npm install` on the Ubuntu server ensures the packages are installed correctly.

11. Run the `configure-svc-userauth` script like so:

```
$ multipass exec svc-userauth -- sh -x /build/configure-svc-userauth.sh
```

Remember that the `multipass` directory in the source tree is mounted inside the instance as `/build`. As soon as we created this file, it showed up in the `/build` directory, and we can execute it inside the instance.

Several times in this book, we've talked about the value of explicitly declaring all dependencies and of automating everything. This demonstrates this value, because now, we can just run a couple of shell scripts and the server is configured. And we don't have to remember how to launch the server because of the `scripts` section in `package.json`.

12. We can now start the user authentication server, like so:

```
$ multipass shell svc-userauth
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-96-generic
x86_64)

...
ubuntu@svc-userauth:~$ cd /opt/userauth/
ubuntu@svc-userauth:/opt/userauth$ npm run on-server

> user-auth-server@1.0.0 on-server /opt/userauth
> DEBUG=users:* REST_LISTEN=0.0.0.0 PORT=5858
```

```
SEQUELIZE_CONNECT=sequelize-mysql.yaml node ./user-server.mjs

users:service User-Auth-Service listening at
http://0.0.0.0:5858
```

Notice that our notation is to use \$ to represent a command typed on the host computer, and ubuntu@svc-userauth:~\$ to represent a command typed inside the instance. This is meant to help you understand where the commands are to be executed.

In this case, we've logged into the instance, changed directory to /opt/userauth, and started the server using the corresponding npm script.

Testing the deployed user authentication service

Our next step at this point is to test the service. We created a script, cli.mjs, for that purpose. In the past, we ran this script on the same computer where the authentication service was running. But this time, we want to ensure the ability to access the service remotely.

Notice that the URL printed is http://[::]:5858. This is shorthand for listening to connections from any IP address.

On our laptop, we can see the following:

```
$ multipass list
Name      State    IPv4          Image
svc-userauth Running 192.168.64.8 Ubuntu 18.04 LTS
```

Multipass assigned an IP address to the instance. Your IP address will likely be different.

On our laptop is a copy of the source code, including a copy of cli.mjs. This means we can run cli.mjs on our laptop, telling it to access the service on svc-userauth. That's because we thought ahead and added --host and --port options to cli.mjs. In theory, using those options, we can access this server anywhere on the internet. At the moment, we simply need to reach into the virtual environment on our laptop.

On your laptop, in the regular command environment rather than inside Multipass, run these commands:

```
$ node cli.mjs --host 192.168.64.8 --port 5858 add --password w0rd --
family-name Einarsson --given-name Ashildr --email me@stolen.tardis
```

```
me
Created {
  id: 'me',
  username: 'me',
  provider: 'local',
  familyName: 'Einarsdottir',
  givenName: 'Ashildr',
  middleName: null,
  emails: [ 'me@stolen.tardis' ],
  photos: []
}
$ node cli.mjs --host 192.168.64.8 --port 5858 add --password foooo --
family-name Smith --given-name John --middle-name Snuffy --email
snuffy@example.com snuffy-smith
Created {
  id: 'snuffy-smith',
  username: 'snuffy-smith',
  provider: 'local',
  familyName: 'Smith',
  givenName: 'John',
  middleName: 'Snuffy',
  emails: [ 'snuffy@example.com' ],
  photos: []
}
```

Make sure to specify the correct host IP address and port number.

If you remember, the script retrieves the newly created user entry and prints it out. But we need to verify this and can do so using the `list-users` command. But let's do something a little different, and learn how to access the database server.

In another command window on your laptop, type these commands:

```
$ multipass shell svc-userauth
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-76-generic x86_64)
...
ubuntu@userauth:~$ mysql -u userauth -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
...
mysql> USE userauth;
mysql> SHOW tables;
+-----+
| Tables_in_userauth |
+-----+
| SQUsers           |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT id,username,familyName,givenName,emails FROM SQUsers;
+----+-----+-----+-----+
| id | username      | familyName    | givenName | emails
|----+-----+-----+-----+
| 1  | me           | Einarssdottir | Ashildr   | ["me@stolen.tardis"]
|----+-----+-----+-----+
| 2  | snuffy-smith | Smith        | John       |
|     | ["snuffy@example.com"] |
+----+-----+-----+-----+
---+
2 rows in set (0.00 sec)
```

This shows database entries for the users we created. Notice that while logged in to the Multipass instance, we can use any Ubuntu command because we have the full OS in front of us.

We have not only launched the user authentication service on an Ubuntu server, but we've verified that we can access that service from outside the server.

In this section, we set up the first of the two servers we want to run. We still have to create the `svc-notes` server.

But before we do that, we first need to discuss running scripts on Windows.

Script execution in PowerShell on Windows

In this chapter, we'll write several shell scripts. Some of these scripts need to run on your laptop, rather than on a Ubuntu-hosted server. Some developers use Windows, and therefore we need to discuss running scripts on PowerShell.

Executing scripts on Windows is different because it uses PowerShell rather than Bash, along with a large number of other considerations. For this and the scripts that follow, make the following changes.

PowerShell script filenames must end with the `.ps1` extension. For most of these scripts, all that is required is to duplicate the `.sh` scripts as `.ps1` files, because the scripts are so simple. To execute the script, simply type `.\scriptname.ps1` in the PowerShell window. In other words, on Windows, the script just shown must be named `configure-svc-userauth.ps1`, and is executed as `.\configure-svc-userauth.ps1`.

To execute the scripts, you may need to change the PowerShell execution policy:

```
PS C:\Users\david\chap10\authnet> Get-ExecutionPolicy  
Restricted  
PS C:\Users\david\chap10\authnet> Set-ExecutionPolicy Unrestricted
```

Obviously, there are security considerations with this change, so change the execution policy back when you're done.

A simpler method on Windows is to simply paste these commands into a PowerShell window.

It was useful to discuss script execution on PowerShell. Let's return to the task at hand, which is provisioning the Notes stack on Ubuntu. Since we have a functioning user authentication service, the remaining task is the Notes service.

Provisioning a server for the Notes service

So far, we have set up the user authentication service on Multipass. Of course, to have the full Notes application stack running, the Notes service must also be running. So let's take care of that now.

The first server, `svc-userauth`, is running the user authentication service. Of course, the second server will be called `svc-notes`, and will run the Notes service. What we'll do is very similar to how we set up `svc-userauth`.

There are several tasks in the `multipass` directory to prepare this second server. As we did with the `svc-userauth` server, here, we set up the `svc-notes` server by installing and configuring required Ubuntu packages, then set up the Notes application:

1. Create a script named `multipass/create-svc-notes.sh` containing the following:

```
multipass launch --name svc-notes  
multipass mount ../notes svc-notes:/build-notes  
multipass mount `pwd` svc-notes:/build
```

This is tasked with launching the Multipass instance, and is very similar to `create-svc-userauth` but changed to use the word `notes`.

For Windows, instead create a file called `multipass/create-svc-notes.ps1` containing the following:

```
multipass launch --name svc-notes
multipass mount ..../notes svc-notes:/build-notes
multipass mount (get-location) svc-notes:/build
```

This is the same as before, but using `(get-location)` this time.

2. Create the instance by running the script as follows:

```
$ sh ./create-svc-notes.sh
```

Or, on Windows, run the following command:

```
PS C:\Path> .\create-svc-notes.ps1
```

Either one runs the commands in the scripts that will launch the instance and mount directories from the host filesystem.

3. Install the required packages like so:

```
$ multipass exec svc-notes -- sh -x /build/install-packages.sh
```

This script installs Node.js, the MySQL server, and a few other required packages.

4. Now create a file, `notes/models/sequelize-mysql.yaml`, containing the following:

```
dbname: notes
username: notes
password: notes
params:
  host: localhost
  port: 3306
  dialect: mysql
```

This is the database name, username, and password credentials for the database configured previously.

5. Because we are now using MySQL, run this command:

```
$ npm install mysql2 --save
```

We need the MySQL driver package to use MySQL.

6. Then, in the `notes/package.json` file, add this entry to the `scripts` section:

```
"on-server": "cross-env DEBUG=notes:*
  SEQUELIZE_CONNECT=models/sequelize-mysql.yaml
  NOTES_MODEL=sequelize TWITTER_CALLBACK_HOST=
  http://172.23.89.142:3000 USER_SERVICE_URL=
  http://172.23.83.119:5858 PORT=3000 node./app.mjs"
```

This uses the new database configuration for the MySQL server and the IP address for the user authentication service. Make sure that the IP address matches what Multipass assigned to `svc-userauth`.

You'll, of course, get the IP address in the following way:

```
$ multipass list
Name      State    IPv4          Image
svc-notes Running 172.23.89.142 Ubuntu 18.04 LTS
svc-userauth Running 172.23.83.119 Ubuntu 18.04 LTS
```

The `on-server` script will have to be updated appropriately.

7. Duplicate `multipass/configure-svc-userauth.sh` to create a script named `multipass/configure-svc-notes.sh`, and change the final two sections to the following:

```
### Create the database for the UserAuthentication service

sudo mysql --user=root <<EOF
CREATE DATABASE notes;
CREATE USER 'notes'@'localhost' IDENTIFIED BY 'notes';
GRANT ALL PRIVILEGES ON notes.* TO 'notes'@'localhost' WITH
GRANT OPTION;
EOF

### Set up the UserAuthentication service code

sudo mkdir -p /opt/notes
sudo chmod 777 /opt/notes
(cd /build-notes; tar cf - .) | (cd /opt/notes; tar xf -)
(
```

```
cd /opt/notes
rm -rf node_modules package-lock.json *.sqlite3
npm install
)
```

This is also similar to what we did for `svc-userauth`. This also changes things to use the word `notes` where we used `userauth` before.

Something not explicitly covered here is ensuring the `.env` file you created to hold Twitter secrets is deployed to this server. We suggested ensuring this file is not committed to a source repository. That means you'll be handling it semi-manually perhaps, or you'll have to use some developer ingenuity to create a process for managing this file securely.

8. Run the `configure-svc-notes` script like so:

```
$ multipass exec svc-notes-- sh -x /build/configure-svc-
notes.sh
```

Remember that the `multipass` directory in the source tree is mounted inside the instance as `/build`. As soon as we created this file, it showed up in the `/build` directory, and we can execute it inside the instance.

9. We can now run the Notes service with the following command:

```
$ multipass shell svc-notes
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-96-generic
x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

...
ubuntu@svc-notes:~$ cd /opt/notes/
approotdir.mjs minty node_modules package.json public theme
ubuntu@svc-notes:/opt/notes$ npm run on-server

> notes@0.0.0 on-server /opt/notes
> cross-env DEBUG=notes:* SEQUELIZE_CONNECT=models/sequelize-
mysql.yaml NOTES_MODEL=sequelize
TWITTER_CALLBACK_HOST=http://172.23.89.142:3000
USER_SERVICE_URL=http://172.23.83.119:5858 PORT=3000 node
./app.mjs

notes:debug Listening on port 3000 +0ms
notes:notes-store [Module] { SQNote: SQNote, default:
```

```
[Function: SequelizeNotesStore] } +0ms
notes:notes-store [Function: SequelizeNotesStore] +14ms
notes:debug Using NotesStore [object Object] +0ms
```

As with `svc-userauth`, we shell into the server, change the directory to `/opt/notes`, and run the `on-server` script. If you want Notes to be visible on port 80, simply change the `PORT` environment variable. After that, the URL in the `TWITTER_CALLBACK_HOST` variable must contain the port number on which Notes is listening. For that to work, the `on-server` script needs to run as `root`, so therefore we will run the following:

```
ubuntu@svc-notes:/opt/notes$ sudo npm run on-server
```

The change is to use `sudo` to execute the command as `root`.

To test this we must of course use a browser to connect with the Notes service. For that, we need to use the IP address for `svc-notes`, which we learned from Multipass earlier. Using that example, the URL is `http://172.23.89.142:3000`.

You'll find that since we haven't changed anything in the look-and-feel category, that our *Notes* application looks like it has all along. Functionally, you will be unable to log in using Twitter credentials, but you can log in using one of the local accounts we created during testing.

Once both services are running, you can use your browser to interact with the *Notes* application and run it through its paces.

What we've done is build the second of two servers, `svc-userauth` and `svc-notes`, on which we'll run the Notes application stack. That gives us two Ubuntu instances each of which are configured with a database and a Node.js service. We were able to manually run the authentication and Notes services together, connecting from one Ubuntu instance to the other, each working with their corresponding database. To have this as a fully deployed server, we will use PM2 in a later section.

We have learned a little about configuring Ubuntu servers, though there is an outstanding issue of running the services as background processes. Before we get to that, let's rectify the situation with the Twitter login functionality. The issue with Twitter login is that the application is now on a different IP address, so to resolve this, we now have to add that IP address in Twitter's management backend.

Adjusting Twitter authentication to work on the server

As we just noted, the *Notes* application as currently deployed does not support Twitter-based logins. Any attempt will result in an error. Obviously we can't deploy it like this.

The Twitter application we set up for *Notes* previously won't work because the authentication URL that refers to our laptop is incorrect for the server. To get OAuth to work with Twitter, while deployed on this new server, go to developer.twitter.com/en/apps and reconfigure the application to use the IP address of your server.

That page is the dashboard of your applications that you've registered with Twitter. Click on the **Details** button, and you'll see the details of the configuration. Click on the **Edit** button, and edit the list of **Callback URLs** like so:

```
Callback URL
http://localhost:3000
http://localhost:3000/users/auth/twitter/callback
http://localhost:3000/users/auth/twitter
http://192.168.64.9:3000/users/auth/twitter
http://192.168.64.9:3000/users/auth/twitter/callback
```

Of course, you must substitute the IP address of your server. The URL shown here is correct if your Multipass instance was assigned an IP address of 192.168.64.9. This informs Twitter of a new correct callback URL that will be used. Likewise, if you have configured *Notes* to listen to port 80, the URL you point Twitter to must also use port 80. You must update this list for any callback URL you use in the future.

The next thing is to change the *Notes* application so as to use this new callback URL on the `svc-notes` server. In `routes/users.mjs`, the default value was `http://localhost:3000` for use on our laptop. But we now need to use the IP address for the server. Fortunately, we thought ahead and the software has an environment variable for this purpose. In `notes/package.json`, add the following environment variable to the `on-server` script:

```
TWITTER_CALLBACK_HOST=http://192.168.64.9:3000
```

Use the actual IP address or domain name assigned to the server being used. In a real deployment, we'll have a domain name to use here.

Additionally, to enable Twitter login support, it is required to supply Twitter authentication tokens in the environment variables:

```
TWITTER_CONSUMER_KEY="... key" TWITTER_CONSUMER_SECRET="... key"
```

This should not be added in `package.json`, but supplied via another means. We have not yet identified a suitable method, but we did identify that adding these variables to `package.json` means committing them to a source code repository, which might allow those values to leak to the public.

For now, the server can be started as follows:

```
ubuntu@svc-notes:/opt/notes$ TWITTER_CONSUMER_KEY="... key"
TWITTER_CONSUMER_SECRET="... key" npm run on-server
```

This is still a semi-manual process of starting the server and specifying the Twitter keys, but you'll be able to log in using Twitter credentials. Keep in mind that we still need a solution for this that avoids committing these keys to a source repository.

The last thing for us to take care of is ensuring the two service processes restart when the respective servers restart. Right now, the services are running at the command line. If we ran `multipass restart`, the service instances will reboot and the service processes won't be running.

In the next section, we'll learn one way to configure a background process that reliably starts when a computer is booted.

Setting up PM2 to manage Node.js processes

We have two servers, `svc-notes` and `svc-userauth`, configured so we can run the two services making up the Notes application stack. A big task remaining is to ensure the Node.js processes are properly installed as background processes.

To see the problem, start another command window and run these commands:

```
$ multipass restart svc-userauth  
$ multipass restart svc-notes
```

The server instances were running under Multipass, and the `restart` command caused the named instance to `stop` and then `start`. This emulates a server reboot. Since both were running in the foreground, you'll see each command window exit to the host command shell, and running `multipass list` again will show both instances in the `Running` state. The big takeaway is that both services are no longer running.

There are many ways to manage server processes, to ensure restarts if the process crashes, and so on. We'll use **PM2** (<http://pm2.keymetrics.io/>) because it's optimized for Node.js processes. It bundles process management and monitoring into one application.

Let's now see how to use PM2 to correctly manage the Notes and user authentication services as background processes. We'll start by familiarizing ourselves with PM2, then creating scripts to use PM2 to manage the services, and finally, we'll see how to integrate it with the OS so that the services are correctly managed as background processes.

Familiarizing ourselves with PM2

To get ourselves acquainted with PM2, let's set up a test using the `svc-userauth` server. We will create a directory to hold a `pm2-userauth` project, install PM2 in that directory, then use it to start the user authentication service. Along the way, we'll learn how to use PM2.

Start by running these commands on the `svc-userauth` server:

```
ubuntu@svc-userauth:~$ sudo su -  
root@svc-userauth:~#
```

```
root@svc-userauth:~# cd /opt
root@svc-userauth:/opt# mkdir pm2-test
root@svc-userauth:/opt# cd pm2-test
root@svc-userauth:/opt/pm2-test# npm init
This utility will walk you through creating a package.json file.
... answer questions
root@svc-userauth:/opt/pm2-test# npm install pm2 --save
... installation output
root@svc-userauth:/opt/pm2-test# node_modules/.bin/pm2
... help output
```

The result of these commands is an npm project directory containing the PM2 program and a package.json file that we can potentially use to record some scripts.

Now let's start the user authentication server using PM2:

```
root@svc-userauth:/opt/pm2-test# cd ../userauth/
root@svc-userauth:/opt/userauth# DEBUG=users:* PORT=5858
REST_LISTEN=0.0.0.0 SEQUELIZE_CONNECT=sequelize-mysql.yaml /opt/pm2-
test/node_modules/.bin/pm2 start ./user-server.mjs
...
[PM2] Spawning PM2 daemon with pm2_home=/root/.pm2
[PM2] PM2 Successfully daemonized
[PM2] Starting /opt/userauth/user-server.mjs in fork_mode (1 instance)
[PM2] Done.
...
```

This boils down to running pm2 start ./user-server.mjs, except that we are adding the environment variables containing configuration values, and we are specifying the full path to PM2. This runs our user server in the background.

We can repeat our test of using cli.mjs to list users known to the authentication server:

```
root@svc-userauth:/opt/userauth# node cli.mjs list-users
[
  { ... }, { ... }
]
```

Since we had previously launched this service and tested it, there should be user IDs already in the authentication server database. The server is running, but because it's not in the foreground, we cannot see the output. Try this command:

```
root@svc-userauth:/opt/userauth# /opt/pm2-test/node_modules/.bin/pm2
logs user-server
... log output
```

Because PM2 captures the standard output from the server process, any output is saved away. The `logs` command lets us view that output.

Some other useful commands are as follows:

- `pm2 status`: Lists all the commands PM2 is currently managing, and their status
- `pm2 stop SERVICE`: Stops the named service
- `pm2 start SERVICE` or `pm2 restart SERVICE`: Starts the named service
- `pm2 delete SERVICE`: Makes PM2 forget about the named service

There are several other commands, and the PM2 website contains complete documentation for them. <https://pm2.keymetrics.io/docs/usage/pm2-doc-single-page/>

For the moment, let's shut it down and delete the managed process:

```
root@svc-userauth:/opt/userauth# /opt/pm2-test/node_modules/.bin/pm2
stop user-server
root@svc-userauth:/opt/userauth# /opt/pm2-test/node_modules/.bin/pm2
delete user-server
root@svc-userauth:/opt/userauth# rm -rf /opt/pm2-test
```

We have familiarized ourselves with PM2, but this setup is not quite suitable for any kind of deployment. Let's instead set up scripts that will manage the Notes services under PM2 more cleanly.

Scripting the PM2 setup on Multipass

We have two Ubuntu systems onto which we've copied the Notes and user authentication services, and also configured a MySQL server for each machine. On these systems, we've manually run the services and know that they work, and now it's time to use PM2 to manage these services as persistent background processes.

With PM2 we can create a file, `ecosystem.json`, to describe precisely how to launch the processes. Then, with a pair of PM2 commands, we can integrate the process setup so it automatically starts as a background process.

Let's start by creating two directories, `multipass/pm2-notes` and `multipass/pm2-userauth`. These will hold the scripts for the corresponding servers.

In pm2-notes, create a file, package.json, containing the following:

```
{  
  "name": "pm2-notes",  
  "version": "1.0.0",  
  "description": "PM2 Configuration for svc-notes",  
  "scripts": {  
    "start": "pm2 start ecosystem.json",  
    "stop": "pm2 stop ecosystem.json",  
    "restart": "pm2 restart ecosystem.json",  
    "status": "pm2 status",  
    "save": "pm2 save",  
    "startup": "pm2 startup",  
    "monit": "pm2 monit",  
    "logs": "pm2 logs"  
  },  
  "dependencies": {  
    "pm2": "^4.4.x"  
  }  
}
```

This records for us the dependency on PM2, so it can easily be installed, along with some useful scripts we can run with PM2.

Then in the same directory, create an ecosystem.json file, containing the following:

```
{  
  "apps" : [ {  
    "name": "Notes",  
    "script": "app.mjs",  
    "cwd": "/opt/notes",  
    "env": {  
      "REQUEST_LOG_FORMAT": "common",  
      "PORT": "80",  
      "SEQUELIZE_CONNECT": "models/sequelize-mysql.yaml",  
      "NOTES_MODEL": "sequelize",  
      "USER_SERVICE_URL": "http://172.23.83.119:5858",  
      "TWITTER_CALLBACK_HOST": "http://172.23.89.142"  
    },  
    "env_production": {  
      "NODE_ENV": "production"  
    }  
  } ]  
}
```

The ecosystem.json file is how we describe a process to be monitored to PM2.

In this case, we've described a single process, called `Notes`. The `cwd` value declares where the code for this process lives, and the `script` value describes which script to run to launch the service. The `env` value is a list of environment variables to set.

This is where we would specify the Twitter authentication tokens. But since this file is likely to be committed to a source repository, we shouldn't do so. Instead, we'll forego Twitter login functionality for the time being.

The `USER_SERVICE_URL` and `TWITTER_CALLBACK_HOST` variables are set according to the `multipass list` output we showed earlier. These values will, of course, vary based on what was selected by your host system.

These environment variables are the same as we set in `notes/package.json` – except, notice that we've set `PORT` to 80 so that it runs on the normal HTTP port. To successfully specify port 80, PM2 must execute as root.

In pm2-userauth, create a file named package.json containing the following:

```
{
  "name": "pm2-userauth",
  "version": "1.0.0",
  "description": "PM2 Configuration for svc-userauth",
  "scripts": {
    "start": "pm2 start ecosystem.json",
    "stop": "pm2 stop ecosystem.json",
    "restart": "pm2 restart ecosystem.json",
    "status": "pm2 status",
    "save": "pm2 save",
    "startup": "pm2 startup",
    "monit": "pm2 monit",
    "logs": "pm2 logs"
  },
  "dependencies": {
    "pm2": "^4.4.x"
  }
}
```

This is the same as for `pm2-notes`, with the names changed.

Then, in pm2-userauth, create a file named `ecosystem.json` containing the following:

```
{
  "apps" : [
    {
      "name": "User Authentication",
      "script": "user-server.mjs",
    }
  ]
}
```

```
"cwd": "/opt/userauth",
"env": [
    "REQUEST_LOG_FORMAT": "common",
    "PORT": "5858",
    "REST_LISTEN": "0.0.0.0",
    "SEQUELIZE_CONNECT": "sequelize-mysql.yaml"
],
"env_production": {
    "NODE_ENV": "production"
}
}]
```

This describes the user authentication service. On the server, it is stored in the `/userauth` directory and is launched using the `user-server.mjs` script, with that set of environment variables.

Next, on both servers create a directory called `/opt/pm2`. Copy the files in `pm2-notes` to the `/opt/pm2` directory on `svc-notes`, and copy the files in `pm2-userauth` to the `/opt/pm2` directory on `svc-userauth`.

On both `svc-notes` and `svc-userauth`, you can run these commands:

```
ubuntu@svc-userauth:/opt/pm2$ sudo npm install
...
ubuntu@svc-userauth:/opt/pm2$ sudo npm start

> pm2-userauth@1.0.0 start /pm2
> pm2 start ecosystem.json

[PM2] Spawning PM2 daemon with pm2_home=/home/ubuntu/.pm2
[PM2] PM2 Successfully daemonized
[PM2] [WARN] Applications User Authentication not running, starting...
[PM2] App [User Authentication] launched (1 instances)

ubuntu@svc-userauth:/opt/pm2$ sudo npm run status

> pm2-userauth@1.0.0 status /opt/pm2
> pm2 status
...
ubuntu@svc-userauth:/opt/pm2$ sudo npm run logs

> pm2-userauth@1.0.0 status /opt/pm2
> pm2 logs
...
```

Doing so starts the service running on both server instances. The `npm run logs` command lets us see the log output as it happens. We've configured both services in a more DevOps-friendly logging configuration, without the DEBUG log enabled, and using the *common* log format.

For testing, we go to the same URL as before, but to port 80 rather than port 3000.

Because the Notes service on `svc-notes` is now running on port 80, we need to update the Twitter application configuration again, as follows:

Callback URL
<code>http://192.168.64.9/users/auth/twitter</code>
<code>http://localhost:3000</code>
<code>http://192.168.64.9/users/auth/twitter/callback</code>
<code>http://localhost:3000/users/auth/twitter/callback</code>
<code>http://localhost:3000/users/auth/twitter</code>

This drops the port 3000 from the URLs on the server. The application is no longer on port 3000, but on port 80, and we need to tell Twitter about this change.

Integrating the PM2 setup as persistent background processes

The *Notes* application should be fully functioning. There is one remaining small task to perform, and that is to integrate it with the OS.

The traditional way on Unix-like systems is to add a shell script in a directory in `/etc`. The Linux community has defined the LSB Init Script format for this purpose, but since each OS has a different standard for scripts to manage background processes, PM2 has a command to generate the correct script for each.

Let's start with `svc-userauth`, and run these commands:

```
ubuntu@svc-userauth:/opt/pm2$ sudo npm run save  
> pm2-userauth@1.0.0 save /opt/pm2  
> pm2 save  
  
[PM2] Saving current process list...
```

```
[PM2] Successfully saved in /home/ubuntu/.pm2/dump.pm2
ubuntu@svc-userauth:/opt/pm2$ sudo npm run startup

> pm2-userauth@1.0.0 startup /opt/pm2
> pm2 startup

[PM2] Init System found: systemd
Platform systemd
... much output
Target path
/etc/systemd/system/pm2-root.service
Command list
[ 'systemctl enable pm2-root' ]
... much output
```

With `npm run save`, we run the `pm2 save` command. This command saves the current configuration into a file in your home directory.

With `npm run startup`, we run the `pm2 startup` command. This converts the saved current configuration into a script for the current OS that will manage the PM2 system. PM2, in turn, manages the set of processes you've configured with PM2.

In this case, it identified the presence of the `systemd` init system, which is the standard for Ubuntu. It generated a file, `/etc/systemd/system/pm2-root.service`, that tells Ubuntu about PM2. In amongst the output, it tells us how to use `systemctl` to start and stop the PM2 service.

Do the same on `svc-notes` to implement the background service there as well.

And now we can test restarting the two servers with the following commands:

```
$ multipass restart svc-userauth
$ multipass restart svc-notes
$ multipass list
Name      State    IPv4          Image
svc-notes  Running  192.168.64.9  Ubuntu 18.04 LTS
svc-userauth Running 192.168.64.8  Ubuntu 18.04 LTS
```

The machines should restart correctly, and with no intervention on our part, the services will be running. You should be able to put the *Notes* application through its paces and see that it works. The Twitter login functionality will not work at this time because we did not supply Twitter tokens.

It is especially informative to run this on each server:

```
ubuntu@svc-notes:~$ cd /opt/pm2  
ubuntu@svc-notes:/opt/pm2$ sudo ./node_modules/.bin/pm2 monit
```

The `monit` command starts a monitoring console showing some statistics including CPU and memory use, as well as logging output.

When done, run the following command:

```
$ multipass stop svc-userauth  
$ multipass stop svc-notes
```

This, of course, shuts down the service instances. Because of the work we've done, you'll be able to start them back up at any time.

We've learned a lot in this section about configuring the *Notes* application as a managed background process. With a collection of shell scripts and configuration files, we put together a system to manage these services as background processes using PM2. By writing our own scripts, we got a clearer idea of how the underlying plumbing works.

With that, we are ready to wrap up the chapter.

Summary

In this chapter, we started a journey to learn about deploying Node.js services to live servers. The goal was to learn deployment to cloud hosting, but to get there we learned the basics of getting reliable background processes on Linux systems.

We started by reviewing the Notes application architecture and seeing how that will affect deployment. That enabled us to understand the requirements for server deployment.

Then we learned the traditional way to deploy services on Linux using an init script. To do that, we learned how to use PM2 to manage processes, and used it to integrate as a persistent background process. PM2 is a useful tool for managing background processes on Unix/Linux systems. Deploying and managing persistence is a key skill for anyone developing web applications.

While that was performed on your laptop, the exact same steps could be used on a public server such as a VPS rented from a web hosting company. With a little bit of work, we could use these scripts to set up a test server on a public VPS. We do need to work on better automation since the DevOps team requires fully automated deployments.

Even in this age of cloud hosting platforms, many organizations deploy services using the same techniques we discussed in this chapter. Instead of cloud-based deployments, they rent one or a few VPSes. But even in cloud-based deployments using Docker, Kubernetes, and the like, the developer must know how to implement a persistent service on Unix-like systems. Docker containers are typically Linux environments, and must contain reliable persistent background tasks that are observable and maintainable.

In the next chapter, we will pivot to a different deployment technology: Docker. Docker is a popular system for packaging application code in a *container* that can be executed on our laptop or executed unchanged at scale on a cloud hosting platform.

11

Deploying Node.js Microservices with Docker

Now that we've experienced the traditional Linux way to deploy an application, let's turn to Docker, which is a popular new way to manage application deployment.

Docker (<http://docker.com>) is a cool new tool in the software industry. It is described as *an open platform for distributed applications for developers and sysadmins*. It is designed around Linux containerization technology and focuses on describing the configuration of software on any variant of Linux.

A Docker container is a running instantiation of a Docker image. A Docker image is a bundle containing a specific Linux OS, system configuration, and application configuration. Docker images are described using a **Dockerfile**, which is a fairly simple-to-write script describing how to build a Docker image. The Dockerfile starts by specifying a *base image* from which to build, meaning we derive Docker images from other images. The rest of the Dockerfile describes what files to add to the image, which commands to run in order to build or configure the image, which network ports to expose, which directories to mount in the image, and more.

Docker images are stored on a Docker Registry server, with each image stored in its own repository. The largest registry is Docker Hub, but there are also third-party registries available, including registry servers that you can install on your own hardware. Docker images can be uploaded to a repository, and, from the repository, deployed to any Docker server.

We instantiate a Docker image to launch a Docker container. Typically, launching a container is very fast, and often, containers are instantiated for a short time and then discarded when no longer needed.

A running container feels like a virtual server running on a virtual machine. However, Docker containerization is very different from a virtual machine system such as VirtualBox or Multipass. A container is not a virtualization of a complete computer. Instead, it is an extremely lightweight shell creating the appearance of an installed OS. For example, the processes running inside the container are actually running on the host OS with certain Linux technologies (cgroups, kernel namespaces, and so on) creating the illusion of running a specific Linux variant. Your host OS could be Ubuntu and the container OS could be Fedora or OpenSUSE, or even Windows; Docker makes it all work.

While Docker is primarily targeted at x86 flavors of Linux, it is available on several ARM-based OSes, as well as other processors. You can even run Docker on single-board computers, such as Raspberry Pi, for hardware-oriented **Internet of Things (IoT)** projects.

The Docker ecosystem contains many tools, and their number is increasing rapidly. For our purposes, we'll focus on the following two tools:

- **Docker Engine:** This is the core execution system that orchestrates everything. It runs on a Linux host system, exposing a network-based API that client applications use to make Docker requests, such as building, deploying, and running containers.
- **Docker Compose:** This helps you define, in a single file, a multi-container application with all of its dependencies defined.

There are other tools closely associated with Docker, such as Kubernetes, but it all starts with building a container to house your application. By learning about Docker, we learn how to containerize an application, a skill we can use with both Docker and Kubernetes.

Learning how to use Docker is a gateway to learning about other popular systems, such as Kubernetes or AWS ECS. These are two popular orchestration systems for managing container deployments at a large scale on cloud-hosting infrastructure. Typically, the containers are Docker containers, but they are deployed and managed by other systems, whether that is Kubernetes, ECS, or Mesos. That makes learning how to use Docker an excellent starting point for learning these other systems.

In this chapter, we will cover the following topics:

- Installing Docker on our laptop
- Developing our own Docker containers and using third-party containers
- Setting up the user authentication service and its database in Docker

- Setting up the Notes service and its database in Docker
- Deploying MySQL instances in Docker infrastructure, and data persistence for applications such as databases in Docker
- Using Docker Compose to describe the Docker deployment of a full application
- Scaling container instances in Docker infrastructure and using Redis to mitigate scaling issues

The first task is to duplicate the source code from the previous chapter. It's suggested that you create a new directory, `chap11`, as a sibling of the `chap10` directory and copy everything from `chap10` to `chap11`.

By the end of this chapter, you will have a solid grounding of using Docker, creating Docker containers, and using Docker Compose to manage the services required by the Notes application.

With the help of Docker, we will design, on our laptop, the system shown in the diagram in [Chapter 10, Deploying Node.js Applications to Linux Servers](#). That chapter, this one, and [Chapter 12, Deploying a Docker Swarm to AWS EC2 with Terraform](#), form an arc covering three styles of Node.js deployment to servers.

Setting up Docker on your laptop or computer

The best place to learn how to install Docker on your laptop is the Docker documentation. What we're looking for is the Docker **Community Edition (CE)**, which is all we need:

- macOS installation: <https://docs.docker.com/docker-for-mac/install/>
- Windows installation: <https://docs.docker.com/docker-for-windows/install/>
- Ubuntu installation: <https://docs.docker.com/install/linux/docker-ce/ubuntu/>

Instructions are also available for several other distributions. Some useful post-install Linux instructions are available at <https://docs.docker.com/install/linux/linux-postinstall/>.

Docker runs natively on Linux, and the installation is simply the Docker daemon and command-line tools. To run Docker on macOS or Windows, you need to install the **Docker for Windows** or **Docker for Mac** applications. These applications manage a virtual Linux environment in a lightweight virtual machine, within which is a Docker Engine instance running on Linux. In the olden days (a few years ago), we had to handcraft that setup. Thanks must be given to the Docker team, who have made this as easy as installing an application, and all the complexity is hidden away. The result is very lightweight, and Docker containers can be left running in the background with little impact.

Let's now learn how to install Docker on a Windows or macOS machine.

Installing and starting Docker with Docker for Windows or macOS

The Docker team has made installing Docker on Windows or macOS very easy. You simply download an installer and, as with most other applications, you run the installer. It takes care of installation and provides you with an application icon that is used to launch Docker. On Linux, the installation is a little more involved, so it is best to read and follow the official instructions.

Starting Docker for Windows or macOS is very simple, once you've followed the installation instructions. You simply find and double-click on the application icon. There are settings available so that Docker automatically launches every time you start your laptop.

On both Docker for Windows and Docker for Mac, the CPU must support **virtualization**. Bundled inside Docker for Windows and Docker for Mac is an ultra-lightweight hypervisor, which, in turn, requires virtualization support from the CPU.

For Windows, this may require a BIOS configuration. Refer to <https://docs.docker.com/docker-for-windows/troubleshoot/#virtualization-must-be-enabled> for more information.

For macOS, this requires hardware from 2010 or later, with Intel's hardware support for **Memory Management Unit (MMU)** virtualization, including **Extended Page Tables (EPTs)** and unrestricted mode. You can check for this support by running `sysctl kern.hv_support`. It also requires macOS 10.11 or later.

Having installed the software, let's try it out and familiarize ourselves with Docker.

Familiarizing ourselves with Docker

With the setup accomplished, we can use the local Docker instance to create Docker containers, run a few commands, and, in general, learn how to use it.

As in so many software journeys, this one starts with *Hello World*:

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:66ef312bbac49c39a89aa9bcc3cb4f3c9e7de3788c94415
          8df3ee0176d32b751
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working
correctly.

...
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
...
```

The `docker run` command downloads a Docker image, named on the command line, initializes a Docker container from that image, and then runs that container. In this case, the image, named `hello-world`, was not present on the local computer and had to be downloaded and initialized. Once that was done, the `hello-world` container was executed and it printed out these instructions.

The `docker run hello-world` command is a quick way to verify that Docker is installed correctly.

Let's follow the suggestion and start an Ubuntu container:

```
$ docker run -it ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
5c939e3a4d10: Pull complete
c63719cdbe7a: Pull complete
19a861ea6baf: Pull complete
651c9d2d6c4f: Pull complete
Digest: sha256:8d31dad0c58f552e890d68bbfb735588b6b820a46e4596
          72d96e585871acc110
Status: Downloaded newer image for ubuntu:latest
```

```

root@83058f30a327:/# uname -a
Linux 83058f30a327 4.14.131-linuxkit #1 SMP Fri Jul 19 12:31:17 UTC
2019 x86_64 x86_64 x86_64 GNU/Linux
root@83058f30a327:/# exit
$ uname -a
Darwin MacBook-Pro-4 17.7.0 Darwin Kernel Version 17.7.0: Thu Jan 23
07:05:23 PST 2020; root:xnu-4570.71.69~1/RELEASE_X86_64 x86_64

```

The `Unable to find image` phrase means that Docker has not downloaded the named image yet. Therefore, it downloaded not only the Ubuntu image but also the images it depends on. Any Docker image can be built in layers, meaning we always define an image in terms of a base image. In this instance, we see that the Ubuntu image required four layers in total.

Images are identified by an SHA-256 hash, and there is both a long-form identifier and a short-form identifier. We can see both the long and short identifiers in this output.

The `docker run` command downloads an image, configures it for execution, and executes the image. The `-it` flag means to run the image interactively in the terminal.

In the `docker run` command line, the part after the image name to execute is passed into the container as command options to execute. In this case, the command option says to run `bash`, which is the default command shell. Indeed, we were given a command prompt and can run Linux commands.

You can query your computer to see that while the `hello-world` container has executed and finished, it still exists:

MacBook-Pro-4:users david\$ docker ps						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
83058f30a327	ubuntu	"bash"	28 seconds ago	Up 22 seconds		zealous_darwin
MacBook-Pro-4:users david\$ docker ps -a						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
83058f30a327	ubuntu	"bash"	32 seconds ago	Up 27 seconds		zealous_darwin
f0c7e682799f	hello-world	"/hello"	About a minute ago	Exited (0)	About a minute ago	clever_napier
MacBook-Pro-4:users david\$						

The `docker ps` command lists the running Docker containers. As we see here, the `hello-world` container is no longer running, but the Ubuntu container is. With the `-a` switch, `docker ps` also shows those containers that exist but are not currently running.

The last column is the container name. Since we didn't specify a container name when launching the container, Docker created a semi-random name for us.

When you're done using a container, you can clean up with the following command:

```
$ docker rm clever_napier  
clever_napier
```

The `clever_napier` name is the container name automatically generated by Docker. While the image name was `hello-world`, that was not the container name. Docker generated the container name so that you have a more user-friendly identifier for the containers than the hex ID shown in the `CONTAINER_ID` column:

```
$ docker rm 83058f30a327  
83058f30a327
```

It's also possible to specify the hex ID. However, it is, of course, more user friendly to have a name for the container than a hex ID. When creating a container, it's easy to specify any container name you like.

We've installed Docker on our laptop or computer and tried a couple of simple commands to familiarize ourselves with Docker. Let's now get down to some work. We'll start by setting up the user authentication service in Docker containers.

Setting up the user authentication service in Docker

With all that theory spinning around in our heads, it's time to do something practical. Let's start by setting up the user authentication service. We'll call this AuthNet, and it comprises a MySQL instance to store the user database, the authentication server, and a private subnet to connect them.

It is best for each container to focus on providing one service. Having one service per container is a useful architectural decision because we can focus on optimizing each container for a specific purpose. Another rationale has to do with scaling, in that each service has different requirements to satisfy the traffic it serves. In our case, we might need a single MySQL instance, and 10 user authentication instances, depending on the traffic load.



There is a large library of predefined Docker images available on Docker Hub (<https://hub.docker.com>). It is best to reuse one of those images as a starting point to build our desired service.

The Docker environment lets us not only define and instantiate Docker containers but also the networking connections between containers. That's what we meant by a *private subnet* earlier. With Docker, we not only manage containers, but we can also configure subnets, data storage services, and more.

In the next few sections, we'll carefully dockerize the user authentication service infrastructure. We'll learn how to set up a MySQL container for Docker and launch a Node.js service in Docker.

Let's start by learning how to launch a MySQL container in Docker.

Launching a MySQL container in Docker

Among the publicly available Docker images, there are over 11,000 available for MySQL. Fortunately, the image provided by the MySQL team, `mysql/mysql-server`, is easy to use and configure, so let's use that.

A Docker image name can be specified, along with a *tag* that is usually the software version number. In this case, we'll use `mysql/mysql-server:8.0`, where `mysql/mysql-server` is the image repository URL, `mysql-server` is the image name, and `8.0` is the tag. The MySQL 8.x release train is the current version as of the time of writing. As with many projects, the MySQL project tags the Docker images with the version number.

Download the image, as follows:

```
$ docker pull mysql/mysql-server:8.0
8.0.13: Pulling from mysql/mysql-server
e64f6e679e1a: Pull complete
799d60100a25: Pull complete
85ce9d0534d0: Pull complete
d3565df0a804: Pull complete
Digest: sha256:59a5854dca16488305aee60c8dea4d88b68d816aee62
7de022b19d9bead48d04
Status: Downloaded newer image for mysql/mysql-server:8.0.13
docker.io/mysql/mysql-server:8.0.13
```

The `docker pull` command retrieves an image from a Docker repository and is conceptually similar to the `git pull` command, which retrieves changes from a `git` repository.

This downloaded four image layers in total because this image is built on top of three other images. We'll see later how that works when we learn how to build a Dockerfile.

We can query which images are stored on our laptop with the following command:

```
$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
mysql/mysql-server  8.0      716286be47c6  8 days ago   381MB
hello-world         latest   bf756fb1ae65  4 months ago  13.3kB
```

There are two images currently available—the `mysql-server` image we just downloaded and the `hello-world` image we ran earlier.

We can remove unwanted images with the following command:

```
$ docker rmi hello-world
Untagged: hello-world:latest
Untagged: hello-world@sha256:8e3114318a995a1ee497790535e
    7b88365222a21771ae7e53687ad76563e8e76
Deleted: sha256:bf756fb1ae65adf866bd8c456593cd24beb6a0
    a061dedf42b26a993176745f6b
Deleted: sha256:9c27e219663c25e0f28493790cc0b88bc973ba
    3b1686355f221c38a36978ac63
```

Notice that the actual delete operation works with the SHA256 image identifier.

A container can be launched with the image, as follows:

```
$ docker run --name=mysql --env MYSQL_ROOT_PASSWORD=w0rdw0rd
mysql/mysql-server:8.0
[Entrypoint] MySQL Docker Image 8.0.13-1.1.8
[Entrypoint] Initializing database
2020-02-17T00:08:15.685715Z 0 [System] [MY-013169] [Server]
/usr/sbin/mysqld (mysqld 8.0.13) initializing of server in progress as
process 25
...
2020-02-17T00:08:44.490724Z 0 [System] [MY-013170] [Server]
/usr/sbin/mysqld (mysqld 8.0.13) initializing of server has completed
[Entrypoint] Database initialized
2020-02-17T00:08:48.625254Z 0 [System] [MY-010116] [Server]
/usr/sbin/mysqld (mysqld 8.0.13) starting as process 76
...
```

```
[Entrypoint] MySQL init process done. Ready for start up.  
  
[Entrypoint] Starting MySQL 8.0.13-1.1.8  
2020-02-17T00:09:14.611614Z 0 [System] [MY-010116] [Server]  
/usr/sbin/mysqld (mysqld 8.0.13) starting as process 1  
...
```

The `docker run` command takes an image name, along with various arguments, and launches it as a running container.

We started this service in the foreground, and there is a tremendous amount of output as MySQL initializes its container. Because of the `--name` option, the container name is `mysql`. Using an environment variable, we tell the container to initialize the `root` password.

Since we have a running server, let's use the MySQL CLI to make sure it's actually running. In another window, we can run the MySQL client inside the container, as follows:

```
$ docker exec -it mysql mysql -u root -p  
Enter password:  
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 14  
Server version: 8.0.13 MySQL Community Server - GPL  
...  
Type 'help;' or '\h' for help. Type '\c' to clear the current input  
statement.  
  
mysql> show databases;  
+-----+  
| Database |  
+-----+  
| information_schema |  
| mysql |  
| performance_schema |  
| sys |  
+-----+  
4 rows in set (0.02 sec)  
  
mysql>
```

The `docker exec` command lets you run programs inside the container. The `-it` option says the command is run interactively on an assigned terminal. In this case, we used the `mysql` command to run the MySQL client so that we could interact with the database. Substitute `bash` for `mysql`, and you will land in an interactive bash command shell.

This `mysql` command instance is running inside the container. The container is configured by default to not expose any external ports, and it has a default `my.cnf` file.

Docker containers are meant to be ephemeral, created and destroyed as needed, while databases are meant to be permanent, with lifetimes sometimes measured in decades. A very important discussion on this point and how it applies to database containers is presented in the next section.

It is cool that we can easily install and launch a MySQL instance. However, there are several considerations to be made:

- Access to the database from other software, specifically from another container
- Storing the database files outside the container for a longer lifespan
- Custom configuration, because database admins love to tweak the settings
- We need a path to connect the MySQL container to the AuthNet network that we'll be creating

Before proceeding, let's clean up. In a terminal window, type the following:

```
$ docker stop mysql
mysql
$ docker rm mysql
mysql
```

This closes out and cleans up the container we created. To reiterate the point made earlier, the database in that container went away. If that database contained critical information, you just lost it, with no chance of recovering the data.

Before moving on, let's discuss how this impacts the design of our services.

The ephemeral nature of Docker containers

Docker containers are designed to be easy to create and easy to destroy. In the course of kicking the tires, we've already created and destroyed three containers.

In the olden days (a few years ago), setting up a database required the provisioning of specially configured hardware, hiring a database admin with special skills, and carefully optimizing everything for the expected workload. In the space of a few paragraphs, we just instantiated and destroyed three database instances. What a brave new world this is!

In terms of databases and Docker containers, the database is relatively eternal, and the Docker container is ephemeral. Databases are expected to last for years, or perhaps even decades. In computer years, that's practically immortal. By contrast, a Docker container that is used and then immediately thrown away is merely a brief flicker of time compared to the expected lifetime of a database.

Those containers can be created and destroyed quickly, and this gives us a lot of flexibility. For example, orchestration systems, such as Kubernetes or AWS ECS, can automatically increase or decrease the number of containers to match traffic volume, restart containers that crash, and more.

But where does the data in a database container live? With the commands we ran in the previous section, the database data directory lives inside the container. When the container was destroyed, the data directory was destroyed, and any data in our database was vaporized. Obviously, this is not compatible with the life cycle requirements of the data we store in a database.

Fortunately, Docker allows us to attach a variety of mass storage services to a Docker container. The container itself might be ephemeral, but we can attach eternal data to the ephemeral container. It's just a matter of configuring the database container so that the data directory is on the correct storage system.

Enough theory, let's now do something. Specifically, let's create the infrastructure for the authentication service.

Defining the Docker architecture for the authentication service

Docker supports the creation of virtual bridge networks between containers. Remember that a Docker container has many of the features of an installed Linux OS. Each container can have its own IP address and exposed ports. Docker supports the creation of what amounts to a virtual Ethernet segment, called a **bridge network**. These networks live solely within the host computer and, by default, are not reachable by anything outside the host computer.

A Docker bridge network, therefore, has strictly limited access. Any Docker containers attached to a bridge network can communicate with other containers attached to that network and, by default, that network does not allow external traffic. The containers find each other by hostname, and Docker includes an embedded DNS server to set up the hostnames required. That DNS server is configured to not require dots in domain names, meaning the DNS/hostname of each container is simply the container name. We'll find later that the hostname of the container is actually `container-name.network-name`, and that the DNS configuration lets you skip using the `network-name` portion of the hostname. This policy of using hostnames to identify containers is Docker's implementation of service discovery.

Create a directory named `authnet` as a sibling to the `users` and `notes` directories. We'll be working on `authnet` in that directory.

In that directory, create a file—`package.json`—which we'll use solely to record commands for managing AuthNet:

```
{  
  "name": "authnet",  
  "version": "1.0.0",  
  "description": "Scripts to define and manage AuthNet",  
  "scripts": {  
    "build-authnet": "docker network create --driver bridge authnet"  
  },  
  "license": "ISC"  
}
```

We'll be adding more scripts to this file. The `build-authnet` command builds a virtual network using the `bridge` driver, as we just discussed. The name for this network is `authnet`.

Having created `authnet`, we can attach containers to it so that the containers can communicate with one another.

Our goal for the Notes application stack is to use private networking between containers to implement a security firewall around the containers. The containers will be able to communicate with one another, but the private network is not reachable by any other software and is, therefore, more or less safe from intrusion.

Type the following command:

```
$ npm run build-authnet  
  
> authnet@1.0.0 build-authnet /home/david/Chapter10/authnet  
> docker network create --driver bridge authnet  
  
876232c4f2268c5fb192702cd2a339036dc2e74fe777d863620dded498fc56d0  
$ docker network ls  
NETWORK ID      NAME      DRIVER SCOPE  
876232c4f226  authnet   bridge  local
```

This creates a Docker bridge network. The long coded string is the identifier for this network. The `docker network ls` command lists the existing networks in the current Docker system. In addition to the short hex ID, the network has the name we specified.

Look at details regarding the network with this command:

```
$ docker network inspect authnet  
... much JSON output
```

At the moment, this won't show any containers attached to `authnet`. The output shows the network name, the IP range of this network, the default gateway, and other useful network configuration information. Since nothing is connected to the network, let's get started with building the required containers:

```
$ docker network rm authnet  
authnet  
$ docker network ls  
NETWORK ID      NAME      DRIVER SCOPE
```

This command lets us remove a network from the Docker system. However, since we need this network, rerun the command to recreate it.

We have explored setting up a bridge network, and so our next step is to populate it with a database server.

Creating the MySQL container for the authentication service

Now that we have a network, we can start connecting containers to that network. In addition to attaching the MySQL container to a private network, we'll be able to control the username and password used with the database, and we'll also give it external storage. That will correct the issues we named earlier.

To create the container, we can run the following command:

```
$ docker run --name db-userauth \
  --env MYSQL_USER=userauth \
  --env MYSQL_PASSWORD=userauth \
  --env MYSQL_DATABASE=userauth \
  --mount type=bind,src=`pwd`/userauth-data,dst=/var/lib/mysql \
  --network authnet -p 3306:3306 \
  --env MYSQL_ROOT_PASSWORD=w0rdw0rd \
  mysql/mysql-server:8.0 \
  --bind_address=0.0.0.0 \
  --socket=/tmp/mysql.sock
```

This does several useful things all at once. It initializes an empty database configured with the named users and passwords, it mounts a host directory as the MySQL data directory, it attaches the new container to `authnet`, and it exposes the MySQL port to connections from outside the container.

The `docker run` command is only run the first time the container is started. It combines building the container by running it for the first time. With the MySQL container, its first run is when the database is initialized. The options that are passed to this `docker run` command are meant to tailor the database initialization.

The `--env` option sets environment variables inside the container. The scripts driving the MySQL container look to these environment variables to determine the user IDs, passwords, and database to create.

In this case, we configured a password for the `root` user, and we configured a second user—`userauth`—with a matching password and database name.

There are many more environment variables available.

The official MySQL Docker documentation provides more information on configuring a MySQL Docker container (<https://dev.mysql.com/doc/refman/8.0/en/docker-mysql-more-topics.html>).



The MySQL server recognizes an additional set of environment variables (<https://dev.mysql.com/doc/refman/8.0/en/environment-variables.html>).

The MySQL server recognizes a long list of configuration options that can be set on the command line or in the MySQL configuration file (<https://dev.mysql.com/doc/refman/8.0/en/server-option-variable-reference.html>).

The `--network` option attaches the container to the `authnet` network.

The `-p` option exposes a TCP port from inside the container so that it is visible outside the container. By default, containers do not expose any TCP ports. This means we can be very selective about what to expose, limiting the attack surface for any miscreants seeking to gain illicit access to the container.

The `--mount` option is meant to replace the older `--volume` option. It is a powerful tool for attaching external data storage to a container. In this case, we are attaching a host directory, `userauth-data`, to the `/var/lib/mysql` directory inside the container. This ensures that the database is not inside the container, and that it will last beyond the lifetime of the container. For example, while creating this example, we deleted this container several times to fine-tune the command line, and it kept using the same data directory.

We should also mention that the `--mount` option requires the `src=` option be a full pathname to the file or directory that is mounted. We are using ``pwd`` to determine the full path to the file. However, this is, of course, specific to Unix-like OSes. If you are on Windows, the command should be run in PowerShell and you can use the `$PSScriptRoot` variable. Alternatively, you can hardcode an absolute pathname.

It is possible to inject a custom `my.cnf` file into the container by adding this option to the `docker run` command:

```
--mount type=bind,src=`pwd`/my.cnf,dst=/etc/my.cnf
```

In other words, Docker lets you mount not only a directory but also a single file.

The command line follows this pattern:

```
$ docker run \
  docker run options \
  mysql/mysql-server:8.0 \
  mysqld options
```

So far, we have talked about the options for the `docker run` command. Those options configure the characteristics of the container. Next on the command line is the image name—in this case, `mysql/mysql-server:8.0`. Any command-line tokens appearing after the image name are passed into the container. In this case, they are interpreted as arguments to the MySQL server, meaning we can configure this server using any of the extensive sets of command-line options it supports. While we can mount a `my.cnf` file in the container, it is possible to achieve most configuration settings this way.

The first of these options, `--bind_address`, tells the server to listen for connections from any IP address.

The second, `--socket=/tmp/mysql.sock`, serves two purposes. One is security, to ensure that the MySQL Unix domain socket is accessible only from inside the container. By default, the scripts inside the MySQL container put this socket in the `/var/lib/mysql` directory, and when we attach the data directory, the socket is suddenly visible from outside the container.

On Windows, if this socket is in `/var/lib/mysql`, when we attach a data directory to the container, that would put the socket in a Windows directory. Since Windows does not support Unix domain sockets, the MySQL container will mysteriously fail to start and give a misleadingly obtuse error message. The `--socket` option ensures that the socket is instead on a filesystem that supports Unix domain sockets, avoiding the possibility of this failure.

When experimenting with different options, it is important to delete the mounted data directory each time you recreate the container to try a new setting. If the MySQL container sees a populated data directory, it skips over most of the container initialization scripts and will not run. A common mistake when trying different container MySQL configuration options is to rerun `docker run` without deleting the data directory. Since the MySQL initialization doesn't run, nothing will have changed and it won't be clear why the behavior isn't changing.

Therefore, to try a different set of MySQL options, execute the following command:

```
$ rm -rf userauth-data  
$ mkdir userauth-data  
$ docker run ... options ... mysql/mysql-server:8.0 ...
```

This will ensure that you are starting with a fresh database each time, as well as ensuring that the container initialization runs.

This also suggests an administrative pattern to follow. Any time you wish to update to a later MySQL release, simply stop the container, leaving the data directory in place. Then, delete the container and re-execute the `docker run` command with a new `mysql/mysql-server` tag. That will cause Docker to recreate the container using a different image, but using the same data directory. Using this technique, you can update the MySQL version by pulling down a newer image.

Once you have the MySQL container running, type this command:

```
MacBook-Pro-4:users davids docker ps -a  
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS                         NAMES  
7c3836505133        mysql/mysql-server:8.0.13   "/entrypoint.sh mys..."   19 seconds ago    Up 17 seconds (health: starting)   0.0.0.0:3306->3306/tcp, 33060/tcp   db-userauth
```

This will show the current container status. If we use `docker ps -a`, we see that the PORTS column says `0.0.0.0:3306->3306/tcp, 33060/tcp`. That says that the container is listening to access from anywhere (`0.0.0.0`) to port 3306, and this traffic will connect to port 3306 inside the container. Additionally, there is a port 33060 that is available, but it is not exposed outside the container.

Even though it is configured to listen to the whole world, the container is attached to authnet, which limits where connections can come from. Limiting the scope of processes that can attach to the database is a good thing. However, since we used the `-p` option, the database port is exposed to the host, and it's not as secure as we want. We'll fix this later.

Security in the database container

A question to ask is whether setting the `root` password like this is a good idea. The `root` user has broad access to the entire MySQL server, where other users, such as `userauth`, have limited access to the given database. Since one of our goals is security, we must consider whether this has created a secure or insecure database container.

We can log in as the `root` user with the following command:

```
$ docker exec -it db-userauth mysql -u root -p
Enter password:
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 115
Server version: 8.0.19 MySQL Community Server - GPL

...
```

This executes the MySQL CLI client inside the newly created container. There are a few commands we can run to check the status of the `root` and `userauth` user IDs. These include the following:

```
mysql> use mysql;
...
mysql> select host,user from user;
+-----+-----+
| host | user |
+-----+-----+
| %   | userauth |
| localhost | healthchecker |
| localhost | mysql.infoschema |
| localhost | mysql.session |
| localhost | mysql.sys |
| localhost | root |
+-----+-----+
6 rows in set (0.00 sec)
```

A connection to a MySQL server includes a user ID, a password, and the source of the connection. This connection might come from inside the same computer, or it might come over a TCP/IP socket from another computer. To approve the connection, the server looks in the `mysql.user` table for a row matching the `user`, `host` (source of connection), and `password` fields. The username and password are matched as a simple string comparison, but the host value is a more complex comparison. Local connections to the MySQL server are matched against rows where the host value is `localhost`.

For remote connections, MySQL compares the IP address and domain name of the connection against entries in the host column. The host column can contain IP addresses, hostnames, or wildcard patterns. The wildcard character for SQL is %. A single % character matches against any connection source, while a pattern of 172.% matches any IP address where the first IPv4 octet is 172, or 172.20.% matches any IP address in the 172.20.x.x range.

Therefore, since the only row for userauth specifies a host value of %, we can use userauth from anywhere. By contrast, the root user can only be used with a localhost connection.

The next task is to examine the access rights for the userauth and root user IDs:

```
mysql> show grants for userauth@'%';
+-----+
| Grants for userauth@% |
+-----+
| GRANT USAGE ON *.* TO `userauth`@`%` |
| GRANT ALL PRIVILEGES ON `userauth`.* TO `userauth`@`%` |
+-----+
2 rows in set (0.01 sec)

mysql> show grants for root@localhost;
... too wide
```

This says that the userauth user has full access to the userauth database. The root user, on the other hand, has full access to every database and has so many permissions that the output of that does not fit here. Fortunately, the root user is only allowed to connect from localhost.

To verify this, try connecting from different locations using these commands:

```
$ docker exec -it db-userauth mysql -u userauth -p
Enter password:
...
Server version: 8.0.19 MySQL Community Server - GPL
...
$ docker run -it --rm --network authnet mysql/mysql-server:8.0 mysql -
u userauth -h db-userauth -p
Enter password:
...
Server version: 8.0.19 MySQL Community Server - GPL
...
$ docker run -it --rm --network authnet mysql/mysql-server:8.0 mysql -
u root -h db-userauth -p
Enter password:
```

```
ERROR 1045 (28000): Access denied for user 'root'@'172.20.0.4' (using password: YES)
```

We've demonstrated four modes of accessing the database, showing that indeed, the `userauth` ID can be accessed either from the same container or from a remote container, while the `root` ID can only be used from the local container.

Using `docker run --it --rm ... container-name ...` starts a container, runs the command associated with the container, and then exits the container and automatically deletes it when it's done.

Therefore, with those last two commands, we created a separate `mysql/mysql-server:8.0` container, connected to `authnet`, to run the `mysql` CLI program.

The `mysql` arguments are to connect using the given username (`root` or `userauth`) to the MySQL server on the host named `db-userauth`. This demonstrates connecting to the database from a separate connector and shows that we can connect remotely with the `userauth` user, but not with the `root` user.

Then, the final access experiment involves leaving off the `--network` option:

```
$ docker run -it --rm mysql/mysql-server:8.0 mysql -u userauth -h db-userauth -p
[Entrypoint] MySQL Docker Image 8.0.19-1.1.15
Enter password:
ERROR 2005 (HY000): Unknown MySQL server host 'db-userauth' (0)
```

This demonstrates that if the container is not attached to `authnet`, it cannot access the MySQL server because the `db-userauth` hostname is not even known.

Where did the `db-userauth` hostname come from? We can find out by inspecting a few things:

```
$ docker network inspect authnet
...
{
    "Config": [
        { "Subnet": "172.20.0.0/16",
          "Gateway": "172.20.0.1" }
    ],
    ...
    "Containers": {
        "7c3836505133fc145743cd74b7220be72fd53ddd408227e961392e881d3b81b8": {
            ...
            "Name": "db-userauth",
            "EndpointID": "6005381b72caed482c699a3b00cf2e0019c
e4edd666b45e35be2afc6192314e4",
            "MacAddress": "02:42:ac:14:00:02",
            ...
        }
    }
}
```

```
"IPv4Address": "172.20.0.2/16",
"IPv6Address": ""
},
...
}
```

In other words, the authnet network has the 172.20.0.0/16 network number, and the db-userauth container was assigned the 172.20.0.2 IP address. This level of detail is rarely important, but it is useful on the first occasion to carefully examine the setup so that we understand what we're dealing with.

There is a gaping security issue that violates our design. Namely, the database port is visible to the host, and therefore, anyone with access to the host can access the database. This happened because we used `-p 3306:3306` in a misguided belief that this was required so that `svc-userauth`, which we'll build in the next section, can access the database. We'll fix this later by removing that option.

Now that we have the database instance set up for the authentication service, let's see how to Dockerize it.

Dockerizing the authentication service

The word *Dockerize* means to create a Docker image for a piece of software. The Docker image can then be shared with others or be deployed to a server. In our case, the goal is to create a Docker image for the user authentication service. It must be attached to authnet so that it can access the database server we just configured in the db-userauth container.

We'll name this new container `svc-userauth` to indicate that this is the user authentication REST service, while the db-userauth container is the database.

Docker images are defined using Dockerfiles, which are files to describe the installation of an application on a server. They document the setup of the Linux OS, installed software, and configuration required in the Docker image. This is literally a file named `Dockerfile`, containing Dockerfile commands. Dockerfile commands are used to describe how the image is constructed.



Refer to <https://docs.docker.com/engine/reference/builder/> for the documentation.

Creating the authentication service Dockerfile

In the `users` directory, create a file named `Dockerfile` containing the following content:

```
FROM node:14

RUN apt-get update -y \
&& apt-get upgrade -y \
&& apt-get -y install curl python build-essential git ca
-certificates

ENV DEBUG="users:__":
ENV PORT="5858"
ENV SEQUELIZE_CONNECT="sequelize-docker-mysql.yaml"
ENV REST_LISTEN="0.0.0.0"

RUN mkdir -p /userauth
COPY package.json *.yaml *.mjs /userauth/
WORKDIR /userauth
RUN npm install --unsafe-perm

EXPOSE 5858
CMD [ "node", "./user-server.mjs" ]
```

The `FROM` command specifies a pre-existing image, called the base image, from which to derive a given image. Frequently, you define a Docker image by starting from an existing image. In this case, we're using the official Node.js Docker image (https://hub.docker.com/_/node/), which, in turn, is derived from `debian`.

Because the base image, `node`, is derived from the `debian` image, the commands available are what are provided on a Debian OS. Therefore, we use `apt-get` to install more packages.

The `RUN` commands are where we run the shell commands required to build the container. The first one installs required Debian packages, such as the `build-essential` package, which brings in compilers required to install native-code Node.js packages.

It's recommended that you always combine `apt-get update`, `apt-get upgrade`, and `apt-get install` in the same command line like this because of the Docker build cache. Docker saves each step of the build to avoid rerunning steps unnecessarily. When rebuilding an image, Docker starts with the first changed step. Therefore, in the set of Debian packages to install changes, we want all three of those commands to run.

Combining them into a single command ensures that this will occur. For a complete discussion, refer to the documentation at https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

The ENV commands define environment variables. In this case, we're using the same environment variables that were defined in the package.json script for launching the user authentication service.

Next, we have a sequence of lines to create the /userauth directory and to populate it with the source code of the user authentication service. The first line creates the /userauth directory. The COPY command, as its name implies, copies the files for the authentication service into that directory. The WORKDIR command changes the working directory to /userauth. This means that the last RUN command, npm install, is executed in /userauth, and therefore, it installs the packages described in /userauth/package.json in /userauth/node_modules.

There is a new SEQUELIZE_CONNECT configuration file mentioned: sequelize-docker-mysql.yaml. This will describe the Sequelize configuration required to connect to the database in the db-userauth container.

Create a new file named users/sequelize-docker-mysql.yaml containing the following:

```
dbname: userauth
username: userauth
password: userauth
params:
  host: db-userauth
  port: 3306
  dialect: mysql
```

The difference is that instead of localhost as the database host, we use db-userauth. Earlier, we explored the db-userauth container and determined that this was the hostname of the container. By using db-userauth in this file, the authentication service will use the database in the container.

The EXPOSE command informs Docker that the container listens on the named TCP port. This does not expose the port beyond the container. The -p flag is what exposes a given port outside the container.

Finally, the CMD command documents the process to launch when the container is executed. The RUN commands are executed while building the container, while CMD says what's executed when the container starts.

We could have installed PM2 in the container, and then used a PM2 command to launch the service. However, Docker is able to fulfill the same function because it automatically supports restarting a container if the service process dies.

Building and running the authentication service Docker container

Now that we've defined the image in a Dockerfile, let's build it.

In `users/package.json`, add the following line to the `scripts` section:

```
"docker-build": "docker build -t svc-userauth ."
```

As has been our habit, this is an administrative task that we can record in `package.json`, making it easier to automate this task.

We can build the authentication service as follows:

```
$ npm run docker-build

> user-auth-server@1.0.0 docker-build /home/david/Chapter10/users
> docker build -t svc-userauth .

Sending build context to Docker daemon 32.03MB
Step 1/12 : FROM node:14
--> 07e774543bdf
Step 2/12 : RUN apt-get update -y && apt-get upgrade -y && apt-get -y
install curl python build-essential git ca-certificates
--> Using cache
--> eb28eaeee8517
Step 3/12 : ENV DEBUG="users:/*"
--> Using cache
--> 99ae7f4bde83
Step 4/12 : ENV PORT="5858"
--> Using cache
--> e7f7567a0ce4
... more output
```

The `docker build` command builds an image from a Dockerfile. Notice that the build executes one step at a time, and that the steps correspond exactly to the commands in the Dockerfile.

Each step is stored in a cache so that it doesn't have to be rerun. On subsequent builds, the only steps executed are the step that changed and all subsequent steps.

In authnet/package.json, we require quite a few scripts to manage the user authentication service:

```
{  
  "name": "authnet",  
  "version": "1.0.0",  
  "description": "Scripts to define and manage AuthNet",  
  "scripts": {  
    "build-authnet": "docker network create --driver bridge authnet",  
    "prebuild-db-userauth": "mkdir userauth-data",  
    "build-db-userauth": "docker run --detach --name db-userauth --env  
      MYSQL_USER=userauth --env MYSQL_PASSWORD=userauth --env  
      MYSQL_DATABASE=userauth --mount type=bind,src=`pwd`/userauth-  
      data,dst=/var/lib/mysql --network authnet --env  
      MYSQL_ROOT_PASSWORD=w0rdw0rd --env DATABASE_HOST=  
        db-userauth mysql/mysql-server:8.0 --  
        bind_address=0.0.0.0 --socket=/tmp/mysql.sock",  
    "stop-db-userauth": "docker stop db-userauth",  
    "start-db-userauth": "docker start db-userauth",  
    "build-userauth": "cd ../users && npm run docker-build",  
    "postbuild-userauth": "docker run --detach --name svc-userauth  
      --network authnet svc-userauth",  
    "start-userauth": "docker start svc-userauth",  
    "stop-userauth": "docker stop svc-userauth",  
    "start-user-service": "npm run start-db-userauth && npm run start  
      -userauth",  
    "stop-user-service": "npm run stop-db-userauth && npm run stop  
      -userauth"  
  },  
  "license": "ISC"  
}
```

This is the set of commands that were found to be useful to manage building the images, starting the containers, and stopping the containers.

Look carefully and you will see that we've added `--detach` to the `docker run` commands. So far, we've used `docker run` without that option, and the container remained in the foreground. While this was useful to see the logging output, it's not so useful for deployment. With the `--detach` option, the container becomes a background task.

On Windows, for the `--mount` option, we need to change the `src=` parameter (as discussed earlier) to use a Windows-style hard-coded path. That means it should read:

```
-mount type=bind,src=C:/Users/path/to/Chapter11/authnet/userauth-
data,dst=/var/lib/mysql
```

This option requires absolute pathnames and specifying the path this way works on Windows.

Another thing to notice is the absence of the `-p 3306:3306` option. It was determined that this was not necessary for two reasons. First, the option exposed the database in `db-userauth` to the host, when our security model required otherwise, and so removing the option got us the desired security. Second, `svc-userauth` was still able to access the `db-userauth` database after this option was removed.

With these commands, we can now type the following to build and then run the containers:

```
$ npm run build-authnet
$ npm run build-db-userauth
$ npm run build-userauth
```

These commands build the pieces required for the user authentication service. As a side effect, the containers are automatically executed and will launch as background tasks.

Once it is running, you can test it using the `cli.mjs` script as before. You can shell into the `svc-userauth` container and run `cli.mjs` there; or, since the port is visible to the host computer, you can run it from outside the container.

Afterward, we can manage the whole service as follows:

```
$ npm run stop-user-service
$ npm run start-user-service
```

This stops and starts both containers making up the user authentication service.

We have created the infrastructure to host the user authentication service, plus a collection of scripts to manage the service. Our next step is to explore what we've created and learn a few things about the infrastructure Docker creates for us.

Exploring AuthNet

Remember that AuthNet is the connection medium for the authentication service. To understand whether this network provides the security gains we're looking for, let's explore what we just created:

```
$ docker network inspect authnet
```

This prints out a large JSON object describing the network, along with its attached containers, which we've looked at before. If everything went well, we will see that there are now two containers attached to `authnet` where there'd previously have just been one.

Let's go into the `svc-userauth` container and poke around:

```
$ docker exec -it svc-userauth bash
root@ba75699519ef:/userauth# ls
cli.mjs          package-lock.json sequelize-docker-mysql.yaml
users-sequelize.mjs node_modules      package.json
user-server.mjs
```

The `/userauth` directory is inside the container and contains the files placed in the container using the `COPY` command, plus the installed files in `node_modules`:

```
root@ba75699519ef:/userauth# node cli.mjs list-users
[ {
  id: 'me', username: 'me', provider: 'local',
  familyName: 'Einarsdottir',
  givenName: 'Ashildr', middleName: null,
  emails: [ 'me@stolen.tardis' ],
  photos: []
}, {
  id: 'snuffy-smith', username: 'snuffy-smith', provider: 'local',
  familyName: 'Smith', givenName: 'John',
  middleName: 'Snuffy',
  emails: [ 'snuffy@example.com' ],
  photos: []
} ]
```

We can run the `cli.mjs` script to test and administer the service. To get these database entries set up, use the `add` command with the appropriate options:

```
root@4996275c4030:/userauth# ps -eafw
UID  PID  PPID C STIME TTY TIME CMD
root   1     0 2 00:08 ? 00:00:01 node ./user-server.mjs
root   19    19 0 00:09 pts/0 00:00:00 bash
root   27    19 0 00:09 pts/0 00:00:00 ps -eafw
```

```
root@ba75699519ef:/userauth# ping db-userauth
PING db-userauth (172.20.0.3) 56(84) bytes of data.
64 bytes from db-userauth.authnet (172.20.0.3): icmp_seq=1 ttl=64
time=0.163 ms
^C
--- db-userauth ping statistics ---
1 packet transmitted, 1 received, 0% packet loss, time 1003ms
root@ba75699519ef:/userauth# ping svc-userauth
PING svc-userauth (172.20.0.2) 56(84) bytes of data.
64 bytes from ba75699519ef (172.20.0.2): icmp_seq=1 ttl=64 time=0.073
ms
^C

--- svc-userauth ping statistics ---
1 packet transmitted, 1 received, 0% packet loss, time 2051ms
```

The process listing is interesting to study. Process PID 1 is the node ./user-server.mjs command in the Dockerfile. The format we used for the CMD line ensured that the node process ended up as process 1. This is important so that process signals are handled correctly, allowing Docker to manage the service process correctly. The tail end of the following blog post has a good discussion of the issue:

<https://www.docker.com/blog/keep-nodejs-rockin-in-docker/>

A ping command proves that the two containers are available as hostnames matching the container names:

```
$ ping 172.20.0.2
PING 172.20.0.2 (172.20.0.2): 56 data bytes
Request timeout for icmp_seq 0
^C
--- 172.20.0.2 ping statistics ---
2 packets transmitted, 0 packets received, 100.0% packet loss
$ ping 172.20.0.3
PING 172.20.0.3 (172.20.0.3): 56 data bytes
Request timeout for icmp_seq 0
^C
--- 172.20.0.3 ping statistics ---
2 packets transmitted, 0 packets received, 100.0% packet loss
```

From outside the containers, on the host system, we cannot ping the containers. That's because they are attached to authnet and are not reachable.

We have successfully Dockerized the user authentication service in two containers—`db-userauth` and `svc-userauth`. We've poked around the insides of a running container and found some interesting things. However, our users need the fantastic Notes application to be running, and we can't afford to rest on our laurels.

Since this was our first time setting up a Docker service, we went through a lot of details. We started by launching a MySQL database container, and what is required to ensure that the data directory is persistent. We then set up a Dockerfile for the authentication service and learned how to connect containers to a common Docker network and how containers can communicate with each other over the network. We also studied the security benefits of this network infrastructure, since we can easily wall off the service and its database from intrusion.

Let's now move on and Dockerize the Notes application, making sure that it is connected to the authentication server.

Creating FrontNet for the Notes application

We have the back half of our system set up in Docker containers, as well as the private bridge network to connect the backend containers. It's now time to do the same for the front half of the system: the Notes application (`svc-notes`) and its associated database (`db-notes`). Fortunately, the tasks required to build FrontNet are more or less the same as what we did for AuthNet.

The first task is to set up another private bridge network, `frontnet`. Like `authnet`, this will be the infrastructure for the front half of the Notes application stack.

Create a directory, `frontnet`, and in that directory, create a `package.json` file that will contain the scripts to manage `frontnet`:

```
{
  "name": "frontnet",
  "version": "1.0.0",
  "description": "Scripts to define and manage FrontNet",
  "scripts": {
    "build-frontnet": "docker network create --driver bridge frontnet",
  },
  "license": "ISC"
}
```

As with authnet, this is just the starting point as we have several more scripts to add.

Let's go ahead and create the `frontnet` bridge network:

```
$ npm run build-frontnet
...
$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
3021e2069278   authnet    bridge      local
f3df227d4bff   frontnet   bridge      local
```

We have two virtual bridge networks. Over the next few sections, we'll set up the database and Notes application containers, connect them to `frontnet`, and then see how to manage everything.

MySQL container for the Notes application

As with authnet, the task is to construct a MySQL server container using the `mysql/mysql-server` image. We must configure the server to be compatible with the `SEQUELIZE_CONNECT` file that we'll use in the `svc-notes` container. For that purpose, we'll use a database named `notes` and a `notes` user ID.

For that purpose, add the following to the `scripts` section of the `package.json` file:

```
"prebuild-db-notes": "mkdir notes-data",
"build-db-notes": "docker run --detach --name db-notes --env
  MYSQL_USER=notes --env MYSQL_PASSWORD=notes12345 --env
  MYSQL_DATABASE=notes --mount type=bind,src=`pwd`/notes-
  data,dst=/var/lib/mysql --network frontnet --env
  MYSQL_ROOT_PASSWORD=w0rdw0rd mysql/mysql-server:8.0 --
  bind_address=0.0.0.0 --socket=/tmp/mysql.sock",
"stop-db-notes": "docker stop db-notes",
"start-db-notes": "docker start db-notes",
```

This is largely the same as for `db-userauth`, with the word `notes` substituted for `userauth`. Remember that on Windows the `--mount` option requires a Windows-style absolute pathname.

Let's now run the script:

```
$ npm run build-db-notes
> frontnet@1.0.0 prebuild-db-notes /home/david/Chapter10/frontnet
> mkdir -p notes-data
```

```
> frontnet@1.0.0 build-db-notes /home/david/Chapter10/frontnet
> docker run --detach --name db-notes --env MYSQL_USER=notes --env
  MYSQL_PASSWORD=notes12345 --env MYSQL_DATABASE=notes --mount
  type=bind,src=`pwd`/notes-data,dst=/var/lib/mysql --network frontnet -
  p 3306:3306 --env MYSQL_ROOT_PASSWORD=w0rdw0rd mysql/mysql-server:8.0
  --bind_address=0.0.0.0

af60afab6994095fcabc11c86159bdb0b02924d3ad8bf08506f4c16171959bc2b
```

This database will be available in the `db-notes` domain name on `frontnet`. Because it's attached to `frontnet`, it won't be reachable by containers connected to `authnet`. To verify this, run the following command:

```
$ docker exec -it svc-userauth bash
root@ba75699519ef:/userauth# ping db-notes
ping: db-notes: Name or service not known
root@ba75699519ef:/userauth# ping db-userauth
PING db-userauth (172.20.0.3) 56(84) bytes of data.
64 bytes from db-userauth.authnet (172.20.0.3): icmp_seq=1 ttl=64
time=10.5 ms
...
root@ba75699519ef:/userauth# ping db-notes.frontnet
ping: db-notes.frontnet: Name or service not known
```

Since `db-notes` is on a different network segment, we've achieved separation. But we can notice something interesting. The `ping` command tells us that the full domain name for `db-userauth` is `db-userauth.authnet`. Therefore, it stands to reason that `db-notes` is also known as `db-notes.frontnet`. But either way, we cannot reach containers on `frontnet` from a container on `authnet`, and so we have achieved the desired separation.

We're able to move more quickly to construct FrontNet because it's so much like AuthNet. We just have to do what we did before and tweak the names.

In this section, we created a database container. In the next section, we will create the Dockerfile for the Notes application.

Dockerizing the Notes application

Our next step is, of course, to Dockerize the Notes application. This starts by creating a Dockerfile, and then adding another Sequelize configuration file, before finishing up by adding more scripts to the `frontnet/package.json` file.

In the notes directory, create a file named Dockerfile containing the following:

```
FROM node:14

RUN apt-get update -y \
    && apt-get -y install curl python build-essential git ca
    -certificates

ENV DEBUG="notes:*,messages:*
ENV SEQUELIZE_CONNECT="models/sequelize-docker-mysql.yaml"
ENV NOTES_MODEL="sequelize"
ENV USER_SERVICE_URL="http://svc-userauth:5858"
ENV PORT="3000"

RUN mkdir -p /notesapp /notesapp/minty /notesapp/partials
/notesapp/public /notesapp/routes /notesapp/theme /notesapp/theme/dist
/notesapp/views
COPY minty/ /notesapp/minty/
COPY models/*.mjs models/*.yaml /notesapp/models/
COPY partials/ /notesapp/partials/
COPY public/ /notesapp/public/
COPY routes/ /notesapp/routes/
COPY theme/dist/ /notesapp/theme/dist/
COPY views/ /notesapp/views/
COPY *.mjs package.json /notesapp/

WORKDIR /notesapp

RUN npm install --unsafe-perm

VOLUME /sessions
EXPOSE 3000
CMD [ "node", "./app.mjs" ]
```

This is similar to the Dockerfile we used for the authentication service. We're using the environment variables from notes/package.json, plus a new one: NOTES_SESSION_DIR.

The most obvious change is the number of COPY commands. The Notes application is a lot more involved, given the number of sub-directories full of files that must be installed. We start by creating the top-level directories of the Notes application deployment tree. Then, one by one, we copy each sub-directory into its corresponding sub-directory in the container filesystem.

In a `COPY` command, the trailing slash on the destination directory is important. Why? Because the Docker documentation says that the trailing slash is important, that's why.

The big question is *why use multiple COPY commands like this?* This would have been incredibly simple:

```
COPY . /notesapp
```

However, the multiple `COPY` commands let us control exactly what's copied. It's most important to avoid copying the `node_modules` directory into the container. Not only is the `node_modules` file on the host large, which would bloat the container if copied, but it is set up for the host OS and not the container OS. The `node_modules` directory must be built inside the container, with the installation happening on the container's OS. That constraint led to the choice to explicitly copy specific files to the destination.

We also have a new `SEQUELIZE_CONNECT` file. Create `models/sequelize-docker-mysql.yaml` containing the following:

```
dbname: notes
username: notes
password: notes12345
params:
  host: db-notes
  port: 3306
  dialect: mysql
```

This will access a database server on the `db-notes` domain name using the named database, username, and password.

Notice that the `USER_SERVICE_URL` variable no longer accesses the authentication service at `localhost`, but at `svc-userauth`. The `svc-userauth` domain name is currently only advertised by the DNS server on AuthNet, but the Notes service is on FrontNet. Therefore, this will cause a failure for us when we get to running the Notes application, and we'll have to make some connections so that the `svc-userauth` container can be accessed from `svc-notes`.

In Chapter 8, *Authenticating Users with a Microservice*, we discussed the need to protect the API keys supplied by Twitter. We could copy the `.env` file to the Dockerfile, but this may not be the best choice, and so we've left it out of the Dockerfile.



Unfortunately, this does not protect the Twitter credentials to the level required. The `.env` file is available as plaintext inside the container. Docker has a feature, Docker Secrets, that can be used to securely store data of this sort. Unfortunately, it is only available when using Swarm mode, which we are not doing at this time; but we will use this feature in Chapter 12, *Deploying a Docker Swarm to AWS EC2 Using Terraform*.

The value of `TWITTER_CALLBACK_HOST` needs to reflect where Notes is deployed. Right now, it is still on your laptop, but if it is deployed to a server, this variable will require the IP address or domain name of the server.

In `notes/package.json`, add the following `scripts` entry:

```
"scripts": {  
    ...  
    "docker-build": "docker build -t svc-notes ."  
    ...  
}
```

As with the authentication server, this lets us build the container image for the Notes application service.

Then, in `frontnet/package.json`, add these scripts:

```
"build-notes": "cd ../notes && npm run docker-build",  
"postbuild-notes": "docker run --detach --name svc-notes --network  
    frontnet -p 80:3000 svc-notes",  
"start-notes": "docker start svc-notes",  
"stop-notes": "docker stop svc-notes",  
"start-notes-service": "npm run start-db-notes && npm run start  
    -notes",  
"stop-notes-service": "npm run stop-db-notes && npm run stop-notes"
```

Now, we can build the container image:

```
$ npm run build-notes  
  
> frontnet@1.0.0 build-notes /home/david/Chapter10/frontnet  
> cd ../notes && npm run docker-build  
  
> notes@0.0.0 docker-build /home/david/Chapter10/notes  
> docker build -t svc-notes .  
  
Sending build context to Docker daemon 223.9MB  
Step 1/22 : FROM node:13.8
```

```
----> 07e774543bdf
...
> frontnet@1.0.0 postbuild-notes /home/david/Chapter10/frontnet
> docker run --detach --name svc-notes --network frontnet -p 80:3000
svc-notes
01bfff4818aedeb082760c9d39087c08f2ba167d601413f1a745fdf305cdc3d
```

This creates the container image and then launches the container.

Notice that the exposed port 3000 is mapped with `-p 80:3000` onto the normal HTTP port. Since we're getting ready for deployment on a real service, we can stop using port 3000.

At this point, we can connect our browser to `http://localhost` and start using the Notes application. However, we'll quickly run into a problem:



The user experience team is going to scream about this ugly error message, so put it on your backlog to generate a prettier error screen. For example, a flock of birds pulling a whale out of the ocean is popular.

This error means that Notes cannot access anything at the host named `svc-userauth`. That host does exist because the container is running, but it's not on `frontnet`, and is not reachable from the `notes` container. Instead, it is on `authnet`, which is currently not reachable by `svc-notes`:

```
$ docker exec -it svc-notes bash
root@9318fa2ecbb6:/notesapp# ping svc-userauth
ping: svc-userauth: Name or service not known
root@9318fa2ecbb6:/notesapp# ping svc-userauth.authnet
ping: svc-userauth.authnet: Name or service not known

root@9318fa2ecbb6:/notesapp# ping db-notes
PING db-notes (172.21.0.2) 56(84) bytes of data.
64 bytes from db-notes.frontnet (172.21.0.2): icmp_seq=1 ttl=64
time=1.33 ms
```

We can reach `db-notes` from `svc-notes` but not `svc-userauth`. This is as expected since we have attached these containers to different networks.

If you inspect `FrontNet` and `AuthNet`, you'll see that the containers attached to each do not overlap:

```
$ docker network inspect frontnet
$ docker network inspect authnet
```

In the architecture diagram presented in Chapter 10, *Deploying Node.js Applications to Linux Servers*, we showed a connection between the `svc-notes` and `svc-userauth` containers. This connection is required so that Notes can authenticate its users. But that connection does not yet exist.

Docker requires you to take a second step to attach the container to a second network:

```
$ docker network connect authnet svc-notes
```

With no other change, the Notes application will now allow you to log in and start adding and editing notes. Furthermore, start a shell in `svc-notes` and you'll be able to ping both `svc-userauth` and `db-userauth`.

There is a glaring architecture question staring at us. Do we connect the `svc-userauth` service to `frontnet`, or do we connect the `svc-notes` service to `authnet`? We just connected `svc-notes` to `authnet`, but maybe that's not the best choice. To verify which network setup solves the problem, run the following commands:

```
$ docker network disconnect authnet svc-notes
$ docker network connect frontnet svc-userauth
```

The first time around, we connected `svc-notes` to `authnet`, then we disconnected it from `authnet`, and then connected `svc-userauth` to `frontnet`. That means we tried both combinations and, as expected, in both cases, `svc-notes` and `svc-userauth` were able to communicate.

This is a question for security experts since the consideration is the attack vectors available to any intruders. Suppose Notes has a security hole allowing an invader to gain access. How do we limit what is reachable via that hole?

The primary observation is that by connecting `svc-notes` to `authnet`, `svc-notes` not only has access to `svc-userauth` but also to `db-userauth`. To see this, run these commands:

```
$ docker network disconnect frontnet svc-userauth
$ docker network connect authnet svc-notes
$ docker exec -it svc-notes bash
root@9318fa2ecbb6:/notesapp#
root@9318fa2ecbb6:/notesapp# ping svc-userauth
PING svc-userauth (172.20.0.2) 56(84) bytes of data.
64 bytes from svc-userauth.authnet (172.20.0.2): icmp_seq=1 ttl=64
time=0.151 ms
...
root@9318fa2ecbb6:/notesapp# ping db-userauth
PING db-userauth (172.20.0.3) 56(84) bytes of data.
64 bytes from db-userauth.authnet (172.20.0.3): icmp_seq=1 ttl=64
time=0.379 ms
```

This sequence reconnects `svc-notes` to `authnet` and demonstrates the ability to access both the `svc-userauth` and `db-userauth` containers. Therefore, a successful invader could access the `db-userauth` database, a result we wanted to prevent. Our diagram in Chapter 10, *Deploying Node.js Applications to Linux Servers*, showed no such connection between `svc-notes` and `db-userauth`.

Given that our goal for using Docker was to limit the attack vectors, we have a clear distinction between the two container/network connection setups. Attaching `svc-userauth` to `frontnet` limits the number of containers that can access `db-userauth`. For an intruder to access the user information database, they must first break into `svc-notes`, and then break into `svc-userauth`; unless, that is, our amateur attempt at a security audit is flawed.

For this and a number of other reasons, we arrive at this final set of scripts for `frontnet/package.json`:

```
"build-frontnet": "docker network create --driver bridge frontnet",
"connect-userauth": "docker network connect frontnet svc-userauth",
"prebuild-db-notes": "mkdir -p notes-data",
"build-db-notes": "docker run --detach --name db-notes --env
  MYSQL_USER=notes --env MYSQL_PASSWORD=notes12345 --env
  MYSQL_DATABASE=notes --mount type=bind,src=`pwd`/notes-
  data,dst=/var/lib/mysql --network frontnet --env
  MYSQL_ROOT_PASSWORD=w0rdw0rd mysql/mysql-server:8.0
  --bind_address=0.0.0.0",
"stop-db-notes": "docker stop db-notes",
"start-db-notes": "docker start db-notes",
```

```
"build-notes": "cd ../notes && npm run docker-build",
"postbuild-notes": "npm run launch-notes",
"launch-notes": "docker run --detach --name svc-notes --network
  frontnet -p 80:3000 node-web-development/notes",
"start-notes": "docker start svc-notes",
"stop-notes": "docker stop svc-notes",
"start-notes-service": "npm run start-db-notes && npm run start
  -notes",
"stop-notes-service": "npm run stop-db-notes && npm run stop-notes"
```

Primarily, this adds a command, `connect-userauth`, to connect `svc-userauth` to `frontnet`. That helps us remember our decision on how to join the containers. We also took the opportunity to do a little reorganization.

We've learned a lot in this section about Docker—using Docker images, creating Docker containers from images, and configuring a group of Docker containers with some security constraints in mind. We came out of this section having implemented our initial architecture idea. We have two private networks with the containers connected to their appropriate network. The only exposed TCP port is the Notes application, visible on port 80. The other containers connect with one another using TCP/IP connections that are not available from outside the containers.

Before proceeding to the next section, you may want to shut down the services we've launched. Simply execute the following command:

```
$ cd frontnet
$ npm run stop-notes-service
$ cd ../authnet
$ npm run stop-user-service
```

Because we've automated many things, it is this simple to administer the system. However, it is not as automated as we want it to be. To address that, let's learn how to make the Notes stack more easily deployable by using Docker Compose to describe the infrastructure.

Managing multiple containers with Docker Compose

It is cool that we can create encapsulated instantiations of the software services that we've created. In theory, we can publish these images to Docker repositories, and then launch the containers on any server we want. For example, our task in [Chapter 10, Deploying Node.js Applications to Linux Servers](#), would be greatly simplified with Docker. We could simply install Docker Engine on the Linux host and then deploy our containers on that server, and not have to deal with all those scripts and the PM2 application.

But we haven't properly automated the process. The promise was to use the Dockerized application for deployment on cloud services. In other words, we need to take all this learning and apply it to the task of simplifying deployment.

We've demonstrated that, with Docker, Notes can be built using four containers that have a high degree of isolation from each other and from the outside world.

There is a glaring problem: our process in the previous section was partly manual, partly automated. We created scripts to launch each portion of the system, which is good practice. However, we did not automate the entire process to bring up Notes and the authentication services, nor is this solution scalable beyond one machine.

Let's start with the last issue first—scalability. Within the Docker ecosystem, several **Docker orchestrator** services are available. An orchestrator automatically deploys and manages Docker containers over a group of machines. Some examples of Docker orchestrators are Docker Swarm, Kubernetes, CoreOS Fleet, and Apache Mesos. These are powerful systems that can automatically increase/decrease resources as needed to move containers from one host to another, and more. We mention these systems for you to further study as your needs grow. In [Chapter 12, Deploying a Docker Swarm to AWS EC2 with Terraform](#), we will build on the work we're about to do in order to deploy Notes in a Docker Swarm cluster that we'll build on AWS EC2 infrastructure.

Docker Compose (<https://docs.docker.com/compose/overview/>) will solve the other problems we've identified. It lets us easily define and run several Docker containers together as a complete application. It uses a YAML file, `docker-compose.yml`, to describe the containers, their dependencies, the virtual networks, and the volumes. While we'll be using it to describe deployment on a single host machine, Docker Compose can be used for multi-machine deployments. Namely, Docker Swarm directly uses compose files to describe the services you launch in a swarm. In any case, learning about Docker Compose will give you a headstart on understanding the other systems.

Before proceeding, ensure that Docker Compose is installed. If you've installed Docker for Windows or Docker for Mac, everything that is required is installed. On Linux, you must install it separately by following the instructions in the links provided earlier.

Docker Compose file for the Notes stack

We just talked about Docker orchestration services, but Docker Compose is not itself such a service. Instead, Docker Compose uses a specific YAML file structure to describe how to deploy Docker containers. With a Docker Compose file, we can describe one or more containers, networks, and volumes involved in launching a Docker-based service.

Let's start by creating a directory, `compose-local`, as a sibling to the `users` and `notes` directories. In that directory, create a file named `docker-compose.yml`:

```
version: '3'
services:
  db-userauth:
    image: "mysql/mysql-server:8.0"
    container_name: db-userauth
    command: [ "mysqld",
               "--character-set-server=utf8mb4",
               "--collation-server=utf8mb4_unicode_ci",
               "--bind-address=0.0.0.0",
               "--socket=/tmp/mysql.sock" ]
    expose:
      - "3306"
    networks:
      - authnet
    volumes:
```

```
        - db-userauth-data:/var/lib/mysql
    restart: always
    environment:
        MYSQL_ROOT_PASSWORD: "w0rdw0rd"
        MYSQL_USER: userauth
        MYSQL_PASSWORD: userauth
        MYSQL_DATABASE: userauth

  svc-userauth:
    build: ../users
    container_name: svc-userauth
    depends_on:
        - db-userauth
    networks:
        - authnet
    # DO NOT EXPOSE THIS PORT ON PRODUCTION
    ports:
        - "5858:5858"
    restart: always

db-notes:
  image: "mysql/mysql-server:8.0"
  container_name: db-notes
  command: [ "mysqld",
             "--character-set-server=utf8mb4",
             "--collation-server=utf8mb4_unicode_ci",
             "--bind-address=0.0.0.0",
             "--socket=/tmp/mysql.sock" ]
  expose:
      - "3306"
  networks:
      - frontnet
  volumes:
      - db-notes-data:/var/lib/mysql
  restart: always
  environment:
      MYSQL_ROOT_PASSWORD: "w0rdw0rd"
      MYSQL_USER: notes
      MYSQL_PASSWORD: notes12345
      MYSQL_DATABASE: notes

  svc-notes:
    build: ../notes
    container_name: svc-notes
    depends_on:
        - db-notes
    networks:
        - frontnet
```

```
ports:
  - "3000:3000"
restart: always

networks:
  frontnet:
    driver: bridge
  authnet:
    driver: bridge

volumes:
  db-userauth-data:
  db-notes-data:
```

That's the description of the entire Notes deployment. It's at a fairly high level of abstraction, roughly equivalent to the options in the command-line tools we've used so far. It's fairly succinct and self-explanatory, and, as we'll see, the `docker-compose` command makes these files a convenient way to manage Docker services.

The `version` line says that this is a version 3 Compose file. The version number is inspected by the `docker-compose` command so that it can correctly interpret its content. The full documentation is worth reading at <https://docs.docker.com/compose/compose-file/>.

There are three major sections used here: `services`, `volumes`, and `networks`. The `services` section describes the containers being used, the `networks` section describes the networks, and the `volumes` section describes the volumes. The content of each section matches the containers we created earlier. The configuration we've already dealt with is all here, just rearranged.

There are the two database containers—`db-userauth` and `db-notes`—and the two service containers—`svc-userauth` and `svc-notes`. The service containers are built from a Dockerfile located in the directory named in the `build` attribute. The database containers are instantiated from images downloaded from Docker Hub. Both correspond directly to what we did previously, using the `docker run` command to create the database containers and using `docker build` to generate the images for the services.

The `container_name` attribute is equivalent to the `--name` attribute and specifies a user-friendly name for the container. We must specify the container name in order to specify the container hostname to effect a Docker-style service discovery.

The `networks` attribute lists the networks to which this container must be connected and is exactly equivalent to the `--net` argument. Even though the `docker` command doesn't support multiple `--net` options, we can list multiple networks in the Compose file. In this case, the networks are bridge networks. As we did earlier, the networks themselves must be created separately and, in a Compose file, this is done in the `networks` section.

The `ports` attribute declares the ports that are to be published and the mapping to container ports. In the `ports` declaration, we have two port numbers, the first being the published port number and the second being the port number inside the container. This is exactly equivalent to the `-p` option used earlier.

The `depends_on` attribute lets us control the start up order. A container that depends on another will wait to start until the depended-on container is running.

The `volumes` attribute describes mappings of a container directory to a host directory. In this case, we've defined two volume names—`db-userauth-data` and `db-notes-data`—and then used them for the volume mapping. However, when we deploy to Docker Swarm on AWS EC2, we'll need to change how this is implemented.

Notice that we haven't defined a host directory for the volumes. Docker will assign a directory for us, which we can learn about by using the `docker volume inspect` command.

The `restart` attribute controls what happens if or when the container dies. When a container starts, it runs the program named in the `CMD` instruction, and when that program exits, the container exits. But what if that program is meant to run *forever*; shouldn't Docker know that it should restart the process? We could use a background process supervisor, such as Supervisord or PM2. However, the Docker `restart` option takes care of it.

The `restart` attribute can take one of the following four values:

- `no`: Do not restart.
- `on-failure:count`: Restart up to N times.
- `always`: Always restart.
- `unless-stopped`: Start the container unless it was explicitly stopped.

In this section, we've learned how to build a Docker Compose file by creating one that describes the Notes application stack. With that in hand, let's see how to use this tool to launch the containers.

Building and running the Notes application with Docker Compose

With the Docker Compose CLI tool, we can manage any sets of Docker containers that can be described in a `docker-compose.yml` file. We can build the containers, bring them up and down, view the logs, and more. On Windows, we're able to run the commands in this section unchanged.

Our first task is to create a clean slate by running these commands:

```
$ docker stop db-notes svc-userauth db-auth svc-notes
db-notes
svc-userauth
db-auth
svc-notes
$ docker rm db-notes svc-userauth db-auth svc-notes
db-notes
svc-userauth
db-auth
svc-notes
```

We first needed to stop and delete any existing containers left over from our previous work. We can also use the scripts in the `frontnet` and `authnet` directories to do this. `docker-compose.yml` used the same container names, so we need the ability to launch new containers with those names.

To get started, use this command:

```
$ docker-compose build
db-userauth uses an image, skipping
db-notes uses an image, skipping
Building svc-userauth
Step 1/12 : FROM node:13.8
--> 07e774543bdf$ docker-compose up
...
Successfully built f714b877dbec
Successfully tagged compose-local_svc-userauth:latest
Building svc-notes
Step 1/26 : FROM node:13.8
--> 07e774543bdf
...
Successfully built 36b358e3dd0e
Successfully tagged compose-local_svc-notes:latest
```

This builds the images listed in `docker-compose.yml`. Note that the image names we end up with all start with `compose-local`, which is the name of the directory containing the file. Because this is the equivalent of running `docker build` in each of the directories, it only builds the images.

Having built the containers, we can start them all at once using either `docker-compose up` or `docker-compose start`:

```
$ docker-compose start
Starting db-userauth ... done
Starting svc-userauth ... done
Starting db-notes ... done
Starting svc-notes ... done
$ docker-compose stop
Stopping svc-notes ... done
Stopping svc-userauth ... done
Stopping db-notes ... done
Stopping db-userauth ... done
```

We can use `docker-compose stop` to shut down the containers. With `docker-compose start`, the containers run in the background.

We can also run `docker-compose up` to get a different experience:

```
$ docker-compose up
Recreating db-notes ... done
Starting db-userauth ... done
Starting svc-userauth ... done
Recreating svc-notes ... done
Attaching to db-userauth, db-notes, svc-userauth, svc-notes
db-userauth | [Entrypoint] MySQL Docker Image 8.0.19-1.1.15
db-userauth | [Entrypoint] Starting MySQL 8.0.19-1.1.15
db-notes    | [Entrypoint] MySQL Docker Image 8.0.19-1.1.15
db-notes    | [Entrypoint] Starting MySQL 8.0.19-1.1.15
```

If necessary, `docker-compose up` will first build the containers. In addition, it keeps the containers all in the foreground so that we can see the logging. It combines the log output for all the containers together in one output, with the container name shown at the beginning of each line. For a multi-container system such as Notes, this is very helpful.

We can check the status using this command:

```
$ docker-compose ps
  Name   Command   State   Ports
-----
```

```
db-notes    /entrypoint.sh mysqld --ch ... Up (healthy) 3306/tcp,
33060/tcp
db-userauth /entrypoint.sh mysqld --ch ... Up (healthy) 3306/tcp,
33060/tcp
svc-notes   docker-entrypoint.sh /bin/ ... Up 0.0.0.0:3000->3000/tcp
svc-userauth docker-entrypoint.sh /bin/ ... Up 0.0.0.0:5858->5858/tcp
```

This is related to running `docker ps`, but the presentation is a little different and more compact.

In `docker-compose.yml`, we insert the following declaration for `svc-userauth`:

```
# DO NOT EXPOSE THIS PORT ON PRODUCTION
ports:
- "5858:5858"
```

This means that the REST service port for `svc-userauth` was published. Indeed, in the status output, we see that the port is published. That violates our security design, but it does let us run the tests with `users/cli.mjs` from our laptop. That is, we can add users to the database as we've done so many times before.

This security violation is acceptable so long as it stays on our laptop. The `compose-local` directory is named specifically to be used with Docker Compose on our laptop.

Alternatively, we can run commands inside the `svc-userauth` container just as before:

```
$ docker exec -it svc-userauth node cli.mjs list-users
[
...
]
$ docker-compose exec svc-userauth node cli.mjs list-users
[
...
]
```

We started the Docker containers using `docker-compose`, and we can use the `docker-compose` command to interact with the containers. In this case, we demonstrated using both the `docker-compose` and `docker` commands to execute a command inside one of the containers. While there are slight differences in the command syntax, it's the same interaction with the same results.

Another test is to go into the containers and explore:

```
$ docker-compose exec svc-notes bash  
...  
$ docker-compose exec svc-userauth bash  
...
```

From there, we can try pinging each of the containers to see which containers can be reached. That will serve as a simplistic security audit to ensure that what we've created fits the security model we desired.

While doing this, we find that `svc-userauth` can ping every container, including `db-notes`. This violates the security plan and has to be changed.

Fortunately, this is easy to fix. Simply by changing the configuration, we can add a new network named `svcnet` to `docker-compose.yml`:

```
services:  
..  
svc-userauth:  
..  
networks:  
- authnet  
- svcnet  
..  
svc-notes:  
..  
networks:  
- frontnet  
- svcnet  
..  
networks:  
frontnet:  
  driver: bridge  
authnet:  
  driver: bridge  
svcnet:  
  driver: bridge
```

`svc-userauth` is no longer connected to `frontnet`, which is how we could ping `db-notes` from `svc-userauth`. Instead, `svc-userauth` and `svc-notes` are both connected to a new network, `svcnet`, which is meant to connect the service containers. Therefore, both service containers have exactly the required access to match the goals outlined at the beginning.

That's an advantage of Docker Compose. We can quickly reconfigure the system without rewriting anything other than the `docker-compose.yml` configuration file. Furthermore, the new configuration is instantly reflected in a file that can be committed to our source repository.

When you're done testing the system, simply type `CTRL + C` in the terminal:

```
^CGracefully stopping... (press Ctrl+C again to force)
Stopping db-userauth ... done
Stopping userauth    ... done
Stopping db-notes    ... done
Stopping notes       ... done
```

As shown here, this stops the whole set of containers. Occasionally, it will instead exit the user to the shell, and the containers will still be running. In that case, the user will have to use an alternative method to shut down the containers:

```
$ docker-compose down
Stopping db-userauth ... done
Stopping userauth    ... done
Stopping db-notes    ... done
Stopping notes       ... done
```

The `docker-compose` commands—`start`, `stop`, and `restart`—all serve as ways to manage the containers as background tasks. The default mode for the `docker-compose up` command is, as we've seen, to start the containers in the foreground. However, we can also run `docker-compose up` with the `-d` option, which says to detach the containers from the terminal to run in the background.

We're getting closer to our end goal. In this section, we learned how to take the Docker containers we've designed and create a system that can be easily brought up and down as a unit by running the `docker-compose` command.

While preparing to deploy this to Docker Swarm on AWS EC2, a horizontal scaling issue was found, which we can fix on our laptop. It is fairly easy with Docker Compose files to test multiple `svc-notes` instances to see whether we can scale Notes for higher traffic loads. Let's take a look at that before deploying to the swarm.

Using Redis for scaling the Notes application stack

In the previous section, we learned how to use Docker Compose to manage the Notes application stack. Looking ahead, we can see the potential need to use multiple instances of the Notes container when we deploy to Docker Swarm on AWS EC2. In this section, we will make a small modification to the Docker Compose file for an ad hoc test with multiple Notes containers. This test will show us a couple of problems. Among the available solutions are two packages that fix both problems by installing a Redis instance.

A common tactic for handling high traffic loads is to deploy multiple service instances as needed. This is called horizontal scaling, where we deploy multiple instances of a service to multiple servers. What we'll do in this section is learn a little about horizontal scaling in Docker by starting two Notes instances to see how it behaves.

As it currently exists, Notes stores some data—the session data—on the local disk space. As orchestrators such as Docker Swarm, ECS, and Kubernetes scale containers up and down, containers are constantly created and destroyed or moved from one host to another. This is done in the name of handling the traffic while optimizing the load on the available servers. In this case, whatever active data we're storing on a local disk will be lost. Losing the session data means users will be randomly logged out. The users will be rightfully upset and will then send us support requests asking what's wrong and whether we have even tested this thing!

In this section, we will learn that Notes does not behave well when we have multiple instances of `svc-notes`. To address this problem, we will add a Redis container to the Docker Compose setup and configure Notes to use Redis to solve the two problems that we have discovered. This will ensure that the session data is shared between multiple Notes instances via a Redis server.

Let's get started by performing a little ad hoc testing to better understand the problem.

Testing session management with multiple Notes service instances

We can easily verify whether Notes properly handles session data if there are multiple `svc-notes` instances. With a small modification to `compose-local/docker-compose.yml`, we can start two `svc-notes` instances, or more. They'll be on separate TCP ports, but it will let us see how Notes behaves with multiple instances of the Notes service.

Create a new service, `svc-notes-2`, by duplicating the `svc-notes` declaration. The only thing to change is the container name, which should be `svc-notes-2`, and the published port, which should be port 3020.

For example, add the following to `compose-local/docker-compose.yml`:

```
svc-notes-2:
  build: ../notes
  container_name: svc-notes-2
  depends_on:
    - db-notes
  networks:
    - frontnet
    - svcnet
  ports:
    - "3020:3020"
  restart: always
  environment:
    PORT: "3020"
```

This is the service definition for the `svc-notes-2` container we just described. Because we set the `PORT` variable, the container will listen on port 3020, which is what is advertised in the `ports` attribute.

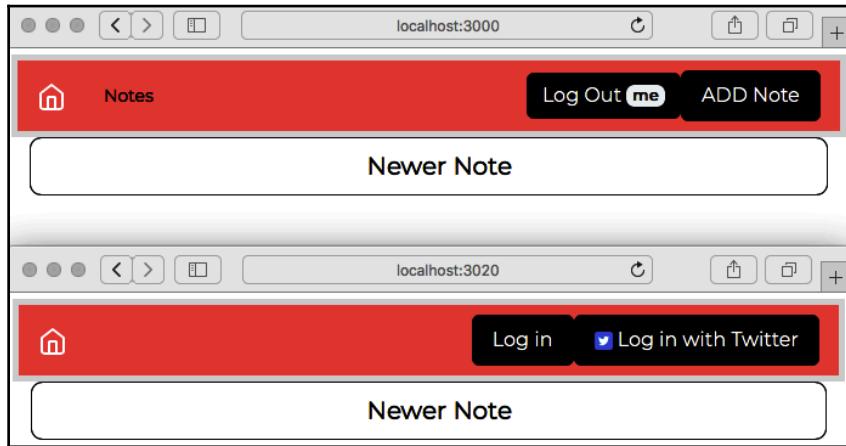
As before, when we quickly reconfigured the network configuration, notice that a simple edit to the Docker Compose file was all that was required to change things around.

Then, relaunch the Notes stack, as follows:

```
$ docker-compose up
```

In this case, there was no source code change, only a configuration change. Therefore, the containers do not need to be rebuilt, and we can simply relaunch with the new configuration.

That will give us two Notes containers on different ports. Each is configured as normal; for example, they connect to the same user authentication service. Using two browser windows, visit both at their respective port numbers. You'll be able to log in with one browser window, but you'll encounter the following situation:



The browser window on port 3020 is logged out, while the window open to port 3000 is logged in. Remember that port 3020 is `svc-notes-2`, while port 3000 is `svc-notes`. However, as you use the two windows, you'll observe some flaky behavior with regard to staying logged in.

The issue is that the session data is not shared between `svc-notes` and `svc-notes-2`. Instead, the session data is in files stored within each container.

We've identified a problem whereby keeping the session data inside the container makes it impossible to share session data across all instances of the Notes service. To fix this, we need a session store that shares the session data across processes.

Storing Express/Passport session data in a Redis server

Looking back, we saw that we might have multiple instances of `svc-notes` deployed on Docker Swarm. To test this, we created a second instance, `svc-notes-2`, and found that user sessions were not maintained between the two Notes instances. This told us that we must store session data in a shared data storage system.

There are several choices when it comes to storing sessions. While it is tempting to use the `express-session-sequelize` package, because we're already using Sequelize to manage a database, we have another issue to solve that requires the use of Redis. We'll discuss this other issue later.



For a list of Express session stores, go to <http://expressjs.com/en/resources/middleware/session.html#compatible-session-stores>.

Redis is a widely used key-value data store that is known for being very fast. It is also very easy to install and use. We won't have to learn anything about Redis, either.

Several steps are required in order to set up Redis:

1. In `compose-local/docker-compose.yml`, add the following definition to the services section:

```
redis:  
  image: "redis:5.0"  
  networks:  
    - frontnet  
  container_name: redis
```

This sets up a Redis server in a container named `redis`. This means that other services wanting to use Redis will access it at the host named `redis`.

For any `svc-notes` services you've defined (`svc-notes` and `svc-notes-2`), we must now tell the Notes application where to find the Redis server. We can do this by using an environment variable.

2. In `compose-local/docker-compose.yml`, add the following environment variable declaration to any such services:

```
svc-notes:  # Also do this for svc-notes-2  
  ...  
  environment:  
    REDIS_ENDPOINT: "redis"
```

Add this to both the `svc-notes` and `svc-notes-2` service declarations. This passes the Redis hostname to the Notes service.

3. Next, install the package:

```
$ cd notes  
$ npm install redis connect-redis --save
```

This installs the required packages. The `redis` package is a client for using Redis from Node.js and the `connect-redis` package is the Express session store for Redis.

4. We need to change the initialization in `app.mjs` to use the `connect-redis` package in order to store session data:

```
import session from 'express-session';  
import sessionFileStore from 'session-file-store';  
import ConnectRedis from 'connect-redis';  
const RedisStore = ConnectRedis(session);  
import redis from 'redis';  
var sessionStore;  
if (typeof process.env.REDIS_ENDPOINT !== 'undefined'  
  && process.env.REDIS_ENDPOINT !== '') {  
  const RedisStore = ConnectRedis(session);  
  const redisClient = redis.createClient({  
    host: process.env.REDIS_ENDPOINT  
  });  
  sessionStore = new RedisStore({ client: redisClient });  
} else {  
  const FileStore = sessionFileStore(session);  
  sessionStore = new FileStore({ path: "sessions" });  
}  
  
export const sessionCookieName = 'notescookie.sid';  
const sessionSecret = 'keyboard mouse';
```

This brings in the Redis-based session store provided by `connect-redis`.

The configuration for these packages is taken directly from the relevant documentation.



For `connect-redis`, refer to <https://www.npmjs.com/package/connect-redis>.

For `redis`, refer to <https://github.com/NodeRedis/node-redis>.

This imports the two packages and then configures the `connect-redis` package to use the `redis` package. We consulted the `REDIS_ENDPOINT` environment variable to configure the `redis` client object. The result landed in the same `sessionStore` variable we used previously. Therefore, no other change is required in `app.mjs`.

If no Redis endpoint is specified, we instead revert to the file-based session store. We might not always deploy Notes in a context where we can run Redis; for example, while developing on our laptop. Therefore, we require the option of not using Redis, and, at the moment, the choice looks to be between using Redis or the filesystem to store session data.

With these changes, we can relaunch the Notes application stack. It might help to relaunch the stack using the following command:

```
$ docker-compose up --build --force-recreate
```

Because source file changes were made, the containers need to be rebuilt. These options ensure that this happens.

We'll now be able to connect to both the Notes service on `http://localhost:3000` (`svc-notes`) and the service on `http://localhost:3020` (`svc-notes-2`), and it will handle the login session on both services.

Another issue should be noted, however, and this is the fact that real-time notifications are not sent between the two servers. To see this, set up four browser windows, two for each of the servers. Navigate all of them to the same note. Then, add and delete some comments. Only the browser windows connected to the same server will dynamically show changes to the comments. Browser windows connected to the other server will not.

This is the second horizontal scaling issue. Fortunately, its solution also involves the use of Redis.

Distributing Socket.IO messages using Redis

While testing what happens when we have multiple `svc-notes` containers, we found that login/logout was not reliable. We fixed this by installing a Redis-based session store to keep session data in a place that is accessible by multiple containers. But we also noticed another issue: the fact that the Socket.IO-based messaging did not reliably cause updates in all browser windows.

Remember that the updates we want to happen in the browser are triggered by updates to the SQNotes or SQMessages tables. The events emitted by updating either table are emitted by the server making the update. An update happening in one service container (say, svc-notes-2) will emit an event from that container, but not from the other one (say, svc-notes). There is no mechanism for the other containers to know that they should emit such events.



The Socket.IO documentation talks about this situation:

<https://socket.io/docs/using-multiple-nodes/>

The Socket.IO team provides the `socket.io-redis` package as the solution to this problem. It ensures that events emitted through Socket.IO by any server will be passed along to other servers so that they can also emit those events.

Since we already have the Redis server installed, we simply need to install the package and configure it as per the instructions. Again, we will not need to learn anything about Redis:

```
$ npm install socket.io-redis --save
```

This installs the `socket.io-redis` package.

Then, we configure it in `app.mjs`, as follows:

```
export const io = socketio(server);
io.use(passportSocketIo.authorize({
...
}));

import redisIO from 'socket.io-redis';
if (typeof process.env.REDIS_ENDPOINT !== 'undefined'
  && process.env.REDIS_ENDPOINT !== '') {
  io.adapter(redisIO({ host: process.env.REDIS_ENDPOINT, port: 6379
}));
```

The only change is to add the lines in bold. The `socket.io-redis` package is what the Socket.IO team calls an adapter. Adapters are added to Socket.IO by using the `io.adapter` call.

We only connect this adapter if a Redis endpoint has been specified. As before, this is so that Notes can be run without Redis as needed.

Nothing else is required. If you relaunch the Notes application stack, you will now receive updates in every browser window connected to every instance of the Notes service.

In this section, we thought ahead about deployment to a cloud-hosting service. Knowing that we might want to implement multiple Notes containers, we tested this scenario on our laptop and found a couple of issues. They were easily fixed by installing a Redis server and adding a couple of packages.

We're getting ready to finish this chapter, and there's one task to take care of before we do. The `svc-notes-2` container was useful for ad hoc testing, but it is not the correct way to deploy multiple Notes instances. Therefore, in `compose-local/docker-compose.yml`, comment out the `svc-notes-2` definition.

This gave us some valuable exposure to a new tool that's widely used—Redis. Our application now also appears to be ready for deployment. We'll take care of that in the next chapter.

Summary

In this chapter, we took a huge step toward the vision of deploying Notes on a cloud-hosting platform. Docker containers are widely used on cloud-hosting systems for application deployment. Even if we don't end up using the Docker Compose file once, we can still carry out the deployment and we have worked out how to Dockerize every aspect of the Notes stack.

In this chapter, we learned not only about creating Docker images for Node.js applications, but also about launching a whole system of services comprising a web application. We have learned that a web application is not just about the application code but also the databases, the frameworks we use, and even other services, such as Redis.

For that purpose, we learned both how to create our own Docker containers as well as how to use third-party containers. We learned how to launch containers using `docker run` and Docker Compose. We learned how to build custom Docker containers using a Dockerfile, and how to customize third-party containers.

For connecting containers, we learned about the Docker bridge network. This is useful on a single-host Docker installation and is a private communication channel where containers can find each other. As a private channel, the bridge network is relatively safe from outside intrusion, giving us a way to securely tie services together. We had the opportunity to try different network architectures inside Docker and to explore the security implications of each. We learned that Docker offers an excellent way to securely deploy persistent services on a host system.

Looking ahead to the task of deploying Notes on a cloud hosting service, we did some ad hoc testing with multiple instances of the Notes service. This highlighted a few issues that will crop up with multiple instances, and we remedied those issues by adding Redis to the application stack.

This gave us a well-rounded view of how Node.js services are prepared for deployment to cloud-hosting providers. Remember that our goal is to deploy the Notes application as Docker containers on AWS EC2 as an example of cloud deployment. In this chapter, we explored different aspects of Dockerizing a Node.js application stack, giving us a solid grounding in deploying services with Docker. We're now ready to take this application to a server on the public internet.

In the next chapter, we will learn about two very important technologies. The first is **Docker Swarm**, which is a Docker orchestrator that comes bundled with Docker. We'll learn how to deploy our Docker stack as services in a Swarm that we'll build on the AWS EC2 infrastructure. The second technology we'll learn about is Terraform, which is an open source tool for describing service configuration on cloud-hosting systems. We'll use it to describe the AWS EC2 configuration for the Notes application stack.

12

Deploying a Docker Swarm to AWS EC2 with Terraform

So far in this book, we've created a Node.js-based application stack comprising two Node.js microservices, a pair of MySQL databases, and a Redis instance. In the previous chapter, we learned how to use Docker to easily launch those services, intending to do so on a cloud hosting platform. Docker is widely used for deploying services such as ours, and there are lots of options available to us for deploying Docker on the public internet.

Because **Amazon Web Services (AWS)** is a mature feature-filled cloud hosting platform, we've chosen to deploy there. There are many options available for hosting Notes on AWS. The most direct path from our work in *Chapter 11, Deploying Node.js Microservices with Docker*, is to create a Docker Swarm cluster on AWS. That enables us to directly reuse the Docker compose file we created.

Docker Swarm is one of the available Docker orchestration systems. These systems manage a set of Docker containers on one or more Docker host systems. In other words, building a swarm requires provisioning one or more server systems, installing Docker Engine on each, and enabling swarm mode. Docker Swarm is built into Docker Engine, and it's a matter of a few commands to join those servers together in a swarm. We can then deploy Docker-based services to the swarm, and the swarm distributes the containers among the server systems, monitoring each container, restarting any that crash, and so on.

Docker Swarm can be used in any situation with multiple Docker host systems. It is not tied to AWS because we can rent suitable servers from any of hundreds of web hosting providers around the world. It's sufficiently lightweight that you can even experiment with Docker Swarm using **virtual machine (VM)** instances (Multipass, VirtualBox, and so on) on a laptop.

In this chapter, we will use a set of AWS **Elastic Compute Cloud (EC2)** instances. EC2 is the AWS equivalent of a **virtual private server (VPS)** that we would rent from a web hosting provider. The EC2 instances will be deployed within an AWS **virtual private cloud (VPC)**, along with a network infrastructure on which we'll implement the deployment architecture we outlined earlier.

Let's talk a little about the cost since AWS can be costly. AWS offers what's called the Free Tier, where, for certain services, the cost is zero as long as you stay below a certain threshold. In this chapter, we'll strive to stay within the free tier, except that we will have three EC2 instances deployed for a while, which is beyond the free tier for EC2 usage. If you are sensitive to the cost, it is possible to minimize it by destroying the EC2 instances when not needed. We'll discuss how to do this later.

The following topics will be covered in this chapter:

- Signing up with AWS and configuring the AWS **command-line interface (CLI)**
- An overview of the AWS infrastructure to be deployed
- Using Terraform to create an AWS infrastructure
- Setting up a Docker Swarm cluster on AWS EC2
- Setting up **Elastic Container Registry (ECR)** repositories for Notes Docker images
- Creating a Docker stack file for deployment to Docker Swarm
- Provisioning EC2 instances for a full Docker Swarm
- Deploying the Notes stack file to the swarm

You will be learning a lot in this chapter, starting with how to get started with the AWS Management Console, setting up **Identity and Access Management (IAM)** users on AWS, and how to set up the AWS command-line tools. Since the AWS platform is so vast, it is important to get an overview of what it entails and the facilities we will use in this chapter. Then, we will learn about Terraform, a leading tool for configuring services on all kinds of cloud platforms. We will learn how to use it to configure AWS resources such as the VPC, the associated networking infrastructure, and how to configure EC2 instances. We'll next learn about Docker Swarm, the orchestration system built into Docker, how to set up a swarm, and how to deploy applications in a swarm.

For that purpose, we'll learn about Docker image registries, the AWS **Elastic Container Registry (ECR)**, how to push images to a Docker registry, and how to use images from a private registry in a Docker application stack. Finally, we'll learn about creating a Docker stack file, which lets you describe Docker services to deploy in a swarm.

Let's get started.

Signing up with AWS and configuring the AWS CLI

To use AWS services you must, of course, have an AWS account. The AWS account is how we authenticate ourselves to AWS and is how AWS charges us for services.

As a first step, go to <https://aws.amazon.com> and sign up for an account.



The Amazon Free Tier is a way to experience AWS services at zero cost: <https://aws.amazon.com/free/>.

Documentation is available at <https://docs.aws.amazon.com>.

AWS has two kinds of accounts that we can use, as follows:

- The **root account** is what's created when we sign up for an AWS account. The root account has full access to AWS services.
- An **IAM user account** is a less privileged account you can create within your root account. The owner of a root account creates IAM accounts, assigning the scope of permissions to each IAM account.

It is bad form to use the root account directly since the root account has complete access to AWS resources. If the account credentials for your root account were to be leaked to the public, significant damage could be done to your business. If the credentials for an IAM user account were leaked, the damage is limited to the resources controlled by that user account as well as by the privileges assigned to that account. Furthermore, IAM user credentials can be revoked at any time, and then new credentials generated, preventing anyone who is holding the leaked credentials from doing any further damage. Another security measure is to enable **multi-factor authentication (MFA)** for all accounts.

If you have not already done so, proceed to the AWS website at one of the preceding links and sign up for an account. Remember that the account created that way is your AWS root account.

Our first step is to familiarize ourselves with the AWS Management Console.

Finding your way around the AWS account

Because there are so many services on the AWS platform, it can seem like a maze of twisty little passages, all alike. However, with a little orientation, we can find our way around.

First, look at the navigation bar at the top of the window. On the right, there are three dropdowns. The first has your account name and has account-related choices. The second lets you select which AWS region is your default. AWS has divided its infrastructure into *regions*, which essentially means the area of the world where AWS data centers are located. The third connects you with AWS Support.

On the left is a dropdown marked **Services**. This shows you the list of all AWS services. Since the **Services** list is unwieldy, AWS gives you a search box. Simply type in the name of the service, and it will show up. The AWS Management Console home page also has this search box.

While we are finding our way around, let's record the account number for the root account. We'll need this information later. In the **Account** dropdown, select **My Account**. The account ID is there, along with your account name.

It is recommended to set up MFA on your AWS root account. MFA simply means to authenticate a person in multiple ways. For example, a service might use a code number sent via a text message as a second authentication method, alongside asking for a password. The theory is that the service is more certain of who we are if it verifies both that we've entered a correct password and that we're carrying the same cell phone we had carried on other days.

To set up MFA on your root account, go to the **My Security Credentials** dashboard. A link to that dashboard can be found in the AWS Management Console menu bar. This brings you to a page controlling all forms of authentication with AWS. From there, you follow the directions on the AWS website. There are several possible tools for implementing MFA. The simplest tool is to use the **Google Authenticator** application on your smartphone. Once you set up MFA, every login to the root account will require a code to be entered from the authenticator app.

So far, we have dealt with the online AWS Management Console. Our real goal is to use command-line tools, and to do that, we need the AWS CLI installed and configured on our laptop. Let's take care of that next.

Setting up the AWS CLI using AWS authentication credentials

The AWS CLI tool is a download available through the AWS website. Under the covers, it uses the AWS **application programming interface (API)**, and it also requires that we download and install authentication tokens.

Once you have an account, we can prepare the AWS CLI tool.

The AWS CLI enables you to interact with AWS services from the command line of your laptop. It has an extensive set of sub-commands related to every AWS service.



Instructions to install the AWS CLI can be found here: <https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html>.

Instructions to configure the AWS CLI can be found here: <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html>.

Once you have installed the AWS CLI tool on your laptop, we must configure what is known as a *profile*.

AWS supplies an AWS API that supports a broad range of tools for manipulating the AWS infrastructure. The AWS CLI tools use that API, as do third-party tools such as Terraform. Using the API requires access tokens, so of course, both the AWS CLI and Terraform require those same tokens.

To get the AWS API access tokens, go to the **My Security Credentials** dashboard and click on the **Access Keys** tab.

There will be a button marked **Create New Access Key**. Click on this and you will be shown two security tokens, the **Access Key ID** and the **Secret Access Key**. You will be given a chance to download a **comma-separated values (CSV)** file containing these keys. The CSV file looks like this:

```
$ cat ~/Downloads/accessKeys.csv
Access key ID,Secret access key
AKIAZKY7BHGBVWEKCU7H,41WctREbazP9fULN1C5CrQ0L92iSO27fiVGJKU2A
```

You will receive a file that looks like this. These are the security tokens that identify your account. Don't worry, as no secrets are being leaked in this case. Those particular credentials have been revoked. The good news is that you can revoke these credentials at any time and download new credentials.

Now that we have the credentials file, we can configure an AWS CLI profile.

The `aws configure` command, as the name implies, takes care of configuring your AWS CLI environment. This asks a series of questions, the first two of which are those keys. The interaction looks like this:

```
$ aws configure --profile root-user
AWS Access Key ID [*****E3GA]: ... ENTER ACCESS KEY
AWS Secret Access Key [*****J9cp]: ... ENTER SECRET KEY
Default region name [us-west-2]:
Default output format [json]:
```

For the first two prompts, paste in the keys you downloaded. The **Region name** prompt selects the default Amazon AWS data center in which your service will be provisioned. AWS has facilities all around the world, and each locale has a code name such as `us-west-2` (located in Oregon). The last prompt asks how you wish the AWS CLI to present information to you.

For the region code, in the AWS console, take a look at the **Region** dropdown. This shows you the available regions, describing locales, and the region code for each. For the purpose of this project, it is good to use an AWS region located near you. For production deployment, it is best to use the region closest to your audience. It is possible to configure a deployment that works across multiple regions so that you can serve clients in multiple areas, but that implementation is way beyond what we'll cover in this book.

By using the `--profile` option, we ensured that this created a named profile. If we had left off that option, we would have instead created a profile named `default`. For any of the `aws` commands, the `--profile` option selects which profile to use. As the name suggests, the `default` profile is the one used if we leave off the `--profile` option.

A better choice is to be explicit at all times in which an AWS identity is being used. Some guides suggest to not create a default AWS profile at all, but instead to always use the `--profile` option to be certain of always using the correct AWS profile.

An easy way to verify that AWS is configured is to run the following commands:

```
$ aws s3 ls
Unable to locate credentials. You can configure credentials by running
"aws configure".
$ aws s3 ls --profile root-user
$ export AWS_PROFILE=root-user
$ aws s3 ls
```

The AWS **Simple Storage Service (S3)** is a cloud file-storage system, and we are running these commands solely to verify the correct installation of the credentials. The `ls` command lists any files you have stored in S3. We don't care about the files that may or may not be in an S3 bucket, but whether this executes without error.

The first command shows us that execution with no `--profile` option, and no `default` profile, produces an error. If there were a `default` AWS profile, that would have been used. However, we did not create a `default` profile, so therefore no profile was available and we got an error. The second shows the same command with an explicitly named profile. The third shows the `AWS_PROFILE` environment variable being used to name the profile to be deployed.

Using the environment variables supported by the AWS CLI tool, such as `AWS_PROFILE`, lets us skip using command-line options such as `--profile` while still being explicit about which profile to use.

As we said earlier, it is important that we interact with AWS via an IAM user, and therefore we must learn how to create an IAM user account. Let's do that next.

Creating an IAM user account, groups, and roles

We could do everything in this chapter using our root account but, as we said, that's bad form. Instead, it is recommended to create a second user—an IAM user—and give it only the permissions required by that user.

To get to the IAM dashboard, click on **Services** in the navigation bar, and enter **IAM**. IAM stands for Identity and Access Management. Also, the **My Security Credentials** dashboard is part of the IAM service, so we are probably already in the IAM area.

The first task is to create a role. In AWS, roles are used to associate privileges with a user account. You can create roles with extremely limited privileges or an extremely broad range of privileges.

In the IAM dashboard, you'll find a navigation menu on the left. It has sections for users, groups, roles, and other identity management topics. Click on the **Roles** choice. Then, in the **Roles** area, click on **Create Role**. Perform the following steps:

1. Under **Type of trusted identity**, select **Another AWS account**. Enter the account ID, which you will have recorded earlier while familiarizing yourself with the AWS account. Then, click on **Next**.
2. On the next page, we select the permissions for this role. For our purpose, select **AdministratorAccess**, a privilege that grants full access to the AWS account. Then, click on **Next**.
3. On the next page, you can add tags to the role. We don't need to do this, so click **Next**.
4. On the last page, we give a name to the role. Enter **admin** because this role has administrator permissions. Click on **Create Role**.

You'll see that the role, **admin**, is now listed in the **Role** dashboard. Click on **admin** and you will be taken to a page where you can customize the role further. On this page, notice the characteristic named **Role ARN**. Record this **Amazon Resource Name (ARN)** for future reference.

ARNs are identifiers used within AWS. You can reliably use this ARN in any area of AWS where we can specify a role. ARNs are used with almost every AWS resource.

Next, we have to create an administrator group. In IAM, users are assigned to groups as a way of passing roles and other attributes to a group of IAM user accounts. To do this, perform the following steps:

1. In the left-hand navigation menu, click on **Group**, and then, in the group dashboard, click on **Create Group**.
2. For the group name, enter **Administrators**.
3. Skip the **Attach Policy** page, click **Next Step**, and then, on the **Review** page, simply click **Create Group**.
4. This creates a group with no permissions and directs you back to the group dashboard.
5. Click on the **Administrators** group, and you'll be taken to the overview page. Record the ARN for the group.
6. Click on **Permissions** to open that tab, and then click on the **Inline policies** section header. We will be creating an inline policy, so click on the **Click here** link.
7. Click on **Custom Policy**, and you'll be taken to the policy editor.
8. For the policy name, enter `AssumeAdminRole`. Below that is an area where we enter a block of **JavaScript Object Notation (JSON)** code describing the policy. Once that's done, click the **Apply Policy** button.

The policy document to use is as follows:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "sts:AssumeRole",  
            "Resource": "arn:aws:iam::ACCOUNT-ID:role/admin"  
        }  
    ]  
}
```

This describes the policy created for the **Administrators** group. It gives that group the rights we specified in the admin role earlier. The **Resource** tag is where we enter the ARN for the **admin** group that was created earlier. Make sure to put the entire ARN into this field.

Navigate back to the **Groups** area, and click on **Create Group** again. We'll create a group, **NotesDeveloper**, for use by developers assigned to the Notes project. It will give those user accounts some additional privileges. Perform the following steps:

1. Enter **NotesDeveloper** as the group name. Then, click **Next**.
2. For the **Attach Policy** page, there is a long list of policies to consider; for example, **AmazonRDSFullAccess**, **AmazonEC2FullAccess**, **IAMFullAccess**, **AmazonEC2ContainerRegistryFullAccess**, **AmazonS3FullAccess**, **AdministratorAccess**, and **AmazonElasticFileSystemFullAccess**.
3. Then, click **Next**, and if everything looks right on the **Review** page, click **Create Group**.

These policies cover the services required to finish this chapter. AWS error messages that stipulate that the user is not privileged enough to access that feature do a good job of telling you the required privilege. If it is a privilege the user needs, then come back to this group and add the privilege.

In the left-hand navigation, click on **Users** and then on **Create User**. This starts the steps involved in creating an IAM user, described as follows:

1. For the username, enter **notes-app**, since this user will manage all resources related to the Notes application. For **Access type**, click on both **Programmatic access** and **AWS management console access** since we will be using both. The first grants the ability to use the AWS CLI tools, while the second covers the AWS console. Then, click on **Next**.
2. For permissions, select **Add User to Group** and then select both the **Administrators** and **NotesDeveloper** groups. This adds the user to the groups you select. Then, click on **Next**.
3. There is nothing more to do, so keep clicking **Next** until you get to the **Review** page. If you're satisfied, click on **Create user**.

You'll be taken to a page that declares **Success**. On this page, AWS makes available access tokens (a.k.a. security credentials) that can be used with this account.

Download these credentials before you do anything else. You can always revoke the credentials and generate new access tokens at any time.

Your newly created user is now listed in the **Users** section. Click on that entry, because we have a couple of data items to record. The first is obviously the ARN for the user account. The second is a **Uniform Resource Locator (URL)** you can use to sign in to AWS as this user. For that URL, click on the **Security Credentials** tab and the sign-in link will be there.

It is recommended to also set up MFA for the IAM account. The **My Security Credentials** choice in the AWS taskbar gets you to the screen containing the button to set up MFA. Refer back a few pages to our discussion of setting up MFA for the root account.

To test the new user account, sign out and then go to the sign-in URL. Enter the username and password for the account, and then sign in.

Before finishing this section, return to the command line and run the following command:

```
$ aws configure --profile notes-app  
... Fill in configuration
```

This will create another AWS CLI profile, this time for the notes-app IAM user.

Using the AWS CLI, we can list the users in our account, as follows:

```
$ aws iam list-users --profile root-user  
{  
  "Users": [ {  
    "Path": "/",  
    "UserName": "notes-app",  
    "UserId": "AIDARNEXAMPLEM35LE",  
    "Arn": "arn:aws:iam::USER-ID:user/notes-app",  
    "CreateDate": "2020-03-08T02:19:39+00:00",  
    "PasswordLastUsed": "2020-04-05T15:34:28+00:00"  
  }  
]
```

This is another way to verify that the AWS CLI is correctly installed. This command queries the user information from AWS, and if it executes without error then you've configured the CLI correctly.

AWS CLI commands follow a similar structure, where there is a series of sub-commands followed by options. In this case, the sub-commands are `aws`, `iam`, and `list-users`. The AWS website has extensive online documentation for the AWS CLI tool.

Creating an EC2 key pair

Since we'll be using EC2 instances in this exercise, we need an EC2 key pair. This is an encrypted certificate that serves the same purpose as the normal **Secure Shell (SSH)** key we use for passwordless login to a server. In fact, the key-pair file serves the same purpose, allowing passwordless login with SSH to EC2 instances. Perform the following steps:

1. Log in to the AWS Management Console and then select the region you're using.
2. Next, navigate to the **EC2** dashboard—for example, by entering **EC2** in the search box.
3. In the navigation sidebar, there is a section labeled **Network & Security**, containing a link for **Key pair**.
4. Click on that link. In the upper-right corner is a button marked **Create key pair**. Click on this button, and you will be taken to the following screen:

The screenshot shows the 'Create key pair' wizard. At the top, the breadcrumb navigation shows 'EC2 > Key pairs > Create key pair'. The main title is 'Create key pair'. Below it, a 'Key pair' section defines what a key pair is: 'A key pair, consisting of a private key and a public key, is a set of security credentials that you use to prove your identity when connecting to an instance.' A 'Name' field contains 'notes-app-key-pair'. A note below says 'The name can be up to 255 characters long. Valid characters include _, -, a-z, A-Z, and 0-9.' Under 'File format', 'pem' is selected (radio button is checked) with the note 'For use with OpenSSH'. The 'ppk' option is also listed with the note 'For use with PuTTY'. At the bottom right are 'Cancel' and 'Create key pair' buttons.

5. Enter the desired name for the key pair. Depending on the SSH client you're using, use either a **pem** (used for the `ssh` command) or a **ppk** (used for PuTTY) formatted key-pair file.

6. Click on **Create key pair** and you'll be returned to the dashboard, and the key-pair file will download in your browser.
7. After the key-pair file is downloaded, it is required to make it read-only, which you can do by using the following command:

```
$ chmod 400 /path/to/keypairfile.pem
```

Substitute here the pathname where your browser downloaded the file.

For now, just make sure this file is correctly stored somewhere. When we deploy EC2 instances, we'll talk more about how to use it.

We have familiarized ourselves with the AWS Management Console, and created for ourselves an IAM user account. We have proved that we can log in to the console using the sign-in URL. While doing that, we copied down the AWS access credentials for the account.

We have completed the setup of the AWS command-line tools and user accounts. The next step is to set up Terraform.

An overview of the AWS infrastructure to be deployed

AWS is a complex platform with dozens of services available to us. This project will touch on only the part required to deploy Notes as a Docker swarm on EC2 instances. In this section, let's talk about the infrastructure and AWS services we'll put to use.

An AWS VPC is what it sounds like—namely, a service within AWS where you build your own private cloud service infrastructure. The AWS team designed the VPC service to look like something that you would construct in your own data center, but implemented on the AWS infrastructure. This means that the VPC is a container to which everything else we'll discuss is attached.

The AWS infrastructure is spread across the globe into what AWS calls regions. For example, `us-west-1` refers to Northern California, `us-west-2` refers to Oregon, and `eu-central-1` refers to Frankfurt. For production deployment, it is recommended to use a region nearer your customers, but for experimentation, it is good to use the region closest to you. Within each region, AWS further subdivides its infrastructure into **availability zones** (a.k.a. **AZs**). An AZ might correspond to a specific building at an AWS data center site, but AWS often recommends that we deploy infrastructure to multiple AZs for reliability. In case one AZ goes down, the service can continue in the AZs that are running.

When we allocate a VPC, we specify an address range for resources deployed within the VPC. The address range is specified with a **Classless Inter-Domain Routing (CIDR)** specifier. These are written as `10.3.0.0/16` or `10.3.20.0/24`, which means any **Internet Protocol version 4 (IPv4)** address starting with `10.3` and `10.3.20`, respectively.

Every device we attach to a VPC will be attached to a subnet, a virtual object similar to an Ethernet segment. Each subnet will be assigned a CIDR from the main range. A VPC assigned the `10.3.0.0/16` CIDR might have a subnet with a CIDR of `10.3.20.0/24`. Devices attached to the subnet will have an IP address assigned within the range indicated by the CIDR for the subnet.

EC2 is AWS's answer to a VPS that you might rent from any web hosting provider. An EC2 instance is a virtual computer in the same sense that Multipass or VirtualBox lets you create a virtual computer on your laptop. Each EC2 instance is assigned a **central processing unit (CPU)**, memory, disk capacity, and at least one network interface. Hence, an EC2 instance is attached to a subnet and is assigned an IP address from the subnet's assigned range.

By default, a device attached to a subnet has no internet access. The internet gateway and **network address translation (NAT)** gateway resources on AWS play a critical role in connecting resources attached to a VPC via the internet. Both are what is known as an internet router, meaning that both handle the routing of internet traffic from one network to another. Because a VPC contains a VPN, these gateways handle traffic between that network and the public internet, as follows:

- **Internet gateway:** This handles two-way routing, allowing a resource allocated in a VPC to be reachable from the public internet. An internet gateway allows external traffic to enter the VPC, and it also allows resources in the VPC to access resources on the public internet.

- **NAT gateway:** This handles one-way routing, meaning that resources on the VPC will be able to access resources on the public internet, but does not allow external traffic to enter the VPC. To understand the NAT gateway, think about a common home Wi-Fi router because they also contain a NAT gateway. Such a gateway will manage a local IP address range such as 192.168.0.0/16, while the **internet service provider (ISP)** might assign a public IP address such as 107.123.42.231 to the connection. Local IP addresses, such as 192.168.1.45, will be assigned to devices connecting to the NAT gateway. Those local IP addresses do not appear in packets sent to the public internet. Instead, the NAT gateway translates the IP addresses to the public IP address of the gateway, and then when reply packets arrive, it translates the IP address to that of the local device. NAT translates IP addresses from the local network to the IP address of the NAT gateway.

In practical terms, this determines the difference between a private subnet and a public subnet. A public subnet has a routing table that sends traffic for the public internet to an internet gateway, whereas a private subnet sends its public internet traffic to a NAT gateway.

Routing tables describe how to route internet traffic. Inside any internet router, such as an internet gateway or a NAT gateway, is a function that determines how to handle internet packets destined for a location other than the local subnet. The routing function matches the destination address against routing table entries, and each routing table entry says where to forward matching packets.

Attached to each device deployed in a VPC is a security group. A security group is a firewall controlling what kind of internet traffic can enter or leave that device. For example, an EC2 instance might have a web server supporting HTTP (port 80) and HTTPS (port 443) traffic, and the administrator might also require SSH access (port 22) to the instance. The security group would be configured to allow traffic from any IP address on ports 80 and 443 and to allow traffic on port 22 from IP address ranges used by the administrator.

A network **access control list (ACL)** is another kind of firewall that's attached to subnets. It, too, describes which traffic is allowed to enter or leave the subnet. The security groups and network ACLs are part of the security protections provided by AWS.

If a device connected to a VPC does not seem to work correctly, there might be an error in the configuration of these parts. It's necessary to check the security group attached to the device, and to the NAT gateway or internet gateway, and that the device is connected to the expected subnet, the routing table for the subnet, and any network ACLs.

Using Terraform to create an AWS infrastructure

Terraform is an open source tool for configuring a cloud hosting infrastructure. It uses a declarative language to describe the configuration of cloud services. Through a long list of plugins, called providers, it has support for a variety of cloud services. In this chapter, we'll use Terraform to describe AWS infrastructure deployments.



To install Terraform, download an installer from <https://www.terraform.io/downloads.html>.

Alternatively, you will find the Terraform CLI available in many package management systems.

Once installed, you can view the Terraform help with the following command:

```
$ terraform help
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
  apply      Builds or changes infrastructure
  console    Interactive console for Terraform interpolations
  destroy    Destroy Terraform-managed infrastructure
  ...
  init       Initialize a Terraform working directory
  output     Read an output from a state file
  plan       Generate and show an execution plan
  providers  Prints a tree of the providers used in the configuration
  ...
```

Terraform files have a `.tf` extension and use a fairly simple, easy-to-understand declarative syntax. Terraform doesn't care which filenames you use or the order in which you create the files. It simply reads all the files with a `.tf` extension and looks for resources to deploy. These files do not contain executable code, but declarations. Terraform reads these files, constructs a graph of dependencies, and works out how to implement the declarations on the cloud infrastructure being used.

An example declaration is as follows:

```
variable "base_cidr_block" { default = "10.1.0.0/16" }

resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}
```

The first word, `resource` or `variable`, is the block type, and in this case, we are declaring a resource and a variable. Within the curly braces are the arguments to the block, and it is helpful to think of these as attributes.

Blocks have labels—in this case, the labels are `aws_vpc` and `main`. We can refer to this specific resource elsewhere by joining the labels together as `aws_vpc.main`. The name, `aws_vpc`, comes from the AWS provider and refers to VPC elements. In many cases, a block—be it a resource or another kind—will support attributes that can be accessed. For example, the CIDR for this VPC can be accessed as `aws_vpc.main.cidr_block`.

The general structure is as follows:

```
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

The block types include `resource`, which declares something related to the cloud infrastructure, `variable`, which declares a named value, `output`, which declares a result from a module, and a few others.

The structure of the block labels varies depending on the block type. For resource blocks, the first block label refers to the kind of resource, while the second is a name for the specific instance of that resource.

The type of arguments also varies depending on the block type. The Terraform documentation has an extensive reference to every variant.

A Terraform module is a directory containing Terraform scripts. When the `terraform` command is run in a directory, it reads every script in that directory to build a tree of objects.

Within modules, we are dealing with a variety of values. We've already discussed resources, variables, and outputs. A resource is essentially a value that is an object related to something on the cloud hosting platform being used. A variable can be thought of as an input to a module because there are multiple ways to provide a value for a variable. The output values are, as the name implies, the output from a module. Outputs can be printed on the console when a module is executed, or saved to a file and then used by other modules. The code relating to this can be seen in the following snippet:

```
variable "aws_region" {
    default = "us-west-2"
    type = "string"
    description = "Where in the AWS world the service will be hosted"
}

output "vpc_arn" { value = aws_vpc.notes.arn }
```

This is what the `variable` and `output` declarations look like. Every value has a data type. For variables, we can attach a description to aid in their documentation. The declaration uses the word `default` rather than `value` because there are multiple ways (such as Terraform command-line arguments) to specify a value for a variable. Terraform users can override the default value in several ways, such as the `--var` or `--var-file` command-line options.

Another type of value is local. Locals exist only within a module because they are neither input values (variables) nor output values, as illustrated in the following code snippet:

```
locals {
    vpc_cidr = "10.1.0.0/16"
    cidr_subnet1 = cidrsubnet(local.vpc_cidr, 8, 1)
    cidr_subnet2 = cidrsubnet(local.vpc_cidr, 8, 2)
    cidr_subnet3 = cidrsubnet(local.vpc_cidr, 8, 3)
    cidr_subnet4 = cidrsubnet(local.vpc_cidr, 8, 4)
}
```

In this case, we've defined several locals related to the CIDR of subnets to be created within a VPC. The `cidrsubnet` function is used to calculate subnet masks such as `10.1.1.0/24`.

Another important feature of Terraform is the provider plugin. Each cloud system supported by Terraform requires a plugin module that defines the specifics of using Terraform with that platform.

One effect of the provider plugins is that Terraform makes no attempt to be platform-independent. Instead, all declarable resources for a given platform are unique to that platform. You cannot directly reuse Terraform scripts for AWS on another system such as Azure because the resource objects are all different. What you can reuse is the knowledge of how Terraform approaches the declaration of cloud resources.

Another task is to look for a Terraform extension for your programming editor. Some of them have support for Terraform, with syntax coloring, checking for simple errors, and even code completion.

That's enough theory, though. To really learn this, we need to start using Terraform. In the next section, we'll begin by implementing the VPC structure within which we'll deploy the Notes application stack.

Configuring an AWS VPC with Terraform

An AWS VPC is what it sounds like—namely, a service within AWS to hold cloud services that you've defined. The AWS team designed the VPC service to look something like what you would construct in your own data center, but implemented on the AWS infrastructure.

In this section, we will construct a VPC consisting of a public subnet and a private subnet, an internet gateway, and security group definitions.

In the project work area, create a directory, `terraform-swarm`, that is a sibling to the `notes` and `users` directories.

In that directory, create a file named `main.tf` containing the following:

```
provider "aws" {  
    profile = "notes-app"  
    region  = var.aws_region  
}
```

This says to use the AWS provider plugin. It also configures this script to execute using the named AWS profile. Clearly, the AWS provider plugin requires AWS credential tokens in order to use the AWS API. It knows how to access the credentials file set up by `aws configure`.



To learn more about configuring the AWS provider plugin, refer to <https://www.terraform.io/docs/providers/aws/index.html>.

As shown here, the AWS plugin will look for the AWS credentials file in its default location, and use the `notes-app` profile name.

In addition, we have specified which AWS region to use. The reference, `var.aws_region`, is a Terraform variable. We use variables for any value that can legitimately vary. Variables can be easily customized to any value in several ways.

To support the variables, we create a file named `variables.tf`, starting with this:

```
variable "aws_region" { default = "us-west-2" }
```

The `default` attribute sets a default value for the variable. As we saw earlier, the declaration can also specify the data type for a variable, and a description.

With this, we can now run our first Terraform command, as follows:

```
$ terraform init
Initializing the backend...
Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.56.0...
The following providers do not have any version constraints in
configuration, so the latest version was installed.
...
* provider.aws: version = "~> 2.56"

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan"
to see any changes that are required for your infrastructure. All
Terraform commands should now work.
```

This initializes the current directory as a Terraform workspace. You'll see that it creates a directory, `.terraform`, and a file named `terraform.tfstate` containing data collected by Terraform. The `.tfstate` files are what is known as state files. These are in JSON format and store the data Terraform collects from the platform (in this case, AWS) regarding what has been deployed. State files must not be committed to source code repositories because it is possible for sensitive data to end up in those files. Therefore, a `.gitignore` file listing the state files is recommended.

The instructions say we should run `terraform plan`, but before we do that, let's declare a few more things.

To declare the VPC and its related infrastructure, let's create a file named `vpc.tf`. Start with the following command:

```
resource "aws_vpc" "notes" {
  cidr_block = var.vpc_cidr
  enable_dns_support = var.enable_dns_support
  enable_dns_hostnames = var.enable_dns_hostnames

  tags = {
    Name = var.vpc_name
  }
}
```

This declares the VPC. This will be the container for the infrastructure we're creating.

The `cidr_block` attribute determines the IPv4 address space that will be used for this VPC. The CIDR notation is an internet standard, and an example would be `10.0.0.0/16`. That CIDR would cover any IP address starting with the `10.0` octets.

The `enable_dns_support` and `enable_dns_hostnames` attributes determine whether **Domain Name System (DNS)** names will be generated for certain resources attached to the VPC. DNS names can assist with one resource finding other resources at runtime.

The `tags` attribute is used for attaching name/value pairs to resources. The name tag is used by AWS to have a display name for the resource. Every AWS resource has a computer-generated, user-unfriendly name with a long coded string and, of course, we humans need user-friendly names for things. The name tag is useful in that regard, and the AWS Management Console will respond by using this name in the dashboards.

In `variables.tf`, add this to support these resource declarations:

```
variable "enable_dns_support" { default = true }
variable "enable_dns_hostnames" { default = true }

variable "project_name" { default = "notes" }
variable "vpc_name" { default = "notes-vpc" }
variable "vpc_cidr" { default = "10.0.0.0/16" }
```

These values will be used throughout the project. For example, `var.project_name` will be widely used as the basis for creating name tags for deployed resources.

Add the following to `vpc.tf`:

```
data "aws_availability_zones" "available" {
    state = "available"
}
```

Where `resource` blocks declare something on the hosting platform (in this case, AWS), `data` blocks retrieve data from the hosting platform. In this case, we are retrieving a list of AZs for the currently selected region. We'll use this later when declaring certain resources.

Configuring the AWS gateway and subnet resources

Remember that a public subnet is associated with an internet gateway, and a private subnet is associated with a NAT gateway. The difference determines what type of internet access devices attached to each subnet have.

Create a file named `gw.tf` containing the following:

```
resource "aws_internet_gateway" "igw" {
    vpc_id = aws_vpc.notes.id
    tags = {
        Name = "${var.project_name}-IGW"
    }
}

resource "aws_eip" "gw" {
    vpc = true
    depends_on = [ aws_internet_gateway.igw ]
    tags = {
        Name = "${var.project_name}-EIP"
    }
}
```

```
resource "aws_nat_gateway" "gw" {
  subnet_id = aws_subnet.public1.id
  allocation_id = aws_eip.gw.id
  tags = {
    Name = "${var.project_name}-NAT"
  }
}
```

This declares the internet gateway and the NAT gateway. Remember that internet gateways are used with public subnets, and NAT gateways are used with private subnets.

An **Elastic IP (EIP)** resource is how a public internet IP address is assigned. Any device that is to be visible to the public must be on a public subnet and have an EIP. Because the NAT gateway faces the public internet, it must have an assigned public IP address and an EIP.

For the subnets, create a file named `subnets.tf` containing the following:

```
resource "aws_subnet" "public1" {
  vpc_id = aws_vpc.notes.id
  cidr_block = var.public1_cidr
  availability_zone = data.aws_availability_zones.available.names[0]
  tags = {
    Name = "${var.project_name}-net-public1"
  }
}

resource "aws_subnet" "private1" {
  vpc_id = aws_vpc.notes.id
  cidr_block = var.private1_cidr
  availability_zone = data.aws_availability_zones.available.names[0]
  tags = {
    Name = "${var.project_name}-net-private1"
  }
}
```

This declares the public and private subnets. Notice that these subnets are assigned to a specific AZ. It would be easy to expand this to support more subnets by adding subnets named `public2`, `public3`, `private2`, `private3`, and so on. If you do so, it would be helpful to spread these subnets across AZs. Deployment is recommended in multiple AZs so that if one AZ goes down, the application is still running in the AZ that's still up and running.

This notation with `[0]` is what it looks like—an array. The value, `data.aws_availability_zones.available.names`, is an array, and adding `[0]` does access the first element of that array, just as you'd expect. Arrays are just one of the data structures offered by Terraform.

Each subnet has its own CIDR (IP address range), and to support this, we need these CIDR assignments listed in `variables.tf`, as follows:

```
variable "vpc_cidr"      { default = "10.0.0.0/16" }
variable "public1_cidr"  { default = "10.0.1.0/24" }
variable "private1_cidr" { default = "10.0.3.0/24" }
```

These are the CIDRs corresponding to the resources declared earlier.

For these pieces to work together, we need appropriate routing tables to be configured. Create a file named `routing.tf` containing the following:

```
resource "aws_route" "route-public" {
  route_table_id = aws_vpc.notes.main_route_table_id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id = aws_internet_gateway.igw.id
}

resource "aws_route_table" "private" {
  vpc_id = aws_vpc.notes.id
  route {
    cidr_block = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.gw.id
  }
  tags = {
    Name = "${var.project_name}-rt-private"
  }
}

resource "aws_route_table_association" "public1" {
  subnet_id = aws_subnet.public1.id
  route_table_id = aws_vpc.notes.main_route_table_id
}

resource "aws_route_table_association" "private1" {
  subnet_id = aws_subnet.private1.id
  route_table_id = aws_route_table.private.id
}
```

To configure the routing table for the public subnets, we modify the routing table connected to the main routing table for the VPC. What we're doing here is adding a rule to that table, saying that public internet traffic is to be sent to the internet gateway. We also have a route table association declaring that the public subnet uses this route table.

For `aws_route_table.private`, the routing table for private subnets, the declaration says to send public internet traffic to the NAT gateway. In the route table associations, this table is used for the private subnet.

Earlier, we said the difference between a public and private subnet is whether public internet traffic is sent to the internet gateway or the NAT gateway. These declarations are how that's implemented.

In this section, we've declared the VPC, subnets, gateways, and routing tables—in other words, the infrastructure within which we'll deploy our Docker Swarm.

Before attaching the EC2 instances in which the swarm will live, let's deploy this to AWS and explore what gets set up.

Deploying the infrastructure to AWS using Terraform

We have now declared the bones of the AWS infrastructure we'll need. This is the VPC, the subnets, and routing tables. Let's deploy this to AWS and use the AWS console to explore what was created.

Earlier, we ran `terraform init` to initialize Terraform in our working directory. When we did so, it suggested that we run the following command:

```
$ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not
be
persisted to local or remote state storage.

data.aws_availability_zones.available: Refreshing state...

-----
--



An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
```

```
+ create
```

```
Terraform will perform the following actions:
```

```
# aws_eip.gw will be created
+ resource "aws_eip" "gw" {
    + allocation_id = (known after apply)
    + association_id = (known after apply)
    + customer_owned_ip = (known after apply)
    + domain = (known after apply)
    + id = (known after apply)
    + instance = (known after apply)
    + network_interface = (known after apply)
    + private_dns = (known after apply)
    + private_ip = (known after apply)
    + public_dns = (known after apply)
    + public_ip = (known after apply)
    + public_ipv4_pool = (known after apply)
    + tags = {
        + "Name" = "notes-EIP"
    }
    + vpc = true
}
...
...
```

This command scans the Terraform files in the current directory and first determines that everything has the correct syntax, that all the values are known, and so forth. If any problems are encountered, it stops right away with error messages such as the following:

```
Error: Reference to undeclared resource

on outputs.tf line 8, in output "subnet_public2_id":
  8: output "subnet_public2_id" { value = aws_subnet.public2.id }

A managed resource "aws_subnet" "public2" has not been declared in the
root
module.
```

Terraform's error messages are usually self-explanatory. In this case, the cause was a decision to use only one public and one private subnet. This code was left over from there being two of each. Therefore, this error referred to stale code that was easy to remove.

The other thing `terraform plan` does is construct a graph of all the declarations and print out a listing. This gives you an idea of what Terraform intends to deploy on to the chosen cloud platform. It is therefore your opportunity to examine the intended infrastructure and make sure it is what you want to use.

Once you're satisfied, run the following command:

```
$ terraform apply
data.aws_availability_zones.available: Refreshing state...

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:
...
Plan: 10 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes
...

Apply complete! Resources: 10 added, 0 changed, 0 destroyed.

Outputs:

aws_region = us-west-2
igw_id = igw-006eb101f8cb423d4
private1_cidr = 10.0.3.0/24
public1_cidr = 10.0.1.0/24
subnet_private1_id = subnet-0a9044daea298d1b2
subnet_public1_id = subnet-07e6f8ed6cc6f8397
vpc_arn = arn:aws:ec2:us-west-2:098106984154:vpc/vpc-074b2dfa7b353486f
vpc_cidr = 10.0.0.0/16
vpc_id = vpc-074b2dfa7b353486f
vpc_name = notes-vpc
```

With `terraform apply`, the report shows the difference between the actual deployed state and the desired state as reflected by the Terraform files. In this case, there is no deployed state, so therefore everything that is in the files will be deployed. In other cases, you might have deployed a system and have made a change, in which case Terraform will work out which changes have to be deployed based on the changes you've made. Once it calculates that, Terraform asks for permission to proceed. Finally, if we have said `yes`, it will proceed and launch the desired infrastructure.

Once finished, it tells you what happened. One result is the values of the `output` commands in the scripts. These are both printed on the console and are saved in the backend state file.

To see what was created, let's head to the AWS console and navigate to the VPC area, as follows:

The screenshot shows the AWS VPC console. At the top, there is a table with columns: Name, VPC ID, State, IPv4 CIDR, IPv6 CIDR, DHCP options set, and Main Route table. Two entries are listed:

Name	VPC ID	State	IPv4 CIDR	IPv6 CIDR	DHCP options set	Main Route table
notes-vpc	vpc-074b2dfa7b353486f	available	10.0.0.0/16	-	dopt-e0c05d98	rtb-0e6f5b7a7a4372080
default-vpc	vpc-b11a7ec9	available	172.31.0....	-	dopt-04aa3bde4d66153c6 ha...	rtb-49084932

Below the table, a message says "VPC: vpc-074b2dfa7b353486f". Underneath this, there are tabs: Description (selected), CIDR Blocks, Flow Logs, and Tags. The "Description" tab displays detailed information about the selected VPC:

Attribute	Value
VPC ID	vpc-074b2dfa7b353486f
State	available
IPv4 CIDR	10.0.0.0/16
IPv6 CIDR (Network Border Group)	-
DNS resolution	Enabled
DNS hostnames	Enabled
ClassicLink DNS Support	Disabled
Owner	098106984154
Tenancy	default
Default VPC	No
Classic link	Disabled
IPv6 Pool	-
Network ACL	acl-076a2be7127f1cccd3
DHCP options set	dopt-e0c05d98
Route table	rtb-0e6f5b7a7a4372080

Compare the **VPC ID** in the screenshot with the one shown in the Terraform output, and you'll see that they match. What's shown here is the main routing table, and the CIDR, and other settings we made in our scripts. Every AWS account has a default VPC that's presumably meant for experiments. It is a better form to create a VPC for each project so that resources for each project are separate from other projects.

The sidebar contains links for further dashboards for subnets, route tables, and other things, and an example dashboard can be seen in the following screenshot:

The screenshot shows the AWS NAT Gateway dashboard. At the top, there is a table with columns: Name, NAT Gateway ID, Status, Status Message, Elastic IP Address, Private IP Address, Network Interface, and VPC. One row is visible for 'notes-NAT'. Below the table, a message says 'NAT Gateway: nat-08a6af5cd15e558fc'. There are three tabs at the bottom: Details (selected), Monitoring, and Tags.

Name	NAT Gateway ID	Status	Status Message	Elastic IP Address	Private IP Address	Network Interface	VPC
notes-NAT	nat-08a6af5cd15e558fc...	available	-	54.71.140.194	10.0.1.169	eni-0e0f480b3246e1585...	vpc-074b2dfa7b353486f...

Details **Monitoring** **Tags**

NAT Gateway ID: nat-08a6af5cd15e558fc
Status Message: -
Private IP Address: 10.0.1.169
VPC: vpc-074b2dfa7b353486f | notes-vpc
Created: May 27, 2020 at 5:43:55 PM UTC-7

Status: available
Elastic IP Address: 54.71.140.194
Network Interface ID: eni-0e0f480b3246e1585
Subnet: subnet-07e6fbed6cc6ff8397 | notes-net-public1
Deleted: -

For example, this is the NAT gateway dashboard showing the one created for this project.

Another way to explore is with the AWS CLI tool. Just because we have Terraform doesn't mean we are prevented from using the CLI. Have a look at the following code block:

```
$ aws ec2 describe-vpcs --vpc-ids vpc-074b2dfa7b353486f
{
  "Vpcs": [ {
    "CidrBlock": "10.0.0.0/16",
    "DhcpOptionsId": "dopt-e0c05d98",
    "State": "available",
    "VpcId": "vpc-074b2dfa7b353486f",
    "OwnerId": "098106984154",
    "InstanceTenancy": "default",
    "CidrBlockAssociationSet": [ {
      "AssociationId": "vpc-cidr-assoc-0f827bcc4fbb9fd62",
      "CidrBlock": "10.0.0.0/16",
      "CidrBlockState": {
        "State": "associated"
      }
    }],
    "IsDefault": false,
    "Tags": [ {
      "Key": "Name",
      "Value": "notes-vpc"
    }]
  }]
}
```

This lists the parameters for the VPC that was created.

Remember to either configure the `AWS_PROFILE` environment variable or use `--profile` on the command line.

To list data on the subnets, run the following command:

```
$ aws ec2 describe-subnets --filters "Name=vpc-id,Values=vpc-074b2dfa7b353486f"
{
  "Subnets": [
    { ... },
    { ... }
  ]
}
```

To focus on the subnets for a given VPC, we use the `--filters` option, passing in the filter named `vpc-id` and the VPC ID for which to filter.

Documentation for the AWS CLI can be found at <https://docs.aws.amazon.com/cli/latest/reference/index.html>.



For documentation relating to the EC2 sub-commands, refer to <https://docs.aws.amazon.com/cli/latest/reference/ec2/index.html>.

The AWS CLI tool has an extensive list of sub-commands and options. These are enough to almost guarantee getting lost, so read carefully.

In this section, we learned how to use Terraform to set up the VPC and related infrastructure resources, and we also learned how to navigate both the AWS console and the AWS CLI to explore what had been created.

Our next step is to set up an initial Docker Swarm cluster by deploying an EC2 instance to AWS.

Setting up a Docker Swarm cluster on AWS EC2

What we have set up is essentially a blank slate. AWS has a long list of offerings that could be deployed to the VPC that we've created. What we're looking to do in this section is to set up a single EC2 instance to install Docker, and set up a single-node Docker Swarm cluster. We'll use this to familiarize ourselves with Docker Swarm. In the remainder of the chapter, we'll build more servers to create a larger swarm cluster for full deployment of Notes.

A Docker Swarm cluster is simply a group of servers running Docker that have been joined together into a common pool. The code for the Docker Swarm orchestrator is bundled with the Docker Engine server but it is disabled by default. To create a swarm, we simply enable swarm mode by running `docker swarm init` and then run a `docker swarm join` command on each system we want to be part of the cluster. From there, the Docker Swarm code automatically takes care of a long list of tasks. The features for Docker Swarm include the following:

- **Horizontal scaling:** When deploying a Docker service to a swarm, you tell it the desired number of instances as well as the memory and CPU requirements. The swarm takes that and computes the best distribution of tasks to nodes in the swarm.
- **Maintaining the desired state:** From the services deployed to a swarm, the swarm calculates the desired state of the system and tracks its current actual state. Suppose one of the nodes crashes—the swarm will then readjust the running tasks to replace the ones that vaporized because of the crashed server.
- **Multi-host networking:** The overlay network driver automatically distributes network connections across the network of machines in the swarm.
- **Secure by default:** Swarm mode uses strong **Transport Layer Security (TLS)** encryption for all communication between nodes.
- **Rolling updates:** You can deploy an update to a service in such a manner where the swarm intelligently brings down existing service containers, replacing them with updated newer containers.



For an overview of Docker Swarm, refer to <https://docs.docker.com/engine/swarm/>.

We will use this section to not only learn how to set up a Docker Swarm but to also learn something about how Docker orchestration works.

To get started, we'll set up a single-node swarm on a single EC2 instance in order to learn some basics, before we move on to deploying a multi-node swarm and deploying the full Notes stack.

Deploying a single-node Docker Swarm on a single EC2 instance

For a quick introduction to Docker Swarm, let's start by installing Docker on a single EC2 node. We can kick the tires by trying a few commands and exploring the resulting system.

This will involve deploying Ubuntu 20.04 on an EC2 instance, configuring it to have the latest Docker Engine, and initializing swarm mode.

Adding an EC2 instance and configuring Docker

To launch an EC2 instance, we must first select which operating system to install. There are thousands of operating system configurations available. Each of these configurations is identified by an **AMI** code, where AMI stands for **Amazon Machine Image**.

To find your desired AMI, navigate to the EC2 dashboard on the AWS console. Then, click on the **Launch Instance** button, which starts a wizard-like interface to launch an instance. You can, if you like, go through the whole wizard since that is one way to learn about EC2 instances. We can search the AMIs via the first page of that wizard, where there is a search box.

For this exercise, we will use Ubuntu 20.04, so enter `Ubuntu` and then scroll down to find the correct version, as illustrated in the following screenshot:



This is what the desired entry looks like. The AMI code starts with `ami-` and we see one version for x86 CPUs, and another for **ARM** (previously **Advanced RISC Machine**). ARM processors, by the way, are not just for your cell phone but are also used in servers. There is no need to launch an EC2 instance from here since we will instead do so with Terraform.

Another attribute to select is the instance size. AWS supports a long list of sizes that relate to the amount of memory, CPU cores, and disk space. For a chart of the available instance types, click on the **Select** button to proceed to the second page of the wizard, which shows a table of instance types and their attributes. For this exercise, we will use the `t2.micro` instance type because it is eligible for the free tier.

Create a file named `ec2-public.tf` containing the following:

```
resource "aws_instance" "public" {
    ami = var.ami_id
    instance_type = var.instance_type
    subnet_id = aws_subnet.public1.id
    key_name = var.key_pair
    vpc_security_group_ids = [ aws_security_group.ec2-public-sg.id ]
    associate_public_ip_address = true
    tags = {
        Name = "${var.project_name}-ec2-public"
    }
    depends_on = [ aws_vpc.notes, aws_internet_gateway.igw ]
    user_data = join("\n", [
        "#!/bin/sh",
        file("sh/docker_install.sh"),
        "docker swarm init",
        "sudo hostname ${var.project_name}-public"
    ])
}
```

In the Terraform AWS provider, the resource name for EC2 instances is `aws_instance`. Since this instance is attached to our public subnet, we'll call it `aws_instance.public`. Because it is a public EC2 instance, the `associate_public_ip_address` attribute is set to `true`.

The attributes include the AMI ID, the instance type, the ID for the subnet, and more. The `key_name` attribute refers to the name of an SSH key we'll use to log in to the EC2 instance. We'll discuss these key pairs later. The `vpc_security_group_ids` attribute is a reference to a security group we'll apply to the EC2 instance. The `depends_on` attribute causes Terraform to wait for the creation of the resources named in the array. The `user_data` attribute is a shell script that is executed inside the instance once it is created.

For the AMI, instance type, and key-pair data, add these entries to `variables.tf`, as follows:

```
variable "ami_id"          { default = "ami-09dd2e08d601bff67" }
variable "instance_type" { default = "t2.micro" }
variable "key_pair"       { default = "notes-app-key-pair" }
```

The AMI ID shown here is specifically for Ubuntu 20.04 in `us-west-2`. There will be other AMI IDs in other regions. The `key_pair` name shown here should be the key-pair name you selected when creating your key pair earlier.

It is not necessary to add the key-pair file to this directory, nor to reference the file you downloaded in these scripts. Instead, you simply give the name of the key pair. In our example, we named it `notes-app-key-pair`, and downloaded `notes-app-key-pair.pem`.

The `user_data` feature is very useful since it lets us customize an instance after creation. We're using this to automate the Docker setup on the instances. This field is to receive a string containing a shell script that will execute once the instance is launched. Rather than insert that script inline with the Terraform code, we have created a set of files that are shell script snippets. The Terraform `file` function reads the named file, returning it as a string. The Terraform `join` function takes an array of strings, concatenating them together with the delimiter character in between. Between the two we construct a shell script. The shell script first installs Docker Engine, then initializes Docker Swarm mode, and finally changes the hostname to help us remember that this is the public EC2 instance.

Create a directory named `sh` in which we'll create shell scripts, and in that directory create a file named `docker_install.sh`. To this file, add the following:

```
sudo apt-get update
sudo apt-get upgrade -y

sudo apt-get -y install apt-transport-https \
    ca-certificates curl gnupg-agent software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
    | sudo apt-key add -

sudo apt-key fingerprint 0EBFCD88

sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu
    $(lsb_release -cs) stable"
```

```
sudo apt-get update
sudo apt-get upgrade -y
sudo apt-get install -y docker-ce docker-ce-cli containerd.io
sudo groupadd docker
sudo usermod -aG docker ubuntu
sudo systemctl enable docker
```

This script is derived from the official instructions for installing Docker Engine **Community Edition (CE)** on Ubuntu. The first portion is support for `apt-get` to download packages from HTTPS repositories. It then configures the Docker package repository into Ubuntu, after which it installs Docker and related tools. Finally, it ensures that the `docker` group is created and ensures that the `ubuntu` user ID is a member of that group. The Ubuntu AMI defaults to this user ID, `ubuntu`, to be the one used by the EC2 administrator.

For this EC2 instance, we also run `docker swarm init` to initialize the Docker Swarm. For other EC2 instances, we do not run this command. The method used for initializing the `user_data` attribute lets us easily have a custom configuration script for each EC2 instance. For the other instances, we'll only run `docker_install.sh`, whereas for this instance, we'll also initialize the swarm.

Back in `ec2-public.tf`, we have two more things to do, and then we can launch the EC2 instance. Have a look at the following code block:

```
resource "aws_security_group" "ec2-public-sg" {
  name = "${var.project_name}-public-security-group"
  description = "allow inbound access to the EC2 instance"
  vpc_id = aws_vpc.notes.id

  ingress {
    protocol = "TCP"
    from_port = 22
    to_port = 22
    cidr_blocks = [ "0.0.0.0/0" ]
  }

  ingress {
    protocol = "TCP"
    from_port = 80
    to_port = 80
    cidr_blocks = [ "0.0.0.0/0" ]
  }

  egress {
    protocol = "-1"
    from_port = 0
```

```
    to_port = 0
    cidr_blocks = [ "0.0.0.0/0" ]
}
}
```

This is the security group declaration for the public EC2 instance. Remember that a security group describes the rules of a firewall that is attached to many kinds of AWS objects. This security group was already referenced in declaring `aws_instance.public`.

The main feature of security groups is the `ingress` and `egress` rules. As the words imply, `ingress` rules describe the network traffic allowed to enter the resource, and `egress` rules describe what's allowed to be sent by the resource. If you have to look up those words in a dictionary, you're not alone.

We have two `ingress` rules, and the first allows traffic on port 22, which covers SSH traffic. The second allows traffic on port 80, covering HTTP. We'll add more Docker rules later when they're needed.

The `egress` rule allows the EC2 instance to send any traffic to any machine on the internet.

These `ingress` rules are obviously very strict and limit the attack surface any miscreants can exploit.

The final task is to add these output declarations to `ec2-public.tf`, as follows:

```
output "ec2-public-arn" { value = aws_instance.public.arn }
output "ec2-public-dns" { value = aws_instance.public.public_dns }
output "ec2-public-ip" { value = aws_instance.public.public_ip }
output "ec2-private-dns" { value = aws_instance.public.private_dns }
output "ec2-private-ip" { value = aws_instance.public.private_ip }
```

This will let us know the public IP address and public DNS name. If we're interested, the outputs also tell us the private IP address and DNS name.

Launching the EC2 instance on AWS

We have added to the Terraform declarations for creating an EC2 instance.

We're now ready to deploy this to AWS and see what we can do with it. We already know what to do, so let's run the following command:

```
$ terraform plan  
...  
Plan: 2 to add, 0 to change, 0 to destroy.
```

If the VPC infrastructure were already running, you would get output similar to this. The addition is two new objects, `aws_instance.public` and `aws_security_group.ec2-public-sg`. This looks good, so we proceed to deployment, as follows:

```
$ terraform apply  
...  
Plan: 2 to add, 0 to change, 0 to destroy.  
  
Do you want to perform these actions?  
Terraform will perform the actions described above.  
Only 'yes' will be accepted to approve.  
  
Enter a value: yes  
...  
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.  
  
Outputs:  
  
aws_region = us-west-2  
ec2-private-dns = ip-10-0-1-55.us-west-2.compute.internal  
ec2-private-ip = 10.0.1.55  
ec2-public-arn = arn:aws:ec2:us-  
west-2:098106984154:instance/i-0046b28d65a4f555d  
ec2-public-dns = ec2-54-213-6-249.us-west-2.compute.amazonaws.com  
ec2-public-ip = 54.213.6.249  
igw_id = igw-006eb101f8cb423d4  
private1_cidr = 10.0.3.0/24  
public1_cidr = 10.0.1.0/24  
subnet_private1_id = subnet-0a9044daea298d1b2  
subnet_public1_id = subnet-07e6f8ed6cc6f8397  
vpc_arn = arn:aws:ec2:us-west-2:098106984154:vpc/vpc-074b2dfa7b353486f  
vpc_cidr = 10.0.0.0/16  
vpc_id = vpc-074b2dfa7b353486f  
vpc_name = notes-vpc
```

This built our EC2 instance, and we have the IP address and domain name. Because the initialization script will have required a couple of minutes to run, it is good to wait for a short time before proceeding to test the system.

The `ec2-public-ip` value is the public IP address for the EC2 instance. In the following examples, we will put the text `PUBLIC-IP-ADDRESS`, and you must of course substitute the IP address your EC2 instance is assigned.

We can log in to the EC2 instance like so:

```
$ ssh -i ~/Downloads/notes-app-key-pair.pem ubuntu@PUBLIC-IP-ADDRESS
The authenticity of host '54.213.6.249 (54.213.6.249)' can't be
established.
ECDSA key fingerprint is
SHA256:DOGsiDjWZ6rkj1+AiMcqqy/naAku5b4VJUgZqtlwPg8.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.213.6.249' (ECDSA) to the list of known
hosts.
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.4.0-1009-aws x86_64)

...
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@notes-public:~$ hostname
notes-public
```

On a Linux or macOS system where we're using SSH, the command is as shown here. The `-i` option lets us specify the **Privacy Enhanced Mail (PEM)** file that was provided by AWS for the key pair. If on Windows using PuTTY, you'd instead tell it which **PuTTY Private Key (PPK)** file to use, and the connection parameters will otherwise be similar to this.

This lands us at the command-line prompt of the EC2 instance. We see that it is Ubuntu 20.04, and the hostname is set to `notes-public`, as reflected in Command Prompt and the output of the `hostname` command. This means that our initialization script ran because the hostname was the last configuration task it performed.

Handling the AWS EC2 key-pair file

Earlier, we said to safely store the key-pair file somewhere on your computer. In the previous section, we showed how to use the PEM file with SSH to log in to the EC2 instance. Namely, we use the PEM file like so:

```
$ ssh -i /path/to/key-pair.pem USER-ID@HOST-IP
```

It can be inconvenient to remember to add the `-i` flag every time we use SSH. To avoid having to use this option, run this command:

```
$ ssh-add /path/to/key-pair.pem
```

As the command name implies, this adds the authentication file to SSH. This has to be rerun on every reboot of the computer, but it conveniently lets us access EC2 instances without remembering to specify this option.

Testing the initial Docker Swarm

We have an EC2 instance and it should already be configured with Docker, and we can easily verify that this is the case as follows:

```
ubuntu@notes-public:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
...
```

The setup script was also supposed to have initialized this EC2 instance as a Docker Swarm node, and the following command verifies whether that happened:

```
ubuntu@notes-public:~$ docker info
...
Swarm: active
  NodeID: qfb1ljmw2fgp4ij18klowr8dp
  Is Manager: true
  ClusterID: 14p4sdfsdyoa8el0v9cqirm23
...
```

The `docker info` command, as the name implies, prints out a lot of information about the current Docker instance. In this case, the output includes verification that it is in Docker Swarm mode and that this is a Docker Swarm manager instance.

Let's try a couple of swarm commands, as follows:

```
ubuntu@notes-public:~$ docker node ls
ID                      HOSTNAME        STATUS  AVAILABILITY  MANAGER
STATUS  ENGINE VERSION
qfb11jmw2fgp4ij18klowr8dp *  notes-public  Ready  Active  Leader  19.03.9

ubuntu@notes-public:~$ docker service ls
ID NAME MODE REPLICAS IMAGE PORTS
```

The `docker node` command is for managing the nodes in a swarm. In this case, there is only one node—this one, and it is shown as not only a manager but as the swarm leader. It's easy to be the leader when you're the only node in the cluster, it seems.

The `docker service` command is for managing the services deployed in the swarm. In this context, a service is roughly the same as an entry in the `services` section of a Docker compose file. In other words, a service is not the running container but is an object describing the configuration for launching one or more instances of a given container.

To see what this means, let's start an `nginx` service, as follows:

```
ubuntu@notes-public:~$ docker service create --name nginx --replicas 1
-p 80:80 nginx
ephvpfgjwxgdwx7ab87e7nc9e
overall progress: 1 out of 1 tasks
1/1: running
verify: Service converged

ubuntu@notes-public:~$ docker service ls
ID      NAME      MODE      REPLICAS  IMAGE      PORTS
ephvpfgjwxgd nginx replicated 1/1      nginx:latest *:80->80/tcp

ubuntu@notes-public:~$ docker service ps nginx
ID      NAME      IMAGE      NODE      DESIRED STATE  CURRENT
STATE  ERROR  PORTS
ag8b45t69am1 nginx.1  nginx:latest  notes-public  Running  Running  15
seconds ago
```

We started one service using the `nginx` image. We said to deploy one replica and to expose port 80. We chose the `nginx` image because it has a simple default HTML file that we can easily view, as illustrated in the following screenshot:



Simply paste the IP address of the EC2 instance into the browser location bar, and we're greeted with that default HTML.

We also see by using `docker node ls` and `docker service ps` that there is one instance of the service. Since this is a swarm, let's increase the number of `nginx` instances, as follows:

```
ubuntu@notes-public:~$ docker service update --replicas 3 nginx
nginx
overall progress: 3 out of 3 tasks
1/3: running
2/3: running
3/3: running
verify: Service converged

ubuntu@notes-public:~$ docker service ls
ID          NAME      MODE      REPLICAS  IMAGE          PORTS
ephvpgjwxgd nginx     replicated 3/3      nginx:latest *:80->80/tcp

ubuntu@notes-public:~$ docker service ps nginx
ID          NAME      IMAGE      NODE      DESIRED STATE  CURRENT STATE
STATE ERROR PORTS
ag8b45t69am1 nginx.1  nginx:latest  notes-public  Running  Running  9
minutes ago
ojvbs4n2iriy nginx.2  nginx:latest  notes-public  Running  Running  13
```

```
seconds ago
fqwwk8c4tqck nginx.3 nginx:latest notes-public Running Running 13
seconds ago
```

Once a service is deployed, we can modify the deployment using the `docker service update` command. In this case, we told it to increase the number of instances using the `--replicas` option, and we now have three instances of the `nginx` container all running on the `notes-public` node.

We can also run the normal `docker ps` command to see the actual containers, as illustrated in the following code block:

```
ubuntu@notes-public:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES
6dc274c30fea nginx:latest "nginx -g 'daemon of..." About a minute ago Up
About a minute 80/tcp nginx.2.ojvbs4n2iriyjifeh01jlyvh
4b51455fb2bf nginx:latest "nginx -g 'daemon of..." About a minute ago Up
About a minute 80/tcp nginx.3.fqwwk8c4tqckspcrrzbs0qyii
e7ed31f9471f nginx:latest "nginx -g 'daemon of..." 10 minutes ago Up 10
minutes 80/tcp nginx.1.ag8b45t69am1gzh0b65gfnq14
```

This verifies that the `nginx` service with three replicas is actually three `nginx` containers.

In this section, we were able to launch an EC2 instance and set up a single-node Docker swarm in which we launched a service, which gave us the opportunity to familiarize ourselves with what this can do.

While we're here, there is another thing to learn—namely, how to set up the remote control of Docker hosts.

Setting up remote control access to a Docker Swarm hosted on EC2

A feature that's not well documented in Docker is the ability to control Docker nodes remotely. This will let us, from our laptop, run Docker commands on a server. By extension, this means that we will be able to manage the Docker Swarm from our laptop.

One method for remotely controlling a Docker instance is to expose the Docker **Transmission Control Protocol (TCP)** port. Be aware that miscreants are known to scan an internet infrastructure for Docker ports to hijack. The following technique does not expose the Docker port but instead uses SSH.

The following setup is for Linux and macOS, relying on features of SSH. To do this on Windows would rely on installing OpenSSH. From October 2018, OpenSSH became available for Windows, and the following commands may work in PowerShell (failing that, you can run these commands from a Multipass or **Windows Subsystem for Linux (WSL) 2** instance on Windows):

```
ubuntu@notes-public:~$ logout  
Connection to PUBLIC-IP-ADDRESS closed.
```

Exit the shell on the EC2 instance so that you're at the command line on your laptop.

Run the following command:

```
$ ssh-add ~/Downloads/notes-app-key-pair.pem  
Identity added: /Users/david/Downloads/notes-app-key-pair.pem  
(/Users/david/Downloads/notes-app-key-pair.pem)
```

We discussed this command earlier, noting that it lets us log in to EC2 instances without having to use the `-i` option to specify the PEM file. This is more than a simple convenience when it comes to remotely accessing Docker hosts. The following steps are dependent on having added the PEM file to SSH, as shown here.

To verify you've done this correctly, use this command:

```
$ ssh ubuntu@PUBLIC-IP-ADDRESS
```

Normally with an EC2 instance, we would use the `-i` option, as shown earlier. But after running `ssh-add`, the `-i` option is no longer required.

That enables us to create the following environment variable:

```
$ export DOCKER_HOST=ssh://ubuntu@PUBLIC-IP-ADDRESS

$ docker service ls
ID          NAME    MODE      REPLICAS  IMAGE           PORTS
ephvpfgjwxgd  nginx  replicated  3/3      nginx:latest *:80->80/tcp
```

The `DOCKER_HOST` environment variable enables the remote control of Docker hosts. It relies on a passwordless SSH login to the remote host. Once you have that, it's simply a matter of setting the environment variable and you've got remote control of the Docker host, and in this case, because the host is a swarm manager, a remote swarm.

But this gets even better by using the Docker context feature. A *context* is a configuration required to access a remote node or swarm. Have a look at the following code snippet:

```
$ unset DOCKER_HOST
```

We begin by deleting the environment variable because we'll replace it with something better, as follows:

```
$ docker context create ec2 --docker host=ssh://ubuntu@PUBLIC-IP-
ADDRESS
ec2
Successfully created context "ec2"

$ docker --context ec2 service ls
ID          NAME    MODE      REPLICAS  IMAGE           PORTS
ephvpfgjwxgd  nginx  replicated  3/3      nginx:latest *:80->80/tcp

$ docker context use ec2
ec2
Current context is now "ec2"

$ docker service ls
ID          NAME    MODE      REPLICAS  IMAGE           PORTS
ephvpfgjwxgd  nginx  replicated  3/3      nginx:latest *:80->80/tcp
```

We create a context using `docker context create`, specifying the same SSH URL we used in the `DOCKER_HOST` variable. We can then use it either with the `--context` option or by using `docker context use` to switch between contexts.

With this feature, we can easily maintain configurations for multiple remote servers and switch between them with a simple command.

For example, the Docker instance on our laptop is the *default* context. Therefore, we might find ourselves doing this:

```
$ docker context use default
... run docker commands against Docker on the laptop
$ docker context use ec2
... run docker commands against Docker on the AWS EC2 machines
```

There are times when we must be cognizant of which is the current Docker context and when to use which context. This will be useful in the next section when we learn how to push the images to AWS ECR.

We've learned a lot in this section, so before heading to the next task, let's clean up our AWS infrastructure. There's no need to keep this EC2 instance running since we used it solely for a quick familiarization tour. We can easily delete this instance while leaving the rest of the infrastructure configured. The most effective way to do so is by renaming `ec2-public.tf` to `ec2-public.tf-disable`, and to rerun `terraform apply`, as illustrated in the following code block:

```
$ mv ec2-public.tf ec2-public.tf-disable
$ terraform apply
...
Plan: 0 to add, 0 to change, 2 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes
...
```

The effect of changing the name of one of the Terraform files is that Terraform will not scan those files for objects to deploy. Therefore, when Terraform maps out the state we want Terraform to deploy, it will notice that the deployed EC2 instance and security group are not listed in the local files, and it will, therefore, destroy those objects. In other words, this lets us undeploy some infrastructure with very little fuss.

This tactic can be useful for minimizing costs by turning off unneeded facilities. You can easily redeploy the EC2 instances by renaming the file back to `ec2-public.tf` and rerunning `terraform apply`.

In this section, we familiarized ourselves with Docker Swarm by deploying a single-node swarm on an EC2 instance on AWS. We first added suitable declarations to our Terraform files. We then deployed the EC2 instance on AWS. Following deployment, we set about verifying that, indeed, Docker Swarm was already installed and initialized on the server and that we could easily deploy Docker services on the swarm. We then learned how to set up remote control of the swarm from our laptop.

Taken together, this proved that we can easily deploy Docker-based services to EC2 instances on AWS. In the next section, let's continue preparing for a production-ready deployment by setting up a build process to push Docker images to image repositories.

Setting up ECR repositories for Notes Docker images

We have created Docker images to encapsulate the services making up the Notes application. So far, we've used those images to instantiate Docker containers on our laptop. To deploy containers on the AWS infrastructure will require the images to be hosted in a Docker image repository.

This requires a build procedure by which the `svc-notes` and `svc-userauth` images are correctly pushed to the container repository on the AWS infrastructure. We will go over the commands required and create a few shell scripts to record those commands.

A site such as Docker Hub is what's known as a Docker Registry. Registries are web services that store Docker images by hosting Docker image repositories. When we used the `redis` or `mysql/mysql-server` images earlier, we were using Docker image repositories located on the Docker Hub Registry.

The AWS team offers a Docker image registry, ECR. An ECR instance is available for each account in each AWS region. All we have to do is log in to the registry, create repositories, and push images to the repositories.



It is extremely important to run commands in this section in the default Docker context on your laptop. The reason is that Docker builds must not happen on the Swarm host but on some other host, such as your laptop.

Because it is important to not run Docker build commands on the Swarm infrastructure, execute this command:

```
$ docker context use default
```

This command switches the Docker context to the local system.

To hold the scripts and other files related to managing AWS ECR repositories, create a directory named `ecr` as a sibling to `notes`, `users`, and `terraform-swarm`.

There are several commands required for a build process to create Docker images, tag them, and push them to a remote repository. To simplify things, let's create a few shell scripts, as well as PowerShell scripts, to record those commands.

The first task is to connect with the AWS ECR service. To this end, create a file named `login.sh` containing the following:

```
aws ecr get-login-password --profile $AWS_PROFILE --region $AWS_REGION
\
| docker login --username AWS \
--password-stdin $AWS_USER.dkr.ecr.$AWS_REGION.amazonaws.com
```

This command, and others, are available in the ECR dashboard. If you navigate to that dashboard and then create a repository there, a button labeled **View Push Command** is available. This and other useful commands are listed there, but we have substituted a few variable names to make this configurable.

If you are instead using Windows PowerShell, AWS recommends the following:

```
(Get-ECRLoginCommand).Password | docker login --username AWS --
password-stdin ACCOUNT-ID.dkr.ecr.REGION-NAME.amazonaws.com
```

This relies on the AWS Tools for PowerShell package (see <https://aws.amazon.com/powershell/>), which appears to offer some powerful tools that are useful with AWS services. In testing, however, this command was not found to work very well.

Instead, the following command was found to work much better, which you can put in a file named `login.ps1`:

```
aws ecr get-login-password --region %AWS_REGION% | docker login --  
username AWS --password-stdin  
%AWS_USER%.dkr.ecr.%AWS_REGION%.amazonaws.com
```

This is the same command as is used for Unix-like systems, but with Windows-style references to environment variables.



You may wish to explore the `cross-var` package, since it can convert Unix-style environment variable references to Windows. For the documentation, refer to <https://www.npmjs.com/package/cross-var>.

Several environment variables are being used, but just what are those variables being used and how do we set them?

Using environment variables for AWS CLI commands

Look carefully and you will see that some environment variables are being used. The AWS CLI commands know about those environment variables and will use them instead of command-line options. The environment variables we're using are the following:

- `AWS_PROFILE`: The AWS profile to use with this project.
- `AWS_REGION`: The AWS region to deploy the project to.
- `AWS_USER`: The numeric user ID for the account being used. This ID is available on the IAM dashboard page for the account.



The AWS CLI recognizes some of these environment variables, and others. For further details, refer to <https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-envvars.html>.

The AWS command-line tools will use those environment variables in place of the command-line options. Earlier, we discussed using the `AWS_PROFILE` variable instead of the `--profile` option. The same holds true for other command-line options.

This means that we need an easy way to set those variables. These Bash commands can be recorded in a shell script like this, which you could store as `env-us-west-2`:

```
export AWS_REGION=us-west-2
export AWS_PROFILE=notes-app
export AWS_USER=09E1X6A8MPLE
```

This script is, of course, following the syntax of the Bash shell. For other command environments, you must transliterate it appropriately. To set these variables in the Bash shell, run the following command:

```
$ chmod +x env-us-west-2
$ ./env-us-west-2
```

For other command environments, again transliterate appropriately. For example, in Windows and in PowerShell, the variables can be set with these commands:

```
$env:AWS_USER = "09E1X6A8MPLE"
$env:AWS_PROFILE = "notes-app"
$env:AWS_REGION = "us-west-2"
```

These should be the same values, just in a syntax recognized by Windows.

We have defined the environment variables being used. Let's now get back to defining the process to build Docker images and push them to the ECR.

Defining a process to build Docker images and push them to the AWS ECR

We were exploring a build procedure for pushing Docker containers to ECR repositories until we started talking about environment variables. Let's return to the task at hand, which is to easily build Docker images, create ECR repositories, and push the images to the ECR.

As mentioned at the beginning of this section, make sure to switch to the *default* Docker context. We must do so because it is a policy with Docker Swarm to not use the swarm hosts for building Docker images.

To build the images, let's add a file named `build.sh` containing the following:

```
( cd ../notes && npm run docker-build )
( cd ../users && npm run docker-build )
```

This handles running `docker build` commands for both the Notes and user authentication services. It is expected to be executed in the `ecr` directory and takes care of executing commands in both the `notes` and `users` directories.

Let's now create and delete a pair of registries to hold our images. We have two images to upload to the ECR, and therefore we create two registries.

Create a file named `create.sh` containing the following:

```
aws ecr create-repository --repository-name svc-notes --image-scanning-configuration scanOnPush=true
aws ecr create-repository --repository-name svc-userauth --image-scanning-configuration scanOnPush=true
```

Also, create a companion file named `delete.sh` containing the following:

```
aws ecr delete-repository --force --repository-name svc-notes
aws ecr delete-repository --force --repository-name svc-userauth
```

Between these scripts, we can create and delete the ECR repositories for our Docker images. These scripts are directly usable on Windows; simply change the filenames to `create.ps1` and `delete.ps1`.

In `aws ecr delete-repository`, the `--force` option means to delete the repositories even if they contain images.

With the scripts we've written so far, they are executed in the following order:

```
$ sh login.sh
Login Succeeded
$ sh create.sh
{
  "repository": {
    "repositoryArn": "arn:aws:ecr:us-REGION-2:09E1X6A8MPLE:repository/svc-notes",
    "registryId": "098106984154",
    "repositoryName": "svc-notes",
    "repositoryUri": "09E1X6A8MPLE.dkr.ecr.us-REGION-2.amazonaws.com/svc-notes",
    "createdAt": "2020-06-07T12:34:03-07:00",
    "imageTagMutability": "MUTABLE",
    "imageScanningConfiguration": {
      "scanOnPush": true
    }
  }
}
```

```
"repository": {
    "repositoryArn": "arn:aws:ecr:us-
        REGION-2:09E1X6A8MPLE:repository/svc-userauth",
    "registryId": "098106984154",
    "repositoryName": "svc-userauth",
    "repositoryUri": "09E1X6A8MPLE.dkr.ecr.us-
        REGION-2.amazonaws.com/svc-userauth",
    "createdAt": "2020-06-07T12:34:05-07:00",
    "imageTagMutability": "MUTABLE",
    "imageScanningConfiguration": {
        "scanOnPush": true
    }
}
```

The `aws ecr create-repository` command outputs these descriptors for the image repositories. The important piece of data to note is the `repositoryUri` value. This will be used later in the Docker stack file to name the image to be retrieved.

The `create.sh` script only needs to be executed once.

Beyond creating the repositories, the workflow is as follows:

- Build the images, for which we've already created a script named `build.sh`.
- Tag the images with the ECR repository **Uniform Resource Identifier (URI)**.
- Push the images to the ECR repository.

For the latter two steps, we still have some scripts to create.

Create a file named `tag.sh` containing the following:

```
docker tag svc-notes:latest
$AWS_USER.dkr.ecr.$AWS_REGION.amazonaws.com/svc-notes:latest
docker tag svc-userauth:latest
$AWS_USER.dkr.ecr.$AWS_REGION.amazonaws.com/svc-userauth:latest
```

The `docker tag` command we have here takes `svc-notes:latest`, or `svc-userauth:latest`, and adds what's called a target image to the local image storage area. The target image name we've used is the same as what will be stored in the ECR repository.

For Windows, you should create a file named `tag.ps1` using the same commands, but with Windows-style environment variable references.

Then, create a file named `push.sh` containing the following:

```
docker push $AWS_USER.dkr.ecr.$AWS_REGION.amazonaws.com/svc-  
notes:latest  
docker push $AWS_USER.dkr.ecr.$AWS_REGION.amazonaws.com/svc-  
userauth:latest
```

The `docker push` command causes the target image to be sent to the ECR repository. And again, for Windows, create a file named `push.ps1` containing the same commands but with Windows-style environment variable references.

In both the `tag` and `push` scripts, we are using the repository URI value, but have plugged in the two environment variables. This will make it generalized in case we deploy Notes to another AWS region.

We have the workflow implemented as scripts, so let's see now how it is run, as follows:

```
$ sh -x build.sh  
+ cd ./notes  
+ npm run docker-build  
  
> notes@0.0.0 docker-build /Users/David/Chapter12/notes  
> docker build -t svc-notes .  
  
Sending build context to Docker daemon 84.12MB  
Step 1/25 : FROM node:14  
--> a5a6a9c32877  
Step 2/25 : RUN apt-get update -y && apt-get -y install curl python  
build-essential git ca-certificates  
--> Using cache  
--> 7cf57f90c8b8  
Step 3/25 : ENV DEBUG="notes:*,messages:*"  
--> Using cache  
--> 291652c87cce  
...  
Successfully built e2f6ec294016  
Successfully tagged svc-notes:latest  
+ cd ../users  
+ npm run docker-build  
  
> user-auth-server@1.0.0 docker-build /Users/David/Chapter12/users  
> docker build -t svc-userauth .
```

```
Sending build context to Docker daemon 11.14MB
...
Successfully built 294b9a83ada3
Successfully tagged svc-userauth:latest
```

This builds the Docker images. When we run `docker build`, it stores the built image in an area on our laptop where Docker maintains images. We can inspect that area using the `docker images` command, like this:

```
$ docker images svc-userauth
REPOSITORY      TAG      IMAGE ID      CREATED      SIZE
svc-userauth    latest   b74f92629ed1  3 hours ago  1.11GB
```

The `docker build` command automatically adds the tag, `latest`, if we do not specify a tag.

Then, to push the images to the ECR repositories, we execute these commands:

```
$ sh tag.sh
$ sh push.sh
The push refers to repository [09E1X6A8MPLE.dkr.ecr.us-
west-2.amazonaws.com/svc-notes]
6005576570e9: Pushing 18.94kB
cac3b3d9d486: Pushing 7.014MB/96.89MB
107af8db3a4: Pushing 14.85kB
df143eb62095: Pushing 17.41kB
6b61442be5f8: Pushing 3.717MB
0c719438462a: Waiting
8c98a57451eb: Waiting
...
latest: digest:
sha256:1ea31c507e9714704396f01f5cdad62525d9694e5b09e2e7b08c3cb2ebd6d6f
f size: 4722
The push refers to repository [09E1X6A8MPLE.dkr.ecr.us-
west-2.amazonaws.com/svc-userauth]
343a794bb161: Pushing 9.12MB/65.13MB
51f07622ae50: Pushed
b12bef22bccb: Pushed
...
```

Since the images are rather large, it will take a long time to upload them to the AWS ECR. We should add a task to the backlog to explore ways to trim Docker image sizes. In any case, expect this to take a while.

After a period of time, the images will be uploaded to the ECR repositories, and you can inspect the results on the ECR dashboard.

Once the Docker images are pushed to the AWS ECR repository, we no longer need to stay with the default Docker context. You will be free to run the following command at any time:

```
$ docker context use ec2
```

Remember that swarm hosts are not to be used for building Docker images. At the beginning of this section, we switched to the default context so that builds would occur on our laptop.

In this section, we learned how to set up a build procedure to push our Docker images to repositories on the AWS ECR service. This included using some interesting tools that simplify building complex build procedures in package.json scripts.

Our next step is learning how to use Docker compose files to describe deployment on Docker Swarm.

Creating a Docker stack file for deployment to Docker Swarm

In the previous sections, we learned how to set up an AWS infrastructure using Terraform. We've designed a VPC that will house the Notes application stack, we experimented with a single-node Docker Swarm cluster built on a single EC2 instance, and we set up a procedure to push the Docker images to the ECR.

Our next task is to prepare a Docker stack file for deployment to the swarm. A stack file is nearly identical to the Docker compose file we used in Chapter 11, *Deploying Node.js Microservices with Docker*. Compose files are used with normal Docker hosts, but stack files are used with swarms. To make it a stack file, we add some new tags and change a few things, including the networking implementation.

Earlier, we kicked the tires of Docker Swarm with the `docker service create` command to launch a service on a swarm. While that was easy, it does not constitute code that can be committed to a source repository, nor is it an automated process.

In swarm mode, a service is a definition of the tasks to execute on swarm nodes. Each service consists of a number of tasks, with this number depending on the replica settings. Each task is a container that has been deployed to a node in the swarm. There are, of course, other configuration parameters, such as network ports, volume connections, and environment variables.

The Docker platform allows the use of the compose file for deploying services to a swarm. When used this way, the compose file is referred to as a stack file. There is a set of `docker stack` commands for handling the stack file, as follows:

- On a regular Docker host, the `docker-compose.yml` file is called a compose file. We use the `docker-compose` command on a compose file.
- On a Docker swarm, the `docker-compose.yml` file is called a stack file. We use the `docker stack` command on a stack file.

Remember that a compose file has a `services` tag, and each entry in that tag is a container configuration to deploy. When used as a stack file, each `services` tag entry is, of course, a service in the sense just described. This means that just as there was a lot of similarity between the `docker run` command and container definitions in the compose file, there is a degree of similarity between the `docker service create` command and the service entries in the stack file.

One important consideration is a policy that builds must not happen on Swarm host machines. Instead, these machines must be used solely for deploying and executing containers. This means that any `build` tag in a service listed in a stack file is ignored. Instead, there is a `deploy` tag that has parameters for the deployment in the swarm, and the `deploy` tag is ignored when the file is used with Compose. Put more simply, we can have the same file serve both as a compose file (with the `docker compose` command) and as a stack file (with the `docker stack` command), with the following conditions:

- When used as a compose file, the `build` tag is used and the `deploy` tag is ignored.
- When used as a stack file, the `build` tag is ignored and the `deploy` tag is used.

Another consequence of this policy is the necessity of switching the Docker context as appropriate. We have already discussed this issue—that we use the `default` Docker context to build images on our laptop and we use the EC2 context when interacting with the swarm on the AWS EC2 instances.

To get started, create a directory named `compose-stack` that's a sibling to `compose-local`, `notes`, `terraform-swarm`, and the other directories. Then, copy `compose-local/docker-compose.yml` into `compose-stack`. This way, we can start from something we know is working well.

This means that we'll create a Docker stack file from our compose file. There are several steps involved, which we'll cover over the next several sections. This includes adding deploy tags, configuring networking for the swarm, controlling the placement of services in the swarm, storing secrets in the swarm, and other tasks.

Creating a Docker stack file from the Notes Docker compose file

With that theory under our belts, let's now take a look at the existing Docker compose file and see how to make it useful for deployment to a swarm.

Since we will require some advanced `docker-compose.yml` features, update the version number to the following:

```
version: '3.8'
```

For the Compose file we started with, version '3' was adequate, but to accomplish the tasks in this chapter the higher version number is required, to enable newer features.

Fortunately, most of this is straightforward and will require very little code.



Deployment parameters: These are expressed in the `deploy` tag, which covers things such as the number of replicas, and memory or CPU requirements. For documentation, refer to <https://docs.docker.com/compose/compose-file/#deploy>.

For the deployment parameters, simply add a `deploy` tag to each service. Most of the options for this tag have perfectly reasonable defaults. To start with, let's add this to every service, as follows:

```
deploy:  
  replicas: 1
```

This tells Docker that we want one instance of each service. Later, we will experiment with adding more service instances. We will add other parameters later, such as placement constraints. Later, we will want to experiment with multiple replicas for both `svc-notes` and `svc-userauth`. It is tempting to put CPU and memory limits on the service, but this isn't necessary.

It is nice to learn that with swarm mode, we can simply change the `replicas` setting to change the number of instances.

The next thing to take care of is the image name. While the build tag is present, remember that it is ignored. For the Redis and database containers, we are already using images from Docker Hub, but for svc-notes and svc-userauth, we are building our own containers. This is why, earlier in this chapter, we set up a procedure for pushing the images to ECR repositories. We can now reference those images from the stack file. This means that we must make the following change:

```
services:  
...  
  svc-userauth:  
    build: ../users  
    image: 098E0X9AMPLE.dkr.ecr.us-REGION-2.amazonaws.com/svc-userauth  
...  
  
  svc-notes:  
    build: ../notes  
    image: 098E0X9AMPLE.dkr.ecr.us-REGION-2.amazonaws.com/svc-notes  
...
```

If we use this with `docker-compose`, it will perform the build in the named directories, and then tag the resulting image with the tag in the `image` field. In this case, the `deploy` tag will be ignored as well. However, if we use this with `docker stack deploy`, the `build` tag will be ignored, and the images will be downloaded from the repositories listed in the `image` tag. In this case, the `deploy` tag will be used.



For documentation on the `build` tag, refer to <https://docs.docker.com/compose/compose-file/#build>. For documentation on the `image` tag, refer to <https://docs.docker.com/compose/compose-file/#image>.

When running the compose file on our laptop, we used bridge networking. This works fine for a single host, but with swarm mode, we need another network mode that handles multi-host deployments. The Docker documentation clearly says to use the `overlay` driver in swarm mode, and the `bridge` driver for a single-host deployment.



Virtual networking for containers: Since bridge networking is designed for a single-host deployment, we must use overlay networking in swarm mode. For documentation, refer to <https://docs.docker.com/compose/compose-file/#network-configuration-reference>.

To use overlay networking, change the `networks` tag to the following:

```
networks:  
  frontnet:  
    # driver: bridge  
    driver: overlay  
  authnet:  
    # driver: bridge  
    driver: overlay  
  svcnet:  
    # driver: bridge  
    driver: overlay
```

To support switching between using this for a swarm, or for a single-host deployment, we can leave the `bridge` network setting available but commented out. We would then change whether `overlay` or `bridge` networking is active by changing which is commented, depending on the context.

The `overlay` network driver sets up a virtual network across the swarm nodes. This network supports communication between the containers and also facilitates access to the externally published ports.

The `overlay` network configures the containers in a swarm to have a domain name automatically assigned that matches the service name. As with the `bridge` network we used before, containers find each other via the domain name. For a service deployed with multiple instances, the `overlay` network ensures that requests to that container can be routed to any of its instances. If a connection is made to a container but there is no instance of that container on the same host, the `overlay` network routes the request to an instance on another host. This is a simple approach to service discovery, by using domain names, but extending it across multiple hosts in a swarm.

That took care of the easy tasks for converting the compose file to a stack file. There are a few other tasks that will require more attention, however.

Placing containers across the swarm

We haven't done it yet, but we will add multiple EC2 instances to the swarm. By default, swarm mode distributes tasks (containers) evenly across the swarm nodes. However, we have two considerations that should force some containers to be deployed on specific Docker hosts—namely, the following:

1. We have two database containers and need to arrange persistent storage for the data files. This means that the databases must be deployed to the same instance every time so that it can use the same data directory.
2. The public EC2 instance, named `notes-public`, will be part of the swarm. To maintain the security model, most of the services should not be deployed on this instance but on the instances that will be attached to the private subnet. Therefore, we should strictly control which containers deploy to `notes-public`.

Swarm mode lets us declare the placement requirements for any service. There are several ways to implement this, such as matching against the hostname, or against labels that can be assigned to each node.

For documentation on the stack file placement tag, refer to <https://docs.docker.com/compose/compose-file/#placement>.



The documentation for the `docker stack create` command includes a further explanation of deployment parameters: https://docs.docker.com/engine/reference/commandline/service_create.

Add this `deploy` tag to the `db-userauth` service declaration:

```
services:  
...  
db-userauth:  
..  
deploy:  
  replicas: 1  
  placement:  
    constraints:  
      # - "node.hostname==notes-private-db1"  
      - "node.labels.type==db"  
...
```

The placement tag governs where the containers are deployed. Rather than Docker evenly distributing the containers, we can influence the placement with the fields in this tag. In this case, we have two examples, such as deploying a container to a specific node based on the hostname or selecting a node based on the labels attached to the node.

To set a label on a Docker swarm node, we run the following command:

```
$ docker node update --label-add type=public notes-public
```

This command attaches a label named `type`, with the value `public`, to the node named `notes-public`. We use this to set labels, and, as you can see, the label can have any name and any value. The labels can then be used, along with other attributes, as influence over the placement of containers on swarm nodes.

For the rest of the stack file, add the following placement constraints:

```
services:
...
  svc-userauth:
    ...
    deploy:
      replicas: 1
      placement:
        constraints:
          - "node.labels.type==svc"
...
  db-notes:
    ...
    deploy:
      replicas: 1
      placement:
        constraints:
          - "node.labels.type==db"
...
  svc-notes:
    ...
    deploy:
      replicas: 1
      placement:
        constraints:
          - "node.labels.type==public"
...
  redis:
    ...
    deploy:
      replicas: 1
```

```
placement:  
  constraints:  
    - "node.labels.type!=public"  
...
```

This gives us three labels to assign to our EC2 instances: `db`, `svc`, and `public`. These constraints will cause the databases to be placed on nodes where the `type` label is `db`, the user authentication service is on the node of type `svc`, the Notes service is on the `public` node, and the Redis service is on any node that is not the `public` node.

The reasoning stems from the security model we designed. The containers deployed on the private network should be more secure behind more layers of protection. This placement leaves the Notes container as the only one on the public EC2 instance. The other containers are split between the `db` and `svc` nodes. We'll see later how these labels will be assigned to the EC2 instances we'll create.

Configuring secrets in Docker Swarm

With Notes, as is true for many kinds of applications, there are some secrets we must protect. Primarily, this is the Twitter authentication tokens, and we've claimed it could be a company-ending event if those tokens were to leak to the public. Maybe that's overstating the danger, but leaked credentials could be bad. Therefore, we must take measures to ensure that those secrets do not get committed to a source repository as part of any source code, nor should they be recorded in any other file.

For example, the Terraform state file records all information about the infrastructure, and the Terraform team makes no effort to detect any secrets and suppress recording them. It's up to us to make sure the Terraform state file does not get committed to source code control as a result.

Docker Swarm supports a very interesting method for securely storing secrets and for making them available in a secure manner in containers.

The process starts with the following command:

```
$ printf 'vuTghgEXAMPLE...' | docker secret create  
TWITTER_CONSUMER_KEY -  
$ printf 'tOtJqaEXAMPLE...' | docker secret create  
TWITTER_CONSUMER_SECRET -
```

This is how we store a secret in a Docker swarm. The `docker secret create` command first takes the name of the secret, and then a specifier for a file containing the text for the secret. This means we can either store the data for the secret in a file or—as in this case—we use `-` to specify that the data comes from the standard input. In this case, we are using the `printf` command, which is available for macOS and Linux, to send the value into the standard input.

Docker Swarm securely records the secrets as encrypted data. Once you've given a secret to Docker, you cannot inspect the value of that secret.

In `compose-stack/docker-compose.yml`, add this declaration at the end:

```
secrets:  
  TWITTER_CONSUMER_KEY:  
    external: true  
  TWITTER_CONSUMER_SECRET:  
    external: true
```

This lets Docker know that this stack requires the value of those two secrets.

The declaration for `svc-notes` also needs the following command:

```
services:  
  ...  
  svc-notes:  
    ...  
    secrets:  
      - TWITTER_CONSUMER_KEY  
      - TWITTER_CONSUMER_SECRET  
    ...  
    environment:  
      ...  
      TWITTER_CONSUMER_KEY_FILE: /var/run/secrets/TWITTER_CONSUMER_KEY  
      TWITTER_CONSUMER_SECRET_FILE:  
        /var/run/secrets/TWITTER_CONSUMER_SECRET  
    ...
```

This notifies the swarm that the Notes service requires the two secrets. In response, the swarm will make the data for the secrets available in the filesystem of the container

as `/var/run/secrets/TWITTER_CONSUMER_KEY` and `/var/run/secrets/TWITTER_CONSUMER_SECRET`. They are stored as in-memory files and are relatively secure.

To summarize, the steps required are as follows:

- Use `docker secret create` to register the secret data with the swarm.
- In the stack file, declare `secrets` in a top-level secrets tag.
- In services that require the secrets, declare a `secrets` tag that lists the secrets required by this service.
- In the environments tag for the service, create an environment variable pointing to the `secrets` file.

The Docker team has a suggested convention for configuration of environment variables. You could supply the configuration setting directly in an environment variable, such as `TWITTER_CONSUMER_KEY`. However, if the configuration setting is in a file, then the filename should be given in a different environment variable whose name has `_FILE` appended. For example, we would use `TWITTER_CONSUMER_KEY` or `TWITTER_CONSUMER_KEY_FILE`, depending on whether the value is directly supplied or in a file.

This then means that we must rewrite Notes to support reading these values from the files, in addition to the existing environment variables.

To support reading from files, add this import to the top of `notes/routes/users.mjs`:

```
import fs from 'fs-extra';
```

Then, we'll find the code corresponding to these environment variables further down the file. We should rewrite that section as follows:

```
const twittercallback = process.env.TWITTER_CALLBACK_HOST
  ? process.env.TWITTER_CALLBACK_HOST
  : "http://localhost:3000";
export var twitterLogin = false;
let consumer_key;
let consumer_secret;

if (typeof process.env.TWITTER_CONSUMER_KEY !== 'undefined'
  && process.env.TWITTER_CONSUMER_KEY !== ''
  && typeof process.env.TWITTER_CONSUMER_SECRET !== 'undefined'
  && process.env.TWITTER_CONSUMER_SECRET !== '') {

  consumer_key = process.env.TWITTER_CONSUMER_KEY;
  consumer_secret = process.env.TWITTER_CONSUMER_SECRET;
  twitterLogin = true;

} else if (typeof process.env.TWITTER_CONSUMER_KEY_FILE !==
```

```
'undefined'
&& process.env.TWITTER_CONSUMER_KEY_FILE !== ''
&& typeof process.env.TWITTER_CONSUMER_SECRET_FILE !== 'undefined'
&& process.env.TWITTER_CONSUMER_SECRET_FILE !== '') {

  consumer_key =
    fs.readFileSync(process.env.TWITTER_CONSUMER_KEY_FILE, 'utf8');
  consumer_secret =
    fs.readFileSync(process.env.TWITTER_CONSUMER_SECRET_FILE, 'utf8');
  twitterLogin = true;
}

if (twitterLogin) {
  passport.use(new TwitterStrategy({
    consumerKey: consumer_key,
    consumerSecret: consumer_secret,
    callbackURL: `${twittercallback}/users/auth/twitter/callback`
  },
  async function(token, tokenSecret, profile, done) {
    try {
      done(null, await usersModel.findOrCreate({
        id: profile.username, username: profile.username, password:
        "",
        provider: profile.provider, familyName: profile.displayName,
        givenName: "", middleName: "",
        photos: profile.photos, emails: profile.emails
      }));
    } catch(err) { done(err); }
  }));
}
```

This is similar to the code we've already used but organized a little differently. It first tries to read the Twitter tokens from the environment. Failing that, it tries to read them from the named files. Because this code is executing in the global context, we must read the files using `readFileSync`.

If the tokens are available from either source, the `twitterLogin` variable is set, and then we enable the support for `TwitterStrategy`. Otherwise, Twitter support is disabled. We had already organized the views templates so that if `twitterLogin` is `false`, the Twitter login buttons do not appear.

All of this is what we did in [Chapter 8, *Authenticating Users with a Microservice*](#), but with the addition of reading the tokens from a file.

Persisting data in a Docker swarm

The data persistence strategy we used in [Chapter 11, Deploying Node.js Microservices with Docker](#), required the database files to be stored in a volume. The directory for the volume lives outside the container and survives when we destroy and recreate the container.

That strategy relied on there being a single Docker host for running containers. The volume data is stored in a directory in the host filesystem. But in swarm mode, volumes do not work in a compatible fashion.

With Docker Swarm, unless we use placement criteria, containers can deploy to any swarm node. The default behavior for a named volume in Docker is that the data is stored on the current Docker host. If the container is redeployed, then the volume is destroyed on the one host and a new one is created on the new host. Clearly, that means that the data in that volume is not persistent.



For documentation about using volumes in a Docker Swarm, refer to <https://docs.docker.com/compose/compose-file/#volumes-for-services-swarms-and-stack-files>.

What's recommended in the documentation is to use placement criteria to force such containers to deploy to specific hosts. For example, the criteria we discussed earlier deploy the databases to a node with the `type` label equal to `db`.

In the next section, we will make sure that there is exactly one such node in the swarm. To ensure that the database data directories are at a known location, let's change the declarations for the `db-userauth` and `db-notes` containers, as follows:

```
services:  
  ...  
  db-userauth:  
    ...  
    volumes:  
      # - db-userauth-data:/var/lib/mysql  
      - type: bind  
        source: /data/users  
        target: /var/lib/mysql  
    ...  
  
  db-notes:  
    ...  
    volumes:  
      # - db-notes-data:/var/lib/mysql
```

```
- type: bind  
  source: /data/notes  
  target: /var/lib/mysql  
  ...  
# volumes:  
#   db-userauth-data:  
#   db-notes-data:  
...
```

In `docker-local/docker-compose.yml`, we used the named volumes, `db-userauth-data` and `db-notes-data`. The top-level `volumes` tag is required when doing this. In `docker-swarm/docker-compose.yml`, we've commented all of that out. Instead, we are using a `bind` mount, to mount specific host directories in the `/var/lib/mysql` directory of each database.

Therefore, the database data directories will be in `/data/users` and `/data/notes`, respectively.

This result is fairly good, in that we can destroy and recreate the database containers at will and the data directories will persist. However, this is only as persistent as the EC2 instance this is deployed to. The data directories will vaporize as soon as we execute `terraform destroy`.

That's obviously not good enough for a production deployment, but it is good enough for a test deployment such as this.

It is preferable to use a volume instead of the `bind` mount we just implemented. Docker volumes have a number of advantages, but to make good use of a volume requires finding the right volume driver for your needs. Two examples are as follows:

1. In the Docker documentation, at <https://docs.docker.com/storage/volumes/>, there is an example of mounting a **Network File System (NFS)** volume in a Docker container. AWS offers an NFS service—the **Elastic Filesystem (EFS)** service—that could be used, but this may not be the best choice for a database container.
2. The REX-Ray project (<https://github.com/rexray/rexray>) aims to advance the state of the art for persistent data storage in various containerization systems, including Docker.

Another option is to completely skip running our own database containers and instead use the **Relational Database Service (RDS)**. RDS is an AWS service offering several **Structured Query Language (SQL)** database solutions, including MySQL. It offers a lot of flexibility and scalability, at a price. To use this, you would eliminate the db-notes and db-userauth containers, provision RDS instances, and then update the SEQUELIZE_CONNECT configuration in svc-notes and svc-userauth to use the database host, username, and password you configured in the RDS instances.

For our current requirements, this setup, with a bind mount to a directory on the EC2 host, will suffice. These other options are here for your further exploration.

In this section, we converted our Docker compose file to be useful as a stack file. While doing this, we discussed the need to influence which swarm host has which containers. The most critical thing is ensuring that the database containers are deployed to a host where we can easily persist the data—for example, by running a database backup every so often to external storage. We also discussed storing secrets in a secure manner so that they may be used safely by the containers.

At this point, we cannot test the stack file that we've created because we do not have a suitable swarm to deploy to. Our next step is writing the Terraform configuration to provision the EC2 instances. That will give us the Docker swarm that lets us test the stack file.

Provisioning EC2 instances for a full Docker swarm

So far in this chapter, we have used Terraform to create the required infrastructure on AWS, and then we set up a single-node Docker swarm on an EC2 instance to learn about Docker Swarm. After that, we pushed the Docker images to ECR, and we have set up a Docker stack file for deployment to a swarm. We are ready to set up the EC2 instances required for deploying a full swarm.

Docker Swarm is able to handle Docker deployments to large numbers of host systems. Of course, the Notes application only has delusions of grandeur and doesn't need that many hosts. We'll be able to do everything with three or four EC2 instances. We have declared one so far, and will declare two more that will live on the private subnet. But from this humble beginning, it would be easy to expand to more hosts.

Our goal in this section is to create an infrastructure for deploying Notes on EC2 using Docker Swarm. This will include the following:

- Configuring additional EC2 instances on the private subnet, installing Docker on those instances, and joining them together in a multi-host Docker Swarm
- Creating semi-automated scripting, thereby making it easy to deploy and configure the EC2 instances for the swarm
- Using an `nginx` container on the public EC2 instance as a proxy in front of the Notes container

That's quite a lot of things to take care of, so let's get started.

Configuring EC2 instances and connecting to the swarm

We have one EC2 instance declared for the public subnet, and it is necessary to add two more for the private subnet. The security model we discussed earlier focused on keeping as much as possible in a private secure network infrastructure. On AWS, that means putting as much as possible on the private subnet.

Earlier, you may have renamed `ec2-public.tf` to `ec2-public.tf-disable`. If so, you should now change back the filename to `ec2-public.tf`. Remember that this tactic is useful for minimizing AWS resource usage when it is not needed.

Create a new file in the `terraform-swarm` directory named `ec2-private.tf`, as follows:

```
resource "aws_instance" "private-db1" {
  ami = var.ami_id
  // instance_type = var.instance_type
  instance_type = "t2.medium"
  subnet_id = aws_subnet.private1.id
  key_name = var.key_pair
  vpc_security_group_ids = [ aws_security_group.ec2-private-sg.id ]
  associate_public_ip_address = false

  root_block_device {
    volume_size = 50
  }

  tags = {
```

```
        Name = "${var.project_name}-ec2-private-db1"
    }

depends_on = [ aws_vpc.notes, aws_internet_gateway.igw ]
user_data = join("\n", [
    "#!/bin/sh",
    file("sh/docker_install.sh"),
    "mkdir -p /data/notes /data/users",
    "sudo hostname ${var.project_name}-private-db1"
])
}

resource "aws_instance" "private-svc1" {
    ami = var.ami_id
    instance_type = var.instance_type
    subnet_id = aws_subnet.private1.id
    key_name = var.key_pair
    vpc_security_group_ids = [ aws_security_group.ec2-private-sg.id ]
    associate_public_ip_address = false

    tags = {
        Name = "${var.project_name}-ec2-private-svc1"
    }

depends_on = [ aws_vpc.notes, aws_internet_gateway.igw ]
user_data = join("\n", [
    "#!/bin/sh",
    file("sh/docker_install.sh"),
    "sudo hostname ${var.project_name}-private-svc1"
])
}
```

This declares two EC2 instances that are attached to the private subnet. There's no difference between these instances other than the name. Because they're on the private subnet, they are not assigned a public IP address.

Because we use the `private-db1` instance for databases, we have allocated 50 **gigabytes (GB)** for the root device. The `root_block_device` block is for customizing the root disk of an EC2 instance. Among the available settings, `volume_size` sets its size, in GB.

Another difference in `private-db1` is the `instance_type`, which we've hardcoded to `t2.medium`. The issue is about deploying two database containers to this server. A `t2.micro` instance has 1 GB of memory, and the two databases were observed to overwhelm this server. If you want the adventure of debugging that situation, change this value to be `var.instance_type`, which defaults to `t2.micro`, then read the section at the end of the chapter about debugging what happens.

Notice that for the `user_data` script, we only send in the script to install Docker Support, and not the script to initialize a swarm. The swarm was initialized in the public EC2 instance. The other instances must instead join the swarm using the `docker swarm join` command. Later, we will go over initializing the swarm, and see how that's accomplished. For the `public-db1` instance, we also create the `/data/notes` and `/data/users` directories, which will hold the database data directories.

Add the following code to `ec2-private.tf`:

```
resource "aws_security_group" "ec2-private-sg" {
  name = "${var.project_name}-private-sg"
  description = "allow inbound access to the EC2 instance"
  vpc_id = aws_vpc.notes.id

  ingress {
    protocol = "-1"
    from_port = 0
    to_port = 0
    cidr_blocks = [ aws_vpc.notes.cidr_block ]
  }

  ingress {
    description = "Docker swarm (udp)"
    protocol = "UDP"
    from_port = 0
    to_port = 0
    cidr_blocks = [ aws_vpc.notes.cidr_block ]
  }

  egress {
    protocol = "-1"
    from_port = 0
    to_port = 0
    cidr_blocks = [ "0.0.0.0/0" ]
  }
}
```

This is the security group for these EC2 instances. It allows any traffic from inside the VPC to enter the EC2 instances. This is the sort of security group we'd create when in a hurry and should tighten up the ingress rules, since this is very lax.

Likewise, the `ec2-public-sg` security group needs to be equally lax. We'll find that there is a long list of IP ports used by Docker Swarm and that the swarm will fail to operate unless those ports can communicate. For our immediate purposes, the easiest option is to allow any traffic, and we'll leave a note in the backlog to address this issue in Chapter 14, *Security in Node.js Applications*.

In `ec2-public.tf`, edit the `ec2-public-sg` security group to be the following:

```
resource "aws_security_group" "ec2-public-sg" {
  name = "${var.project_name}-public-sg"
  description = "allow inbound access to the EC2 instance"
  vpc_id = aws_vpc.notes.id

  ingress {
    protocol = "-1"
    from_port = 0
    to_port = 0
    cidr_blocks = [ "0.0.0.0/0" ]
  }

  ingress {
    description = "Docker swarm (udp)"
    protocol = "UDP"
    from_port = 0
    to_port = 0
    cidr_blocks = [ aws_vpc.notes.cidr_block ]
  }

  egress {
    protocol = "-1"
    from_port = 0
    to_port = 0
    cidr_blocks = [ "0.0.0.0/0" ]
  }
}
```

This is literally not a best practice since it allows any network traffic from any IP address to reach the public EC2 instance. However, it does give us the freedom to develop the code without worrying about protocols at this moment. We will address this later and implement the best security practice. Have a look at the following code snippet:

```
output "ec2-private-db1-arn" { value = aws_instance.private-db1.arn }
output "ec2-private-db1-dns" { value = aws_instance.private-
db1.private_dns }
output "ec2-private-db1-ip"   { value = aws_instance.private-
db1.private_ip }
output "ec2-private-svc1-arn" { value = aws_instance.private-svc1.arn
}
output "ec2-private-svc1-dns" { value = aws_instance.private-
svc1.private_dns }
output "ec2-private-svc1-ip"  { value = aws_instance.private-
svc1.private_ip }
```

This outputs the useful attributes of the EC2 instances.

In this section, we declared EC2 instances for deployment on the private subnet. Each will have Docker initialized. However, we still need to do what we can to automate the setup of the swarm.

Implementing semi-automatic initialization of the Docker Swarm

Ideally, when we run `terraform apply`, the infrastructure is automatically set up and ready to go. Automated setup reduces the overhead of running and maintaining the AWS infrastructure. We'll get as close to that goal as possible.

For this purpose, let's revisit the declaration of `aws_instance.public` in `ec2-public.tf`. Let's rewrite it as follows:

```
resource "aws_instance" "public" {
  ami = var.ami_id
  instance_type = var.instance_type
  subnet_id = aws_subnet.public1.id
  key_name = var.key_pair
  vpc_security_group_ids = [ aws_security_group.ec2-public-sg.id ]
  associate_public_ip_address = true
  tags = {
    Name = "${var.project_name}-ec2-public"
  }
```

```
depends_on = [
    aws_vpc.notes, aws_internet_gateway.igw,
    aws_instance.private-db1, aws_instance.private-svc1
]

user_data = join("\n", [
    "#!/bin/sh",
    file("sh/docker_install.sh"),
    "docker swarm init",
    "sudo hostname ${var.project_name}-public",
    "docker node update --label-add type=public ${var.project_name}-public",
    templatefile("sh/swarm-setup.sh", {
        instances = [ {
            dns = aws_instance.private-db1.private_dns,
            type = "db",
            name = "${var.project_name}-private-db1"
        }, {
            dns = aws_instance.private-svc1.private_dns,
            type = "svc",
            name = "${var.project_name}-private-svc1"
        } ]
    })
])
]
```

This is largely the same as before, but with two changes. The first is to add references to the private EC2 instances to the `depends_on` attribute. This will delay the construction of the public EC2 instance until after the other two are running.

The other change is to extend the shell script attached to the `user_data` attribute. The first addition to that script is to set the `type` label on the `notes-public` node. That label is used with service placement.

The last change is a script with which we'll set up the swarm. Instead of setting up the swarm in the `user_data` script directly, it will generate a script that we will use in creating the swarm. In the `sh` directory, create a file named `swarm-setup.sh` containing the following:

```
cat >/home/ubuntu/swarm-setup.sh <<EOF
#!/bin/sh

### Capture the file name for the PEM from the command line
PEM=\$1

join=`docker swarm join-token manager | sed 1,2d | sed 2d`"
```

```
%{ for instance in instances ~}
  ssh -i \$PEM ${instance.dns} \$join
  docker node update --label-add type=${instance.type} ${instance.name}
%{ endfor ~}
EOF
```

This generates a shell script that will be used to initialize the swarm. Because the setup relies on executing commands on the other EC2 instances, the PEM file for the AWS key pair must be present on the `notes-public` instance. However, it is not possible to send the key-pair file to the `notes-public` instance when running `terraform apply`. Therefore, we use the pattern of generating a shell script, which will be run later.

The pattern being followed is shown in the following code snippet:

```
cat >/path/to/file <<EOF
...
text to output
EOF
```

The part between `<<EOF` and `EOF` is supplied as the standard input to the `cat` command. The result is, therefore, for `/home/ubuntu/swarm-setup.sh` to end up with the text between those markers. An additional detail is that a number of variable references are escaped, as in `PEM=\$1`. This is necessary so that those variables are not evaluated while setting up this script but are present in the generated script.

This script is processed using the `templatefile` function so that we can use template commands. Primarily, that is the `%{for ... }` loop with which we generate the commands for configuring each EC2 instance. You'll notice that there is an array of data for each instance, which is passed through the `templatefile` invocation.

Therefore, the `swarm-setup.sh` script will contain a copy of the following pair of commands for each EC2 instance:

```
ssh -i $PEM ${instance.dns} \$join
docker node update --label-add type=${instance.type} ${instance.name}
```

The first line uses SSH to execute the `swarm join` command on the EC2 instance. For this to work, we need to supply the AWS key pair, which must be specified on the command file so that it becomes the `PEM` variable. The second line adds the `type` label with the named value to the named swarm node.

What is the `$join` variable? It has the output of running `docker swarm join-token`, so let's take a look at what it is.

Docker uses a swarm join token to facilitate connecting Docker hosts as a node in a swarm. The token contains cryptographically signed information that authenticates the attempt to join the swarm. We get the token by running the following command:

```
$ docker swarm join-token manager  
To add a manager to this swarm, run the following command:  
  
    docker swarm join --token  
SWMTKN-1-11161hnrrjbmzg1r8a46e34dt21s15n4357qrib29csi0jgi823-3g80csolwa  
ioya580hjanwfsf 10.0.3.14:2377
```

The word `manager` here means that we are requesting a token to join as a manager node. To connect a node as a worker, simply replace `manager` with `worker`.

Once the EC2 instances are deployed, we could log in to `notes-public`, and then run this command to get the join token and run that command on each of the EC2 instances. The `swarm-setup.sh` script, however, handles this for us. All we have to do, once the EC2 hosts are deployed, is to log in to `notes-public` and run this script.

It runs the `docker swarm join-token manager` command, piping that user-friendly text through a couple of `sed` commands to extract out the important part. That leaves the `join` variable containing the text of the `docker swarm join` command, and then it uses SSH to execute that command on each of the instances.

In this section, we examined how to automate, as far as possible, the setup of the Docker swarm.

Let's now do it.

Preparing the Docker Swarm before deploying the Notes stack

When you make an omelet, it's best to cut up all the veggies and sausage, prepare the butter, and whip the milk and eggs into a mix before you heat up the pan. In other words, we prepare the ingredients before undertaking the critical action of preparing the dish. What we've done so far is to prepare all the elements of successfully deploying the Notes stack to AWS using Docker Swarm. It's now time to turn on the pan and see how well it works.

We have everything declared in the Terraform files, and we can deploy our complete system with the following command:

```
$ terraform apply
...
Plan: 5 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes
...
```

This deploys the EC2 instances on AWS. Make sure to record all the output parameters. We're especially interested in the domain names and IP addresses for the three EC2 instances.

As before, the notes-public instance should have a Docker swarm initialized. We have added two more instances, notes-private-db1 and notes-private-svc1. Both will have Docker installed, but they are not joined to the swarm. Instead, we need to run the generated shell script for them to become nodes in the swarm, as follows:

```
$ scp ~/Downloads/notes-app-key-pair.pem ubuntu@PUBLIC-IP-ADDRESS:
The authenticity of host '52.39.219.109 (52.39.219.109)' can't be
established.
ECDSA key fingerprint is
SHA256:qdK5ZPn1EtmO1RW1jb0dG3Nu2mDQHtmFwcw4fq9s6vM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '52.39.219.109' (ECDSA) to the list of
known hosts.
notes-app-key-pair.pem 100% 1670 29.2KB/s 00:00

$ ssh ubuntu@PUBLIC-IP-ADDRESS
Welcome to Ubuntu 20.04 LTS (GNU/Linux 5.4.0-1009-aws x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage
...
```

We have already run `ssh-add` on our laptop, and therefore SSH and **secure copy (SCP)** commands can run without explicitly referencing the PEM file. However, the SSH on the `notes-public` EC2 instance does not have the PEM file. Therefore, to access the other EC2 instances, we need the PEM file to be available. Hence, we've used `scp` to copy it to the `notes-public` instance.

If you want to verify the fact that the instances are running and have Docker active, type the following command:

```
ubuntu@notes-public:~$ ssh -i ./notes-app-key-pair.pem \
    ubuntu@IP-FOR-EC2-INSTANCE docker run hello-world
```

In this case, we are testing the private EC2 instances from a shell running on the public EC2 instance. That means we must use the private IP addresses printed when we ran Terraform. This command verifies SSH connectivity to an EC2 instance and verifies its ability to download and execute a Docker image.

Next, we can run `swarm-setup.sh`. On the command line, we must give the filename for the PEM file as the first argument, as follows:

```
ubuntu@notes-public:~$ sh -x swarm-setup.sh ./notes-app-key-pair.pem
+ PEM=./notes-app-key-pair.pem
+ ssh -i ./notes-app-key-pair.pem ip-10-0-3-151.us-
west-2.compute.internal docker swarm join --token
SWMTKN-1-04shb3msc7a1ydqcqmtvhych60wwptkxwcqiexilou6fetx2kg-7robjlgber
03xo44jwx1yofaw 10.0.1.111:2377
...
This node joined a swarm as a manager.
+ docker node update --label-add type=db notes-private-db1
notes-private-db1
+ ssh -i ./notes-app-key-pair.pem ip-10-0-3-204.us-
west-2.compute.internal docker swarm join --token
SWMTKN-1-04shb3msc7a1ydqcqmtvhych60wwptkxwcqiexilou6fetx2kg-7robjlgber
03xo44jwx1yofaw 10.0.1.111:2377
...
This node joined a swarm as a manager.
+ docker node update --label-add type=svc notes-private-svc1
notes-private-svc1
```

We can see this using SSH to execute the `docker swarm join` command on each EC2 instance, causing these two systems to join the swarm, and to set the labels on the instances, as illustrated in the following code snippet:

```
ubuntu@notes-public:~$ docker node ls
ID                  HOSTNAME          STATUS  AVAILABILITY
MANAGER STATUS ENGINE VERSION
```

```
ct7d65v81hw6hxx0k8uk3lw8m notes-private-db1 Ready Active Reachable  
19.03.11  
k1x2h83b01rxnh38p3pypt91x notes-private-svc1 Ready Active Reachable  
19.03.11  
nikgvfe4aum51yu5obqqnnz5s * notes-public Ready Active Leader 19.03.11
```

Indeed, these systems are now part of the cluster.

The swarm is ready to go, and we no longer need to be logged in to notes-public. Exiting back to our laptop, we can create the Docker context to control the swarm remotely, as follows:

```
$ docker context create ec2 --docker host=ssh://ubuntu@PUBLIC-IP-  
ADDRESS  
ec2  
Successfully created context "ec2"  
$ docker context use ec2
```

We've already seen how this works and that, having done this, we will be able to run Docker commands on our laptop; for example, have a look at the following code snippet:

```
$ docker node ls  
ID HOSTNAME STATUS AVAILABILITY  
MANAGER STATUS ENGINE VERSION  
ct7d65v81hw6hxx0k8uk3lw8m notes-private-db1 Ready Active Reachable  
19.03.11  
k1x2h83b01rxnh38p3pypt91x notes-private-svc1 Ready Active Reachable  
19.03.11  
nikgvfe4aum51yu5obqqnnz5s * notes-public Ready Active Leader 19.03.11
```

From our laptop, we can query the state of the remote swarm that's hosted on AWS. Of course, this isn't limited to querying the state; we can run any other Docker command.

We also need to run the following commands, now that the swarm is set up:

```
$ printf 'vuTghgEXAMPLE...' | docker secret create  
TWITTER_CONSUMER_KEY -  
$ printf 'tOtJqaEXAMPLE...' | docker secret create  
TWITTER_CONSUMER_SECRET -
```

Remember that a newly created swarm does not have any secrets. To install the secrets requires these commands to be rerun.

If you wish to create a shell script to automate this process, consider the following:

```
scp $AWS_KEY_PAIR ubuntu@${NOTES_PUBLIC_IP}:
ssh -i $AWS_KEY_PAIR ubuntu@${NOTES_PUBLIC_IP} swarm-setup.sh
`basename ${AWS_KEY_PAIR}`

docker context update --docker host=ssh://ubuntu@${NOTES_PUBLIC_IP}
ec2
docker context use ec2

printf $TWITTER_CONSUMER_KEY | docker secret create
TWITTER_CONSUMER_KEY -
printf $TWITTER_CONSUMER_SECRET | docker secret create
TWITTER_CONSUMER_SECRET -

sh ../ecr/login.sh
```

This script executes the same commands we just went over to prepare the swarm on the EC2 hosts. It requires the environment variables to be set, as follows:

- AWS_KEY_PAIR: The filename for the PEM file
- NOTES_PUBLIC_IP: The IP address of the notes-public EC2 instance
- TWITTER_CONSUMER_KEY, TWITTER_CONSUMER_SECRET: The access tokens for Twitter authentication

In this section, we have deployed more EC2 instances and set up the Docker swarm. While the process was not completely automated, it's very close. All that's required, after using Terraform to deploy the infrastructure, is to execute a couple of commands to get logged in to notes-public where we run a script, and then go back to our laptop to set up remote access.

We have set up the EC2 instances and verified we have a working swarm. We still have the outstanding issue of verifying the Docker stack file created in the previous section. To do so, our next step is to deploy the Notes app on the swarm.

Deploying the Notes stack file to the swarm

We have prepared all the elements required to set up a Docker Swarm on the AWS EC2 infrastructure, we have run the scripts required to set up that infrastructure, and we have created the stack file required to deploy Notes to the swarm.

What's required next is to run `docker stack deploy` from our laptop, to deploy Notes on the swarm. This will give us the chance to test the stack file created earlier. You should still have the Docker context configured for the remote server, making it possible to remotely deploy the stack. However, there are four things to handle first, as follows:

1. Install the secrets in the newly deployed swarm.
2. Update the `svc-notes` environment configuration for the IP address of `notes-public`.
3. Update the Twitter application for the IP address of `notes-public`.
4. Log in to the ECR instance.

Let's take care of those things and then deploy the Notes stack.

Preparing to deploy the Notes stack to the swarm

We are ready to deploy the Notes stack to the swarm that we've launched. However, we have realized that we have a couple of tasks to take care of.

The environment variables for `svc-notes` configuration require a little adjustment. Have a look at the following code block:

```
services:  
..  
  svc-notes:  
    ..  
    environment:  
      # DEBUG: notes:*,express:*
```

REDIS_ENDPOINT: "redis"
 TWITTER_CALLBACK_HOST: "<http://ec2-18-237-70-108.us-west-2.compute.amazonaws.com>"
 TWITTER_CONSUMER_KEY_FILE: /var/run/secrets/TWITTER_CONSUMER_KEY
 TWITTER_CONSUMER_SECRET_FILE:
 /var/run/secrets/TWITTER_CONSUMER_SECRET
 SEQUELIZE_CONNECT: models/sequelize-docker-mysql.yaml
 SEQUELIZE_DBHOST: db-notes
 NOTES_MODEL: sequelize
 ..

Our primary requirement is to adjust the `TWITTER_CALLBACK_HOST` variable. The domain name for the `notes-public` instance changes every time we deploy the AWS infrastructure. Therefore, `TWITTER_CALLBACK_HOST` must be updated to match.

Similarly, we must go to the Twitter developers' dashboard and update the URLs in the application settings. As we already know, this is required every time we have hosted Notes on a different IP address or domain name. To use the Twitter login, we must change the list of URLs recognized by Twitter.

Updating `TWITTER_CALLBACK_HOST` and the Twitter application settings will let us log in to Notes using a Twitter account.

While here, we should review the other variables and ensure that they're correct as well.

The last preparatory step is to log in to the ECR repository. To do this, simply execute the following commands:

```
$ cd ../ecr  
$ sh ./login.sh
```

This has to be rerun every so often since the tokens that are downloaded time out after a few hours.

We only need to run `login.sh`, and none of the other scripts in the `ecr` directory.

In this section, we prepared to run the deployment. We should now be ready to deploy Notes to the swarm, so let's do it.

Deploying the Notes stack to the swarm

We just did the final preparation for deploying the Notes stack to the swarm. Take a deep breath, yell out *Smoke Test*, and type the following command:

```
$ cd ../compose-stack  
$ docker stack deploy --with-registry-auth --compose-file docker-  
compose.yml notes  
...  
Creating network notes_svcnet  
Creating network notes_frontnet  
Creating network notes_authnet  
Creating service notes_svc-userauth  
Creating service notes_db-notes  
Creating service notes_svc-notes
```

```
Creating service notes_redis
Creating service notes_db-userauth
```

This deploys the services, and the swarm responds by attempting to launch each service. The `--with-registry-auth` option sends the Docker Registry authentication to the swarm so that it can download container images from the ECR repositories. This is why we had to log in to the ECR first.

Verifying the correct launch of the Notes application stack

It will be useful to monitor the startup process using these commands:

```
$ docker service ls
ID          NAME          MODE      REPLICAS IMAGE PORTS
17up46slg32g notes_db-notes    replicated 1/1 mysql/mysql-server:8.0
ufw7vwqjkokv notes_db-userauth replicated 1/1 mysql/mysql-server:8.0
45p6uszd9ixt notes_redis     replicated 1/1 redis:5.0
smcju24hvdkj notes_svc-notes  replicated 1/1
098106984154.dkr.ecr.us-west-2.amazonaws.com/svc-notes:latest
*:80->3000/tcp
iws2ff265sqb notes_svc-userauth replicated 1/1
098106984154.dkr.ecr.us-west-2.amazonaws.com/svc-userauth:latest

$ docker service ps notes_svc-notes      # And.. for other service
names
ID          NAME          IMAGE NODE DESIRED STATE CURRENT STATE
ERROR PORTS
nt5rmgv1cf0q notes_svc-notes.1 09E1X6A8MPLE.dkr.ecr.us-
REGION-2.amazonaws.com/svc-notes:latest notes-public Running Running
18 seconds ago
```

The `service ls` command lists the services, with a high-level overview. Remember that the service is not the running container and, instead, the services are declared by entries in the `services` tag in the stack file. In our case, we declared one replica for each service, but we could have given a different amount. If so, the swarm will attempt to distribute that number of containers across the nodes in the swarm.

Notice that the pattern for service names is the name of the stack that was given in the `docker stack deploy` command, followed by the service name listed in the stack file. When running that command, we named the stack `notes`; so, the services are `notes_db-notes`, `notes_svc-userauth`, `notes_redis`, and so on.

The service `ps` command lists information about the tasks deployed for the service. Remember that a task is essentially the same as a running container. We see here that one instance of the `svc-notes` container has been deployed, as expected, on the `notes-public` host.

Sometimes, the `notes_svc-notes` service doesn't launch, and instead, we'll see the following message:

```
$ docker service ps notes_svc-notes
ID          NAME          IMAGE          NODE          DESIRED STATE  CURRENT STATE
ERROR PORTS
nt5rmgv1cf0q  notes_svc-notes.1  0E8X0A9M4PLE.dkr.ecr.us-
REGION-2.amazonaws.com/svc-notes:latest  Running  Pending  9 minutes ago
"no suitable node (scheduling ...")
```

The error, no suitable node, means that the swarm was not able to find a node that matches the placement criteria. In this case, the `type=public` label might not have been properly set.

The following command is helpful:

```
$ docker node inspect notes-public
[
  {
    ...
    "Spec": {
      "Labels": {},
      "Role": "manager",
      "Availability": "active"
    },
    ...
  }
]
```

Notice that the `Labels` entry is empty. In such a case, you can add the label by running this command:

```
$ docker node update --label-add type=public notes-public
notes-public
```

As soon as this is run, the swarm will place the `svc-notes` service on the `notes-public` node.

If this happens, it may be useful to add the following command to the `user_data` script for `aws_instance.public` (in `ec2-public.tf`), just ahead of setting the `type=public` label:

```
"sleep 20",
```

It would appear that this provides a small window of opportunity to allow the swarm to establish itself.

Diagnosing a failure to launch the database services

Another possible deployment problem is that the database services might fail to launch, and the `notes-public-db1` node might become `Unavailable`. Refer back to the `docker node ls` output and you will see a column marked `Status`. Normally, this column says `Reachable`, meaning that the swarm can reach and communicate with the swarm agent on that node. But with the deployment as it stands, this node might instead show an `Unavailable` status, and in the `docker service ls` output, the database services might never show as having deployed.

With remote access from our laptop, we can run the following command:

```
$ docker service ps notes_db-notes
```

The output will tell you the current status, such as any error in deploying the service. However, to investigate connectivity with the EC2 instances, we must log in to the `notes-public` instance as follows:

```
$ ssh ubuntu@PUBLIC-IP-ADDRESS
```

That gets us access to the public EC2 instance. From there, we can try to ping the `notes-private-db1` instance, as follows:

```
ubuntu@notes-public:~$ ping PRIVATE-IP-ADDRESS
PING 10.0.3.141 (10.0.3.141) 56(84) bytes of data.
64 bytes from 10.0.3.141: icmp_seq=1 ttl=64 time=0.481 ms
^C
```

This should work, but the output from `docker node ls` may show the node as `Unreachable`. Ask yourself: what happens if a computer runs out of memory? Then, recognize that we've deployed two database instances to an EC2 instance that has only 1 GB of memory—the memory capacity of `t2.micro` EC2 instances as of the time of writing. Ask yourself whether it is possible that the services you've deployed to a given server have overwhelmed that server.

To test that theory, make the following change in `ec2-private.tf`:

```
resource "aws_instance" "private-db1" {  
    ...  
    instance_type = "t2.medium" // var.instance_type  
    ...  
}
```

This changes the instance type from `t2.micro` to `t2.medium`, or even `t2.large`, thereby giving the server more memory.

To implement this change, run `terraform apply` to update the configuration. If the swarm does not automatically correct itself, then you may need to run `terraform destroy` and then run through the setup again, starting with `terraform apply`.

Once the `notes-private-db1` instance has sufficient memory, the databases should successfully deploy.

In this section, we deployed the Notes application stack to the swarm cluster on AWS. We also talked a little about how to verify the fact that the stack deployed correctly, and how to handle some common problems.

Next, we have to test the deployed Notes stack to verify that it works on AWS.

Testing the deployed Notes application

Having set up everything required to deploy Notes to AWS using Docker Swarm, we have done so. That means our next step is to put Notes through its paces. We've done enough ad hoc testing on our laptop to have confidence it works, but the Docker swarm deployment might show up some issues.

In fact, the deployment we just made very likely has one or two problems. We can learn a lot about AWS and Docker Swarm by diagnosing those problems together.

The first test is obviously to open the Notes application in the browser. In the outputs from running `terraform apply` was a value labeled `ec2-public-dns`. This is the domain name for the `notes-public` EC2 instance. If we simply paste that domain name into our browser, the Notes application should appear.

However, we cannot do anything because there are no user IDs available to log in with.

Logging in with a regular account on Notes

Obviously, in order to test Notes, we must log in and add some notes, make some comments, and so forth. It will be instructive to log in to the user authentication service and use `cli.mjs` to add a user ID.

The user authentication service is on one of the private EC2 instances, and its port is purposely not exposed to the internet. We could change the configuration to expose its port and then run `cli.mjs` from our laptop, but that would be a security problem and we need to learn how to access the running containers anyway.

We can find out which node the service is deployed on by using the following command:

```
$ docker service ps notes_svc-userauth
ID           NAME          IMAGE NODE DESIRED STATE CURRENT
STATE ERROR PORTS
b8jf5q8xlbs5 notes_svc-userauth.1 0E8X0A9M4PLE.dkr.ecr.us-
REGION-2.amazonaws.com/svc-userauth:latest notes-private-svc1 Running
Running 31 minutes ago
```

The `notes_svc-userauth` task has been deployed to `notes-private-svc1`, as expected.

To run `cli.mjs`, we must get shell access inside the container. Since it is deployed on a private instance, this means that we must first SSH to the `notes-public` instance; from there, SSH to the `notes-private-svc1` instance; and from there, run the `docker exec` command to launch a shell in the running container, as illustrated in the following code block:

```
$ ssh ubuntu@PUBLIC-IP-ADDRESS
...
ubuntu@notes-public:~$ ssh -i notes-app-key-pair.pem ubuntu@PRIVATE-
IP-ADDRESS
...
ubuntu@notes-private-svc1:~$ docker ps | grep userauth
e7398953b808 0E8X0A9M4PLE.dkr.ecr.us-REGION-2.amazonaws.com/svc-
userauth:latest "docker-entrypoint.s..." 37 minutes ago Up 37 minutes
5858/tcp notes_svc-userauth.1.b8jf5q8xlbs5b8xk7qpkz9a3w

ubuntu@notes-private-svc1:~$ docker exec -it notes_svc-
userauth.1.b8jf5q8xlbs5b8xk7qpkz9a3w bash
root@e7398953b808:/userauth#
```

We SSHd to the notes-public server and, from there, SSHd to the notes-private-svc1 server. On that server, we ran `docker ps` to find out the name of the running container. Notice that Docker generated a container name that includes a coded string, called a *nonce*, that guarantees the container name is unique. With that container name, we ran `docker exec -it ... bash` to get a root shell inside the container.

Once there, we can run the following command:

```
root@e7398953b808:/userauth# node cli.mjs add --family-name
Einarsdottir --given-name Ashildr --email me@stolen.tardis --password
w0rd me
Created {
  id: 'me',
  username: 'me',
  provider: 'local',
  familyName: 'Einarsdottir',
  givenName: 'Ashildr',
  middleName: null,
  emails: [ 'me@stolen.tardis' ],
  photos: []
}
```

This verifies that the user authentication server works and that it can communicate with the database. To verify this even further, we can access the database instance, as follows:

```
ubuntu@notes-public:~$ ssh -i notes-app-key-pair.pem ubuntu@10.0.3.141
...
ubuntu@notes-private-db1:~$ docker exec -it notes_db-
userauth.1.0b274ges82otektamq059x7w mysql -u userauth -p --socket
/tmp/mysql.sock
Enter password:
```

From there, we can explore the database and see that, indeed, Ashildr's user ID exists.

With this user ID set up, we can now use our browser to visit the Notes application and log in with that user ID.

Diagnosing an inability to log in with Twitter credentials

The next step will be to test logging in with Twitter credentials. Remember that earlier, we said to ensure that the `TWITTER_CALLBACK_HOST` variable has the domain name of the EC2 instance, and likewise that the Twitter application configuration does as well.

Even with those settings in place, we might run into a problem. Instead of logging in, we might get an error page with a stack trace, starting with the message: Failed to obtain request token.

There are a number of possible issues that can cause this error. For example, the error can occur if the Twitter authentication tokens are not deployed. However, if you followed the directions correctly, they will be deployed correctly.

In `notes/appsupport.mjs`, there is a function, `basicErrorHandler`, which will be invoked by this error. In that function, add this line of code:

```
debug('basicErrorHandler err= ', err);
```

This will print the full error, including the originating error that caused the failure. You may see the following message printed: `getaddrinfo EAI AGAIN api.twitter.com`. That may be puzzling because that domain name is certainly available. However, it might not be available inside the `svc-notes` container due to the DNS configuration.

From the `notes-public` instance, we will be able to ping that domain name, as follows:

```
ubuntu@notes-public:~$ ping api.twitter.com
PING tpop-api.twitter.com (104.244.42.2) 56(84) bytes of data.
64 bytes from 104.244.42.2: icmp_seq=1 ttl=38 time=22.1 ms
```

However, if we attempt this inside the `svc-notes` container, this might fail, as illustrated in the following code snippet:

```
ubuntu@notes-public:~$ docker exec -it notes_svc-
notes.1.et3b1obkp9fup5tj7bdco3188 bash
root@c2d002681f61:/notesapp# ping api.twitter.com
... possible failure
```

Ideally, this will work from inside the container as well. If this fails inside the container, it means that the Notes service cannot reach Twitter to handle the OAuth dance required to log in with Twitter credentials.

The problem is that, in this case, Docker set up an incorrect DNS configuration, and the container was unable to make DNS queries for many domain names. In the Docker Compose documentation, it is suggested to use the following code in the service definition:

```
services:  
  ...  
  svc-notes:  
    ...  
    dns:  
      - 8.8.8.8  
      - 9.9.9.9  
    ...
```

These two DNS servers are operated by Google, and indeed this solves the problem. Once this change has been made, you should be able to log in to Notes using Twitter credentials.

In this section, we tested the Notes application and discussed how to diagnose and remedy a couple of common problems. While doing so, we learned how to navigate our way around the EC2 instances and the Docker Swarm.

Let's now see what happens if we change the number of instances for our services.

Scaling the Notes instances

By now, we have deployed the Notes stack to the cluster on our EC2 instances. We have tested everything and know that we have a correctly functioning system deployed on AWS. Our next task is to increase the number of instances and see what happens.

To increase the instances for `svc-notes`, edit `compose-swarm/docker-compose.yml` as follows:

```
services:  
  ...  
  svc-notes:  
    ...  
    deploy:  
      replicas: 2  
    ...
```

This increases the number of replicas. Because of the existing placement constraints, both instances will deploy to the node with a `type` label of `public`. To update the services, it's just a matter of rerunning the following command:

```
$ docker stack deploy --with-registry-auth --compose-file docker-  
compose.yml notes  
Ignoring unsupported options: build, restart  
...  
Updating service notes_svc-userauth (id: wjugeeaje35v3fsgq9t0r8t98)  
Updating service notes_db-notes (id: ldfmq3na5e3ofoyypub3ppth6)  
Updating service notes_svc-notes (id: p194hcjrwaalqbr9pqahur5aj)  
Updating service notes_redis (id: lrjne8uws8kqocmr0ml3kw2wu)  
Updating service notes_db-userauth (id: 1kbj8ax2cj2qzu7winx4kbju0)
```

Earlier, this command described its actions with the word *Creating*, and this time it used the word *Updating*. This means that the services are being updated with whatever new settings are in the stack file.

After a few minutes, you may see this:

```
$ docker service ls  
ID           NAME          MODE      REPLICAS IMAGE PORTS  
ldfmq3na5e3o notes_db-notes   replicated 1/1 mysql/mysql-server:8.0  
1kbj8ax2cj2q notes_db-userauth replicated 1/1 mysql/mysql-server:8.0  
lrjne8uws8kq notes_redis     replicated 1/1 redis:5.0  
p194hcjrwaal notes_svc-notes replicated 2/2 098106984154.dkr.ecr.us-  
west-2.amazonaws.com/svc-notes:latest *:80->3000/tcp  
wjugeeaje35v notes_svc-userauth replicated 1/1  
098106984154.dkr.ecr.us-west-2.amazonaws.com/svc-userauth:latest
```

And indeed, it shows two instances of the `svc-notes` service. The `2/2` notation says that two instances are currently running out of the two instances that were requested.

To view the details, run the following command:

```
$ docker service ps notes_svc-notes  
...
```

As we saw earlier, this command lists to which swarm nodes the service has been deployed. In this case, we'll see that both instances are on `notes-public`, due to the placement constraints.

Another useful command is the following:

```
$ docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
d46c6bba56d3 098106984154.dkr.ecr.us-west-2.amazonaws.com/svc-  
notes:latest "docker-entrypoint.s..." 7 minutes ago Up 7 minutes  
3000/tcp notes_svc-notes.2.zo2mdxk9fuy33ixe0245y7uui  
a93f1d5d8453 098106984154.dkr.ecr.us-west-2.amazonaws.com/svc-  
notes:latest "docker-entrypoint.s..." 15 minutes ago Up 15 minutes  
3000/tcp notes_svc-notes.1.cc34q3yfeumx0b57y1mnpskar
```

Ultimately, each service deployed to a Docker swarm contains one or more running containers.

You'll notice that this shows `svc-notes` listening on port 3000. In the environment setup, we did not set the `PORT` variable, and therefore `svc-notes` will default to listening to port 3000. Refer back to the output for `docker service ls`, and you should see this: `* :80->3000/tcp`, meaning that there is mapping being handled in Docker from port 80 to port 3000.

That is due to the following setting in `docker-swarm/docker-compose.yml`:

```
services:  
  ...  
  svc-notes:  
    ...  
    ports:  
      - "80:3000"  
  ...
```

This says to publish port 80 and to map it to port 3000 on the containers.

In the Docker documentation (<https://docs.docker.com/network/overlay/#bypass-the-routing-mesh-for-a-swarm-service>), we learned that services deployed in a swarm are reachable by the so-called *routing mesh*. Connecting to a published port routes the connection to one of the containers handling that service. As a result, Docker acts as a load balancer, distributing traffic among the service instances you configure.

In this section, we have—finally—deployed the Notes application stack to a cloud hosting environment we built on AWS EC2 instances. We created a Docker swarm, configured the swarm, created a stack file with which to deploy our services, and we deployed to that infrastructure. We then tested the deployed system and saw that it functioned well.

With that, we can wrap up this chapter.

Summary

This chapter is the culmination of a journey of learning Node.js application deployment. We developed an application existing solely on our laptop and added a number of useful features. With the goal of deploying that application on a public server to gain feedback, we worked on three types of deployment. In Chapter 10, *Deploying Node.js Applications to Linux Servers*, we learned how to launch persistent background tasks on Linux using PM2. In Chapter 11, *Deploying Node.js Microservices with Docker*, we learned how to dockerize the Notes application stack, and how to get it running with Docker.

In this chapter, we built on that and learned how to deploy our Docker containers on a Docker Swarm cluster. AWS is a powerful and comprehensive cloud hosting platform with a long list of possible services to use. We used EC2 instances in a VPC and the related infrastructure.

To facilitate this, we used Terraform, a popular tool for describing cloud deployments not just on AWS but on many other cloud platforms. Both AWS and Terraform are widely used in projects both big and small.

In the process, we learned a lot about AWS, and Terraform, and using Terraform to deploy infrastructure on AWS; how to set up a Docker Swarm cluster; and how to deploy a multi-container service on that infrastructure.

We began by creating an AWS account, setting up the AWS CLI tool on our laptop, and setting up Terraform. We then used Terraform to define a VPC and the network infrastructure within which to deploy EC2 instances. We learned how to use Terraform to automate most of the EC2 configuration details so that we can quickly initialize a Docker swarm.

We learned that a Docker compose file and a Docker stack file are very similar things. The latter is used with Docker Swarm and is a powerful tool for describing the deployment of Docker services.

In the next chapter, we will learn about both unit testing and functional testing. While a core principle of test-driven development is to write the tests before writing the application, we've done it the other way around and put the chapter about unit testing at the end of the book. That's not to say unit testing is unimportant, because it certainly is important.

13

Unit Testing and Functional Testing

Unit testing has become a primary part of good software development practice. It is a method by which individual units of source code are tested to ensure they function properly. Each unit is theoretically the smallest testable part of an application.

In unit testing, each unit is tested separately, isolating the unit under test as much as possible from other parts of the application. If a test fails, you would want it to be due to a bug in your code rather than a bug in the package that your code happens to use. A common technique is to use mock objects or mock data to isolate individual parts of the application from one another.

Functional testing, on the other hand, doesn't try to test individual components. Instead, it tests the whole system. Generally speaking, unit testing is performed by the development team, while functional testing is performed by a **Quality Assurance (QA)** or **Quality Engineering (QE)** team. Both testing models are needed to fully certify an application. An analogy might be that unit testing is similar to ensuring that each word in a sentence is correctly spelled, while functional testing ensures that the paragraph containing that sentence has a good structure.

Writing a book requires not just ensuring the words are correctly spelled, but ensuring that the words string together as useful grammatically correct sentences and chapters that convey the intended meaning. Similarly, a successful software application requires much more than ensuring each "unit" correctly behaves. Does the system as a whole perform the intended actions?

In this chapter, we'll cover the following topics:

- Assertions as the basis of software tests
- The Mocha unit testing framework and the Chai assertions library
- Using tests to find bugs and fix the bug

- Using Docker to manage test infrastructure
- Testing a REST backend service
- UI functional testing in a real web browser using Puppeteer
- Improving UI testability with element ID attributes

By the end of this chapter, you will know how to use Mocha, as well as how to write test cases for both directly invoked code under test and for testing code accessed via REST services. You will have also learned how to use Docker Compose to manage test infrastructure, both on your laptop and on the AWS EC2 Swarm infrastructure from Chapter 12, *Deploying Docker Swarm to AWS EC2 with Terraform*.

That's a lot of territory to cover, so let's get started.

Assert – the basis of testing methodologies

Node.js has a useful built-in testing tool known as the `assert` module. Its functionality is similar to assert libraries in other languages. Namely, it's a collection of functions for testing conditions, and if the conditions indicate an error, the `assert` function throws an exception. It's not a complete test framework by any stretch of the imagination, but it can still be used for some amount of testing.

At its simplest, a test suite is a series of `assert` calls to validate the behavior of the thing being tested. For example, a test suite could instantiate the user authentication service, then make an API call and use `assert` methods to validate the result, then make another API call to validate its results, and so on.

Consider the following code snippet, which you can save in a file named `deleteFile.mjs`:

```
import fs from 'fs';

export function deleteFile(fname, callback) {
    fs.stat(fname, (err, stats) => {
        if (err)
            callback(new Error(`the file ${fname} does not exist`));
        else {
            fs.unlink(fname, err => {
                if (err) callback(new Error(`Could not
                    delete ${fname}`));
                else callback();
            });
        }
    });
}
```

```
    });
  });
}
}
```

The first thing to notice is this contains several layers of asynchronous callback functions. This presents a couple of challenges:

- Capturing errors from deep inside a callback
- Detecting conditions where the callbacks are never called

The following is an example of using `assert` for testing. Create a file named `test-deleteFile.mjs` containing the following:

```
import assert from 'assert';
import { deleteFile } from './deleteFile.mjs';

deleteFile("no-such-file", (err) => {
  assert.ok(err);
  assert.ok(err instanceof Error);
  assert.match(err.message, /does not exist/);
});
```

This is what's called a negative test scenario, in that it's testing whether requesting to delete a nonexistent file throws the correct error. The `deleteFile` function throws an error containing the text that *does not exist* if the file to be deleted does not exist. This test ensures the correct error is thrown and would fail if the wrong error is thrown, or if no error is thrown.

If you are looking for a quick way to test, the `assert` module can be useful when used this way. Each test case would call a function, then use one or more `assert` statements to test the results. In this case, the `assert` statements first ensure that `err` has some kind of value, then ensures that value is an `Error` instance, and finally ensures that the `message` attribute has the expected text. If it runs and no messages are printed, then the test passes. But what happens if the `deleteFile` callback is never called? Will this test case catch that error?

```
$ node test-deleteFile.mjs
```

No news is good news, meaning it ran without messages and therefore the test passed.

The `assert` module is used by many of the test frameworks as a core tool for writing test cases. What the test frameworks do is create a familiar test suite and test case structure to encapsulate your test code, plus create a context in which a series of test cases are robustly executed.

For example, we asked about the error of the callback function never being called. Test frameworks usually have a timeout so that if no result of any kind is supplied within a set number of milliseconds, then the test case is considered an error.

There are many styles of assertion libraries available in Node.js. Later in this chapter, we'll use the Chai assertion library (<http://chaijs.com/>), which gives you a choice between three different assertion styles (`should`, `expect`, and `assert`).

Testing a Notes model

Let's start our unit testing journey with the data models we wrote for the Notes application. Because this is unit testing, the models should be tested separately from the rest of the Notes application.

In the case of most of the Notes models, isolating their dependencies implies creating a mock database. Are you going to test the data model or the underlying database? Mocking out a database means creating a fake database implementation, which does not look like a productive use of our time. You can argue that testing a data model is really about testing the interaction between your code and the database. Since mocking out the database means not testing that interaction, we should test our code against the database engine in order to validate that interaction.

With that line of reasoning in mind, we'll skip mocking out the database, and instead run the tests against a database containing test data. To simplify launching the test database, we'll use Docker to start and stop a version of the Notes application stack that's set up for testing.

Let's start by setting up the tools.

Mocha and Chai – the chosen test tools

If you haven't already done so, duplicate the source tree so that you can use it in this chapter. For example, if you had a directory named `chap12`, create one named `chap13` containing everything from `chap12` to `chap13`.

In the `notes` directory, create a new directory named `test`.

Mocha (<http://mochajs.org/>) is one of many test frameworks available for Node.js. As you'll see shortly, it helps us write test cases and test suites, and it provides a test results reporting mechanism. It was chosen over the alternatives because it supports Promises. It fits very well with the Chai assertion library mentioned earlier.

While in the `notes/test` directory, type the following to install Mocha and Chai:

```
$ npm init
... answer the questions to create package.json
$ npm install mocha@7.x chai@4.2.x cross-env@7.x npm-run-all@4.1.x --
  save-dev
...
```

This, of course, sets up a `package.json` file and installs the required packages.

Beyond Mocha and Chai, we've installed two additional tools. The first, `cross-env`, is one we've used before and it enables cross-platform support for setting environment variables on the command line. The second, `npm-run-all`, simplifies using `package.json` to drive build or test procedures.



For the documentation of `cross-env`, go to <https://www.npmjs.com/package/cross-env>.

For the documentation of `npm-run-all`, go to <https://www.npmjs.com/package/npm-run-all>.

With the tools set up, we can move on to creating tests.

Notes model test suite

Because we have several Notes models, the test suite should run against any model. We can write tests using the `NotesStore` API, and an environment variable should be used to declare the model to test. Therefore, the test script will load `notes-store.mjs` and call functions on the object it supplies. Other environment variables will be used for other configuration settings.

Because we've written the Notes application using ES6 modules, we have a small item to consider. Older Mocha releases only supported running tests in CommonJS modules, so this would require us to jump through a couple of hoops to test Notes modules. But the current release of Mocha does support them, meaning we can freely use ES6 modules.

We'll start by writing a single test case and go through the steps of running that test and getting the results. After that, we'll write several more test cases, and even find a couple of bugs. These bugs will give us a chance to debug the application and fix any problems. We'll close out this section by discussing how to run tests that require us to set up background services, such as a database server.

Creating the initial Notes model test case

In the `test` directory, create a file named `test-model.mjs` containing the following. This will be the outer shell of the test suite:

```
import util from 'util';
import Chai from 'chai';
const assert = Chai.assert;
import { useModel as useNotesModel } from '../models/notes-store.mjs';

var store;

describe('Initialize', function() {
    this.timeout(100000);
    it('should successfully load the model', async function() {
        try {
            // Initialize just as in app.mjs
            // If these execute without exception the test succeeds
            store = await useNotesModel(process.env.NOTES_MODEL);
        } catch (e) {
            console.error(e);
            throw e;
        }
    });
});
```

This loads in the required modules and implements the first test case.

The Chai library supports three flavors of assertions. We're using the `assert` style here, but it's easy to use a different style if you prefer.

For the other assertion styles supported by Chai, see
<http://chaijs.com/guide/styles/>.



Chai's assertions include a very long list of useful assertion functions. For the documentation, see <http://chaijs.com/api/assert/>.

To load the model to be tested, we call the `useModel` function (renamed as `useNotesModel`). You'll remember that this uses the `import()` function to dynamically select the actual NotesStore implementation to use. The `NOTES_MODEL` environment variable is used to select which to load.

Calling `this.timeout` adjusts the time allowed for completing the test. By default, Mocha allows 2,000 milliseconds (2 seconds) for a test case to be completed. This particular test case might take longer than that, so we've given it more time.

The test function is declared as `async`. Mocha can be used in a callback fashion, where Mocha passes in a callback to the test to invoke and indicate errors. However, it can also be used with `async` test functions, meaning that we can throw errors in the normal way and Mocha will automatically capture those errors to determine if the test fails.

Generally, Mocha looks to see if the function throws an exception or whether the test case takes too long to execute (a timeout situation). In either case, Mocha will indicate a test failure. That's, of course, simple to determine for non-asynchronous code. But Node.js is all about asynchronous code, and Mocha has two models for testing asynchronous code. In the first (not seen here), Mocha passes in a callback function, and the test code is to call the callback function. In the second, as seen here, it looks for a Promise being returned by the test function and determines a pass/fail regarding whether the Promise is in the `resolve` or `reject` state.

We are keeping the NotesStore model in the global `store` variable so that it can be used by all tests. The test, in this case, is whether we can load a given NotesStore implementation. As the comment states, if this executes without throwing an exception, the test has succeeded. The other purpose of this test is to initialize the variable for use by other test cases.

It is useful to notice that this code carefully avoids loading `app.mjs`. Instead, it loads the test driver module, `models/notes-store.mjs`, and whatever module is loaded by `useNotesModel`. The NotesStore implementation is what's being tested, and the spirit of unit testing says to isolate it as much as possible.

Before we proceed further, let's talk about how Mocha structures tests.

With Mocha, a test suite is contained within a `describe` block. The first argument is a piece of descriptive text that you use to tailor the presentation of test results. The second argument is a function that contains the contents of the given test suite.

The `it` function is a test case. The intent is for us to read this as *it should successfully load the module*. Then, the code within the function is used to check that assertion.



With Mocha, it is important to not use arrow functions in the `describe` and `it` blocks. By now, you will have grown fond of arrow functions because of how much easier they are to write. However, Mocha calls these functions with a `this` object containing useful functions for Mocha. Because arrow functions avoid setting up a `this` object, Mocha would break.

Now that we have a test case written, let's learn how to run tests.

Running the first test case

Now that we have a test case, let's run the test. In the `package.json` file, add the following `scripts` section:

```
"scripts": {  
  "test-all": "npm-run-all test-notes-memory test-level test-notes-  
    fs test-notes-sqlite3 test-notes-sequelize-sqlite",  
  "test-notes-memory": "cross-env NOTES_MODEL=memory mocha test  
    -model",  
  "test-level": "cross-env NOTES_MODEL=level mocha test-model",  
  "test-notes-fs": "cross-env NOTES_MODEL=fs mocha test-model",  
  "pretest-notes-sqlite3": "rm -f chap13.sqlite3 && sqlite3  
    chap13.sqlite3 --init ../models/schema-sqlite3.sql </dev/null",  
  "test-notes-sqlite3": "cross-env NOTES_MODEL=sqlite3  
    SQLITE_FILE=chap13.sqlite3 mocha test-model",  
  "test-notes-sequelize-sqlite": "cross-env NOTES_MODEL=sequelize  
    SEQUELIZE_CONNECT=sequelize-sqlite.yaml mocha test-model"  
}
```

What we've done here is create a `test-all` script that will run the test suite against the individual NotesStore implementations. We can run this script to run every test combination, or we can run a specific script to test just the one combination. For example, `test-notes-sequelize-sqlite` will run tests against SequelizeNotesStore using the SQLite3 database.

It uses `npm-run-all` to support running the tests in series. Normally, in a `package.json` script, we would write this:

```
"test-all": "npm run test-notes-memory && npm run test-level && npm  
  run test-notes-fs && ..."
```

This runs a series of steps one after another, relying on a feature of the Bash shell. The `npm-run-all` tool serves the same purpose, namely running one package.json script after another in the series. The first advantage is that the code is simpler and more compact, making it easier to read, while the other advantage is that it is cross-platform. We're using `cross-env` for the same purpose so that the test scripts can be executed on Windows as easily as they can be on Linux or macOS.

For the `test-notes-sequelize-sqlite` test, look closely. Here, you can see that we need a database configuration file named `sequelize-sqlite.yaml`. Create that file with the following code:

```
dbname: notestest
username:
password:
params:
  dialect: sqlite
  storage: notestest-sequelize.sqlite3
  logging: false
```

This, as the test script name suggests, uses SQLite3 as the underlying database, storing it in the named file.

We are missing two combinations, `test-notes-sequelize-mysql` for `SequelizerNotesStore` using MySQL and `test-notes-mongodb`, which tests against `MongoDBNotesStore`. We'll implement these combinations later.

Having automated the run of all test combinations, we can try it out:

```
$ npm run test-all

> notes-test@1.0.0 test-all /Users/David/Chapter13/notes/test
> npm-run-all test-notes-memory test-level test-notes-fs test-notes-
sqlite3 test-notes-sequelize-sqlite

> notes-test@1.0.0 test-notes-memory /Users/David/Chapter13/notes/test
> cross-env NOTES_MODEL=memory mocha test-model

  Initialize
    ✓ should successfully load the model

  1 passing (8ms)
  ...
```

If all has gone well, you'll get this result for every test combination currently supported in the `test-all` script.

This completes the first test, which was to demonstrate how to create tests and execute them. All that remains is to write more tests.

Adding some tests

That was easy, but if we want to find what bugs we created, we need to test some functionality. Now, let's create a test suite for testing `NotesStore`, which will contain several test suites for different aspects of `NotesStore`.

What does that mean? Remember that the `describe` function is the container for a test suite and that the `it` function is the container for a test case. By simply nesting `describe` functions, we can contain a test suite within a test suite. It will be clearer what that means after we implement this:

```
describe('Model Test', function() {
  describe('check keylist', function() {
    before(async function() {
      await store.create('n1', 'Note 1', 'Note 1');
      await store.create('n2', 'Note 2', 'Note 2');
      await store.create('n3', 'Note 3', 'Note 3');
    });
    ...
    after(async function() {
      const keyz = await store.keylist();
      for (let key of keyz) {
        await store.destroy(key);
      }
    });
    ...
  });
});
```

Here, we have a `describe` function that defines a test suite containing another `describe` function. That's the structure of a nested test suite.

We do not have test cases in the `it` function defined at the moment, but we do have the `before` and `after` functions. These two functions do what they sound like; namely, the `before` function runs before all the test cases, while the `after` function runs after all the test cases have finished. The `before` function is meant to set up conditions that will be tested, while the `after` function is meant for teardown.

In this case, the `before` function adds entries to `NotesStore`, while the `after` function removes all entries. The idea is to have a clean slate after each nested test suite is executed.

The `before` and `after` functions are what Mocha calls a hook. The other hooks are `beforeEach` and `afterEach`. The difference is that the `Each` hooks are triggered before or after each test case's execution.

These two hooks also serve as test cases since the `create` and `destroy` methods could fail, in which case the hook will fail.

Between the `before` and `after` hook functions, add the following test cases:

```
it("should have three entries", async function() {
  const keyz = await store.keylist();
  assert.exists(keyz);
  assert.isArray(keyz);
  assert.lengthOf(keyz, 3);
});

it("should have keys n1 n2 n3", async function() {
  const keyz = await store.keylist();
  assert.exists(keyz);
  assert.isArray(keyz);
  assert.lengthOf(keyz, 3);
  for (let key of keyz) {
    assert.match(key, /n[123]/, "correct key");
  }
});

it("should have titles Node #", async function() {
  const keyz = await store.keylist();
  assert.exists(keyz);
  assert.isArray(keyz);
  assert.lengthOf(keyz, 3);
  var keyPromises = keyz.map(key => store.read(key));
  const notez = await Promise.all(keyPromises);
  for (let note of notez) {
    assert.match(note.title, /Note [123]/, "correct title");
  }
});
```

As suggested by the description for this test suite, the functions all test the `keylist` method.

For each test case, we start by calling `keylist`, then using `assert` methods to check different aspects of the array that is returned. The idea is to call `NotesStore` API functions, then test the results to check whether they matched the expected results.

Now, we can run the tests and get the following:

```
$ npm run test-all
...
> notes-test@1.0.0 test-notes-fs /Users/David/Chapter13/notes/test
> NOTES_MODEL=fs mocha test-model

  Initialize
    ✓ should successfully load the model (174ms)

  Model Test
    check keylist
      ✓ should have three entries
      ✓ should have keys n1 n2 n3
      ✓ should have titles Node #

  4 passing (226ms)
...
```

Compare the outputs with the descriptive strings in the `describe` and `it` functions. You'll see that the structure of this output matches the structure of the test suites and test cases. In other words, we should structure them so that they have well-structured test output.

As they say, testing is never completed, only exhausted. So, let's see how far we can go before exhausting ourselves.

More tests for the Notes model

That wasn't enough to test much, so let's go ahead and add some more tests:

```
describe('Model Test', function() {
  ...
  describe('read note', function() {
    before(async function() {
      await store.create('n1', 'Note 1', 'Note 1');
    });

    it('should have proper note', async function() {
      const note = await store.read('n1');
      assert.exists(note);
      assert.deepEqual({
        key: note.key, title: note.title, body: note.body
      }, {
        key: 'n1',
        title: 'Note 1',
      });
    });
  });
});
```

```
        body: 'Note 1'
    });
});
it('Unknown note should fail', async function() {
    try {
        const note = await store.read('badkey12');
        assert.notExists(note);
        throw new Error('should not get here');
    } catch(err) {
        // An error is expected, so it is an error if
        // the 'should not get here' error is thrown
        assert.notEqual(err.message, 'should not get here');
    }
});

after(async function() {
    const keyz = await store.keylist();
    for (let key of keyz) {
        await store.destroy(key);
    }
});
...
});
```

These tests check the `read` method. In the first test case, we check whether it successfully reads a known Note, while in the second test case, we have a negative test of what happens if we read a non-existent Note.

Negative tests are very important to ensure that functions fail when they're supposed to fail and that failures are indicated correctly.

The Chai Assertions API includes some very expressive assertions. In this case, we've used the `deepEqual` method, which does a deep comparison of two objects. You'll see that for the first argument, we pass in an object and that for the second, we pass an object that's used to check the first. To see why this is useful, let's force it to indicate an error by inserting `FAIL` into one of the test strings.

After running the tests, we get the following output:

```
> notes-test@1.0.0 test-notes-memory /Users/David/Chapter13/notes/test
> NOTES_MODEL=memory mocha test-model

Initialize
  ✓ should successfully load the model

Model Test
```

```
check keylist
  ✓ should have three entries
  ✓ should have keys n1 n2 n3
  ✓ should have titles Node #
read note
  1) should have proper note
    ✓ Unknown note should fail

5 passing (35ms)
1 failing

1) Model Test
  read note
    should have proper note:

AssertionError: expected { Object (key, title, ...) } to deeply equal
{ Object (key, title, ...) }
+ expected - actual
{
  "body": "Note 1"
  "key": "n1"
- "title": "Note 1"
+ "title": "Note 1 FAIL"
}

at Context.<anonymous>
(file:///Users/David/Chapter13/notes/test/test-model.mjs:76:16)
```

This is what a failed test looks like. Instead of the checkmark, there is a number, and the number corresponds to a report below it. In the failure report, the `deepEqual` function gave us clear information about how the object fields differed. In this case, it is the test we forced to fail because we wanted to see how the `deepEqual` function works.

Notice that for the negative tests – where the test passes if an error is thrown – we run it in a `try/catch` block. The `throw new Error` line in each case should not execute because the preceding code should throw an error. Therefore, we can check if the message in that thrown error is the message that arrives, and fail the test if that's the case.

Diagnosing test failures

We can add more tests because, obviously, these tests are not sufficient to be able to ship Notes to the public. After doing so, and then running the tests against the different test combinations, we will find this result for the SQLite3 combination:

```
$ npm run test-notes-sqlite3

> notes-test@1.0.0 test-notes-sqlite3
/Users/David/Chapter11/notes/test
> rm -f chap11.sqlite3 && sqlite3 chap11.sqlite3 --init
./models/schema-sqlite3.sql </dev/null && NOTES_MODEL=sqlite3
SQLITE_FILE=chap11.sqlite3 mocha test-model

Initialize
  ✓ should successfully load the model (89ms)

Model Test
  check keylist
    ✓ should have three entries
    ✓ should have keys n1 n2 n3
    ✓ should have titles Node #
  read note
    ✓ should have proper note
    1) Unknown note should fail
  change note
    ✓ after a successful model.update
  destroy note
    ✓ should remove note
    2) should fail to remove unknown note

7 passing (183ms)
2 failing

1) Model Test
  read note
  Unknown note should fail:
  Uncaught TypeError: Cannot read property 'notekey' of undefined
  at Statement.<anonymous>
  (file:///Users/David/Chapter11/notes/models/notes-sqlite3.mjs:79:43)

2) Model Test
  destroy note
  should fail to remove unknown note:
  AssertionError: expected 'should not get here' to not equal 'should
    not get here'
  + expected - actual
  at Context.<anonymous>
```

```
(file:///Users/David/Chapter11/notes/test/test-
model.mjs:152:20)
```

Our test suite found two errors, one of which is the error we mentioned in Chapter 7, *Data Storage and Retrieval*. Both failures came from the negative test cases. In one case, the test calls `store.read("badkey12")`, while in the other, it calls `store.delete("badkey12")`.

It is easy enough to insert `console.log` calls and learn what is going on.

For the `read` method, SQLite3 gave us `undefined` for `row`. The test suite successfully calls the `read` function multiple times with a `notekey` value that does exist. Obviously, the failure is limited to the case of an invalid `notekey` value. In such cases, the query gives an empty result set and SQLite3 invokes the callback with `undefined` in both the error and the `row` values. Indeed, the equivalent SQL `SELECT` statement does not throw an error; it simply returns an empty result set. An empty result set isn't an error, so we received no error and an `undefined` `row`.

However, we defined `read` to throw an error if no such Note exists. This means this function must be written to detect this condition and throw an error.

There is a difference between the `read` functions in `models/notes-sqlite3.mjs` and `models/notes-sequelize.mjs`. On the day we wrote `SequelizeNotesStore`, we must have thought through this function more carefully than we did on the day we wrote `SQLITE3NotesStore`. In `SequelizeNotesStore.read`, there is an error that's thrown when we receive an empty result set, and it has a check that we can adapt. Let's rewrite the `read` function in `models/notes-sqlite.mjs` so that it reads as follows:

```
async read(key) {
  var db = await connectDB();
  var note = await new Promise((resolve, reject) => {
    db.get("SELECT * FROM notes WHERE notekey = ?", [key], (err, row) => {
      if (err) return reject(err);
      if (!row) {
        reject(new Error(`No note found for ${key}`));
      } else {
        const note = new Note(row.notekey, row.title, row.body);
        resolve(note);
      }
    });
  });
  return note;
}
```

If this receives an empty result, an error is thrown. While the database doesn't see empty results set as an error, Notes does. Furthermore, Notes already knows how to deal with a thrown error in this case. Make this change and that particular test case will pass.

There is a second similar error in the `destroy` logic. In SQL, it obviously is not an SQL error if this SQL (from `models/notes-sqlite3.mjs`) does not delete anything:

```
db.run("DELETE FROM notes WHERE notekey = ?;", ...);
```

Unfortunately, there isn't a method in the SQL option to fail if it does not delete any records. Therefore, we must add a check to see if a record exists, namely the following:

```
async destroy(key) {
  var db = await connectDB();
  const note = await this.read(key);
  return await new Promise((resolve, reject) => {
    db.run("DELETE FROM notes WHERE notekey = ?;",
      [ key ], err => {
        if (err) return reject(err);
        this.emitDestroyed(key);
        resolve();
      });
  });
}
```

Therefore, we read the note and, as a byproduct, we verify the note exists. If the note doesn't exist, `read` will throw an error, and the `DELETE` operation will not even run.

When we run `test-notes-sequelize-sqlite`, there is also a similar failure in its `destroy` method. In `models/notes-sequelize.mjs`, make the following change:

```
async destroy(key) {
  await connectDB();
  const note = await SQNote.findOne({ where: { notekey: key } });
  if (!note) {
    throw new Error(`No note found for ${key}`);
  } else {
    await SQNote.destroy({ where: { notekey: key } });
  }
  this.emitDestroyed(key);
}
```

This is the same change; that is, to first `read` the Note corresponding to the given `key`, and if the Note does not exist, to throw an error.

Likewise, when running `test-level`, we get a similar failure, and the solution is to edit `models/notes-level.mjs` to make the following change:

```
async destroy(key) {
  const db = await connectDB();
  const note = Note.fromJSON(await db.get(key));
  await db.del(key);
  this.emitDestroyed(key);
}
```

As with the other NotesStore implementations, this reads the Note before trying to destroy it. If the `read` operation fails, then the test case sees the expected error.

These are the bugs we referred to in Chapter 7, *Data Storage and Retrieval*. We simply forgot to check for these conditions in this particular model. Thankfully, our diligent testing caught the problem. At least, that's the story to tell the managers rather than telling them that we forgot to check for something we already knew could happen.

Testing against databases that require server setup – MySQL and MongoDB

That was good, but we obviously won't run Notes in production with a database such as SQLite3 or Level. We can run Notes against the SQL databases supported by Sequelize (such as MySQL) and against MongoDB. Clearly, we've been remiss in not testing those two combinations.

Our test results matrix reads as follows:

- notes-fs: PASS
- notes-memory: PASS
- notes-level: 1 failure, now fixed
- notes-sqlite3: 2 failures, now fixed
- notes-sequelize: With SQLite3: 1 failure, now fixed
- notes-sequelize: With MySQL: untested
- notes-mongodb: Untested

The two untested NotesStore implementations both require that we set up a database server. We avoided testing these combinations, but our manager won't accept that excuse because the CEO needs to know we've completed the test cycles. Notes must be tested with a configuration similar to the production environments'.

In production, we'll be using a regular database server, with MySQL or MongoDB being the primary choices. Therefore, we need a way to incur a low overhead to run tests against those databases. Testing against the production configuration must be so easy that we should feel no resistance in doing so, to ensure that tests are run often enough to make the desired impact.

In this section, we made a lot of progress and have a decent start on a test suite for the NotesStore database modules. We learned how to set up test suites and test cases in Mocha, as well as how to get useful test reporting. We learned how to use `package.json` to drive test suite execution. We also learned about negative test scenarios and how to diagnose errors that come up.

But we need to work on this issue of testing against a database server. Fortunately, we've already worked with a piece of technology that supports easily creating and destroying the deployment infrastructure. Hello, Docker!

In the next section, we'll learn how to repurpose the Docker Compose deployment as a test infrastructure.

Using Docker Swarm to manage test infrastructure

One advantage Docker gives is the ability to install the production environment on our laptop. In Chapter 12, *Deploying Docker Swarm to AWS EC2 Using Terraform*, we converted a Docker setup that ran on our laptop so that it could be deployed on real cloud hosting infrastructure. That relied on converting a Docker Compose file into a Docker Stack file, along with customization for the environment we built on AWS EC2 instances.

In this section, we'll repurpose the Stack file as test infrastructure deployed to a Docker Swarm. One approach is to simply run the same deployment, to AWS EC2, and substitute new values for the `var.project_name` and `var.vpc_name` variables. In other words, the EC2 infrastructure could be deployed this way:

```
$ terraform apply --var project_name=notes-test --var vpc_name=notes-test-vpc
```

This would deploy a second VPC with a different name that's explicitly for test execution and that would not disturb the production deployment. It's quite common in Terraform to customize the deployment this way for different targets.

In this section, we'll try something different. We can use Docker Swarm in other contexts, not just the AWS EC2 infrastructure we set up. Specifically, it is easy to use Docker Swarm with the Docker for Windows or Docker for macOS that's running on our laptop.

What we'll do is configure Docker on our laptop so that it supports swarm mode and create a slightly modified version of the Stack file in order to run the tests on our laptop. This will solve the issue of running tests against a MySQL database server, and also lets us test the long-neglected MongoDB module. This will demonstrate how to use Docker Swarm for test infrastructure and how to perform semi-automated test execution inside the containers using a shell script.

Let's get started.

Using Docker Swarm to deploy test infrastructure

We had a great experience using Docker Compose and Swarm to orchestrate Notes application deployment on both our laptop and our AWS infrastructure. The whole system, with five independent services, is easily described in `compose-local/docker-compose.yml` and `compose-swarm/docker-compose.yml`. What we'll do is duplicate the Stack file, then make a couple of small changes required to support test execution in a local swarm.

To configure the Docker installation on our laptop for swarm mode, simply type the following:

```
$ docker swarm init
```

As before, this will print a message about the join token. If desired, if you have multiple computers in your office, it might be interesting for you to experiment with setting up a local Swarm. But for this exercise, that's not important. This is because we can do everything required with a single-node Swarm.

This isn't a one-way street, meaning that when you're done with this exercise, it is easy to turn off swarm mode. Simply shut down anything deployed to your local Swarm and run the following command:

```
$ docker swarm leave --force
```

Normally, this is used for a host that you wish to detach from an existing swarm. If there is only one host remaining in a swarm, the effect will be to shut down the swarm.

Now that we know how to initialize swarm mode on our laptop, let's set about creating a stack file suitable for use on our laptop.

Create a new directory, `compose-stack-test-local`, as a sibling to the `notes`, `users`, and `compose-local` directories. Copy `compose-stack/docker-compose.yml` to that directory. We'll be making several small changes to this file and no changes to the existing Dockerfiles. As much as it is possible, it is important to test the same containers that are used in the production deployment. This means it's acceptable to inject test files into the containers, but not modify them.

Make every `deploy` tag look like this:

```
deploy:  
  replicas: 1
```

This deletes the placement constraints we declared for use on AWS EC2 and sets it to one replica for each service. For a single-node cluster, we don't worry about placement, of course, and there is no need for more than one instance of any service.

For the database services, remove the `volumes` tag. Using this tag is required when it's necessary to persist in the database data directory. For test infrastructure, the data directory is unimportant and can be thrown away at will. Likewise, remove the top-level `volumes` tag.

For the `svc-notes` and `svc-userauth` services, make these changes:

```
services:  
  ...  
  svc-userauth:  
    image: compose-stack-test-local/svc-userauth  
    ...  
    ports:  
      - "5858:5858"  
    ...  
    environment:
```

```
SEQUELIZE_CONNECT: sequelize-docker-mysql.yaml
SEQUELIZE_DBHOST: db-userauth
...
svc-notes:
  image: compose-stack-test-local/svc-notes
  ...
  volumes:
    - type: bind
      source: ../notes/test
      target: /notesapp/test
    - type: bind
      source: ../notes/models/schema-sqlite3.sql
      target: /notesapp/models/schema-sqlite3.sql
  ports:
    - "3000:3000"
  environment:
    ...
    TWITTER_CALLBACK_HOST: "http://localhost:3000"
    SEQUELIZE_CONNECT: models/sequelize-docker-mysql.yaml
    SEQUELIZE_DBHOST: db-notes
    NOTES_MODEL: sequelize
  ...

```

This injects the files required for testing into the `svc-notes` container. Obviously, this is the `test` directory that we created in the previous section for the Notes service. Those tests also require the SQLite3 schema file since it is used by the corresponding test script. In both cases, we can use bind mounts to inject the files into the running container.

The Notes test suite follows a normal practice for Node.js projects of putting test files in the test directory. When building the container, we obviously don't include the test files because they're not required for deployment. But running tests requires having that directory inside the running container. Fortunately, Docker makes this easy. We simply mount the directory into the correct place.

The bottom line is this approach gives us the following advantages:

- The test code is in `notes/test`, where it belongs.
 - The test code is not copied into the production container.
 - In test mode, the `test` directory appears where it belongs.

For Docker (using `docker run`) and Docker Compose, the volume is mounted from a directory on the localhost. But for swarm mode, with a multi-node swarm, the container could be deployed on any host matching the placement constraints we declare. In a swarm, bind volume mounts like the ones shown here will try to mount from a directory on the host that the container has been deployed in. But we are not using a multi-node swarm; instead, we are using a single-node swarm. Therefore, the container will mount the named directory from our laptop, and all will be fine. But as soon as we decide to run testing on a multi-node swarm, we'll need to come up with a different strategy for injecting these files into the container.

We've also changed the `ports` mappings. For `svc-userauth`, we've made its port visible to give ourselves the option of testing the REST service from the host computer. For the `svc-notes` service, this will make it appear on port 3000. In the environment section, make sure you did not set a `PORT` variable. Finally, we adjust `TWITTER_CALLBACK_HOST` so that it uses `localhost:3000` since we're deploying on the localhost.

For both services, we're changing the image tag from the one associated with the AWS ECR repository to one of our own designs. We won't be publishing these images to an image repository, so we can use any image tag we like.

For both services, we are using the Sequelize data model, using the existing MySQL-oriented configuration file, and setting the `SEQUELIZE_DBHOST` variable to refer to the container holding the database.

We've defined a Docker Stack file that should be useful for deploying the Notes application stack in a Swarm. The difference between the deployment on AWS EC2 and here is simply the configuration. With a few simple configuration changes, we've mounted test files into the appropriate container, reconfigured the volumes and the environment variables, and changed the deployment descriptors so that they're suitable for a single-node swarm running on our laptop.

Let's deploy this and see how well we did.

Executing tests under Docker Swarm

We've repurposed our Docker Stack file so that it describes deploying to a single-node swarm, ensuring the containers are set up to be useful for testing. Our next step is to deploy the Stack to a swarm and execute the tests inside the Notes container.

To set it up, run the following commands:

```
$ docker swarm init
... ignore the output showing the docker swarm join command

$ printf '....' | docker secret create TWITTER_CONSUMER_SECRET -
$ printf '....' | docker secret create TWITTER_CONSUMER_KEY -
```

We run `swarm init` to turn on swarm mode on our laptop, then add the two TWITTER secrets to the swarm. Since it is a single-node swarm, we don't need to run a `docker swarm join` command to add new nodes to the swarm.

Then, in the `compose-stack-test-local` directory, we can run these commands:

```
$ docker-compose build
...
Building svc-userauth
...
Successfully built 876860f15968
Successfully tagged compose-stack-test-local/svc-userauth:latest
Building svc-notes
...
Successfully built 1c4651c37a86
Successfully tagged compose-stack-test-local/svc-notes:latest

$ docker stack deploy --compose-file docker-compose.yml notes
Ignoring unsupported options: build, restart
...
Creating network notes_authnet
Creating network notes_svchnet
Creating network notes_frontnet
Creating service notes_db-userauth
Creating service notes_svc-userauth
Creating service notes_db-notes
Creating service notes_svc-notes
Creating service notes_redis
```

Because a Stack file is also a Compose file, we can run `docker-compose build` to build the images. Because of the `image` tags, this will automatically tag the images so that they match the image names we specified.

Then, we use `docker stack deploy`, as we did when deploying to AWS EC2. Unlike the AWS deployment, we do not need to push the images to repositories, which means we do not need to use the `--with-registry-auth` option. This will behave almost identically to the swarm we deployed to EC2, so we explore the deployed services in the same way:

```
$ docker service ls
... output of current services
$ docker service ps notes_svc-notes
... status information for the named service
$ docker ps
... running container list for local host
```

Because this is a single-host swarm, we don't need to use SSH to access the swarm nodes, nor do we need to set up remote access using `docker context`. Instead, we run the Docker commands, and they act on the Docker instance on the localhost.

The `docker ps` command will tell us the precise container name for each service. With that knowledge, we can run the following to gain access:

```
$ docker exec -it notes_svc-notes.1.c8ojirrbv2sfbva91505s3nv bash
root@265672675de1:/notesapp#
root@265672675de1:/notesapp# cd test
root@265672675de1:/notesapp/test# apt-get -y install sqlite3
...
root@265672675de1:/notesapp/test# rm -rf node_modules/
root@265672675de1:/notesapp/test# npm install
...
```

Because, in swarm mode, the containers have unique names, we have to run `docker ps` to get the container name, then paste it into this command to start a Bash shell inside the container.

Inside the container, we see the `test` directory is there as expected. But we have a couple of setup steps to perform. The first is to install the SQLite3 command-line tools since the scripts in `package.json` use that command. The second is to remove any existing `node_modules` directory because we don't know if it was built for this container or for the laptop. After that, we need to run `npm install` to install the dependencies.

Having done this, we can run the tests:

```
root@265672675de1:/notesapp/test# npm run test-all
...
```

The tests should execute as they did on our laptop, but they're running inside the container instead. However, the MySQL test won't have run because the package.json scripts are not set up to run that one automatically. Therefore, we can add this to package.json:

```
"test-notes-sequelize-mysql": "cross-env NOTES_MODEL=sequelize
  SEQUELIZE_CONNECT=../models/sequelize-docker-mysql.yaml
  SEQUELIZE_DBHOST=db-notes mocha test-model"
```

This is the command that's required to execute the test suite against the MySQL database.

Then, we can run the tests against MySQL, like so:

```
root@265672675de1:/notesapp/test# npm run test-notes-sequelize-mysql
...
```

The tests should execute correctly against MySQL.

To automate this, we can create a file named run.sh containing the following code:

```
#!/bin/sh
SVC_NOTES=$1
# docker exec -it ${SVC_NOTES} apt-get -y install sqlite3
docker exec -it --workdir /notesapp/test -e DEBUG= ${SVC_NOTES} \
  rm -rf node_modules
docker exec -it --workdir /notesapp/test -e DEBUG= ${SVC_NOTES} \
  npm install
docker exec -it --workdir /notesapp/test -e DEBUG= ${SVC_NOTES} \
  npm run test-notes-memory
docker exec -it --workdir /notesapp/test -e DEBUG= ${SVC_NOTES} \
  npm run test-notes-fs
docker exec -it --workdir /notesapp/test -e DEBUG= ${SVC_NOTES} \
  npm run test-level
docker exec -it --workdir /notesapp/test -e DEBUG= ${SVC_NOTES} \
  npm run test-notes-sqlite3
docker exec -it --workdir /notesapp/test -e DEBUG= ${SVC_NOTES} \
  npm run test-notes-sequelize-sqlite
docker exec -it --workdir /notesapp/test -e DEBUG= ${SVC_NOTES} \
  npm run test-notes-sequelize-mysql
# docker exec -it --workdir /notesapp/test -e DEBUG= ${SVC_NOTES} \
  npm run test-notes-mongodb
```

The script executes each script in `notes/test/package.json` individually. If you prefer, you can replace these with a single line that executes `npm run test-all`.

This script takes a command-line argument for the container name holding the `svc-notes` service. Since the tests are located in that container, that's where the tests must be run. The script can be executed like so:

```
$ sh run.sh notes_svc-notes.1.c8ojirrbvv2sfbva91505s3nv
```

This runs the preceding script, which will run each test combination individually and also make sure the `DEBUG` variable is not set. This variable is set in the Dockerfile and causes debugging information to be printed among the test results output. Inside the script, the `--workdir` option sets the current directory of the command's execution in the `test` directory to simplify running the test scripts.

Of course, this script won't execute as-is on Windows. To convert this for use on PowerShell, save the text starting at the second line into `run.ps1`, and then change `SVC_NOTES` references into `%SVC_NOTES%` references.

We have succeeded in semi-automating test execution for most of our test matrix. However, there is a glaring hole in the test matrix, namely the lack of testing on MongoDB. Plugging that hole will let us see how we can set up MongoDB under Docker.

MongoDB setup under Docker and testing Notes against MongoDB

In Chapter 7, *Data Storage and Retrieval*, we developed MongoDB support for Notes. Since then, we've focused on Sequelize. To make up for that slight, let's make sure we at least test our MongoDB support. Testing on MongoDB simply requires defining a container for the MongoDB database and a little bit of configuration.

Visit https://hub.docker.com/_/mongo/ for the official MongoDB container. You'll be able to retrofit this in order to deploy the Notes application running on MongoDB.

Add the following code to `compose-stack-test-local/docker-compose.yml`:

```
# Uncomment this for testing MongoDB
db-notes-mongo:
  image: mongo:4.2
  container_name: db-notes-mongo
  networks:
    - frontnet
```

```
# volumes:  
# - ./db-notes-mongo:/data/db
```

That's all that's required to add a MongoDB container to a Docker Compose/Stack file. We've connected it to `frontnet` so that the database is accessible by `svc-notes`. If we wanted the `svc-notes` container to use MongoDB, we'd need some environment variables (`MONGO_URL`, `MONGO_DBNAME`, and `NOTES_MODEL`) to tell Notes to use MongoDB.

But we'd also run into a problem that we created for ourselves in [Chapter 9, Dynamic Client/Server Interaction with Socket.IO](#). In that chapter, we created a messaging subsystem so that our users can leave messages for each other. That messaging system is currently implemented to store messages in the same Sequelize database where the Notes are stored. But to run Notes with no Sequelize database would mean a failure in the messaging system. Obviously, the messaging system can be rewritten, for instance, to allow storage in a MongoDB database, or to support running both MongoDB and Sequelize at the same time.

Because we were careful, we can execute code in `models/notes-mongodb.mjs` without it being affected by other code. With that in mind, we'll simply execute the Notes test suite against MongoDB and report the results.

Then, in `notes/test/package.json`, we can add a line to facilitate running tests on MongoDB:

```
"test-notes-mongodb": "cross-env MONGO_URL=mongodb://db-notes-mongo/  
MONGO_DBNAME=chap13-test NOTES_MODEL=mongodb mocha --no-timeouts  
test-  
model"
```

We simply added the MongoDB container to `frontnet`, making the database available at the URL shown here. Hence, it's simple to now run the test suite using the Notes MongoDB model.

The `--no-timeouts` option was necessary to avoid a spurious error while testing the suite against MongoDB. This option instructs Mocha to not check whether a test case execution takes too long.

The final requirement is to add the following line to `run.sh` (or `run.ps1` for Windows):

```
docker exec -it --workdir /notesapp/test -e DEBUG= notes-test \  
npm run test-notes-mongodb
```

This ensures MongoDB can be tested alongside the other test combinations. But when we run this, an error might crop up:

```
(node:475) DeprecationWarning: current Server Discovery and Monitoring engine is deprecated, and will be removed in a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to the MongoClient constructor.
```

The problem is that the initializer for the MongoClient object has changed slightly. Therefore, we must modify `notes/models/notes-mongodb.mjs` with this new `connectDB` function:

```
const connectDB = async () => {
  if (!client) {
    client = await MongoClient.connect(process.env.MONGO_URL, {
      useNewUrlParser: true, useUnifiedTopology: true
    });
  }
}
```

This adds a pair of useful configuration options, including the option explicitly named in the error message. Otherwise, the code is unchanged.

To make sure the container is running with the updated code, rerun the `docker-compose build` and `docker stack deploy` steps shown earlier. Doing so rebuilds the images, and then updates the services. Because the `svc-notes` container will relaunch, you'll need to install the Ubuntu `sqlite3` package again.

Once you've done that, the tests will all execute correctly, including the MongoDB combination.

We can now report the final test results matrix to the manager:

- `models-fs`: PASS
- `models-memory`: PASS
- `models-levelup`: 1 failure, now fixed, PASS
- `models-sqlite3`: Two failures, now fixed, PASS
- `models-sequelize` with SQLite3: 1 failure, now fixed, PASS
- `models-sequelize` with MySQL: PASS
- `models-mongodb`: PASS

The manager will tell you "good job" and then remember that the models are only a portion of the Notes application. We've left two areas completely untested:

- The REST API for the user authentication service
- Functional testing of the user interface

In this section, we've learned how to repurpose a Docker Stack file so that we can launch the Notes stack on our laptop. It took a few simple reconfigurations of the Stack file and we were ready to go, and we even injected the files that are useful for testing. With a little bit more work, we finished testing against all configuration combinations of the Notes database modules.

Our next task is to handle testing the REST API for the user authentication service.

Testing REST backend services

It's now time to turn our attention to the user authentication service. We've mentioned testing this service, saying that we'll get to them later. We developed a command-line tool for both administration and ad hoc testing. While that has been useful all along, it's time to get cracking with some real tests.

There's a question of which tool to use for testing the authentication service. Mocha does a good job of organizing a series of test cases, and we should reuse it here. But the thing we have to test is a REST service. The customer of this service, the Notes application, uses it through the REST API, giving us a perfect rationalization to test the REST interface rather than calling the functions directly. Our ad hoc scripts used the SuperAgent library to simplify making REST API calls. There happens to be a companion library, SuperTest, that is meant for REST API testing. It's easy to use that library within a Mocha test suite, so let's take that route.



For the documentation on SuperTest, look here: <https://www.npmjs.com/package/supertest>.

Create a directory named `compose-stack-test-local/userauth`. This directory will contain a test suite for the user authentication REST service. In that directory, create a file named `test.mjs` that contains the following code:

```
import Chai from 'chai';
const assert = Chai.assert;
import supertest from 'supertest';
```

```
const request = supertest(process.env.URL_USERS_TEST);
const authUser = 'them';
const authKey = 'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF';

describe('Users Test', function() {
  ...
});
```

This sets up Mocha and the SuperTest client. The URL_USERS_TEST environment variable specifies the base URL of the server to run the test against. You'll almost certainly be using `http://localhost:5858`, given the configuration we've used earlier, but it can be any URL pointing to any host. SuperTest initializes itself a little differently to SuperAgent.

The SuperTest module supplies a function, and we call that function with the URL_USERS_TEST variable. That gives us an object, which we call `request`, that is used for interacting with the service under test.

We've also set up a pair of variables to store the authentication user ID and key. These are the same values that are in the user authentication server. We simply need to supply them when making API calls.

Finally, there's the outer shell of the Mocha test suite. So, let's start filling in the before and after test cases:

```
before(async function() {
  await request
    .post('/create-user')
    .send({
      username: "me", password: "w0rd", provider: "local",
      familyName: "Einarrsdottir", givenName: "Ashildr",
      middleName: "",
      emails: [], photos: []
    })
    .set('Content-Type', 'application/json')
    .set('Accept', 'application/json')
    .auth(authUser, authKey);
});

after(async function() {
  await request
    .delete('/destroy/me')
    .set('Content-Type', 'application/json')
    .set('Accept', 'application/json')
    .auth(authUser, authKey);
});
```

These are our `before` and `after` tests. We'll use them to establish a user and then clean them up by removing the user at the end.

This gives us a taste of how the SuperTest API works. If you refer back to `cli.mjs`, you'll see the similarities to `SuperAgent`.

The `post` and `delete` methods we can see here declare the HTTP verb to use. The `send` method provides an object for the POST operation. The `set` method sets header values, while the `auth` method sets up authentication:

```
describe('List user', function() {
  it("list created users", async function() {
    const res = await request.get('/list')
      .set('Content-Type', 'application/json')
      .set('Accept', 'application/json')
      .auth(authUser, authKey);
    assert.exists(res.body);
    assert.isArray(res.body);
    assert.lengthOf(res.body, 1);
    assert.deepEqual(res.body[0], {
      username: "me", id: "me", provider: "local",
      familyName: "Einarssdottir", givenName: "Ashildr",
      middleName: "",
      emails: [], photos: []
    });
  });
});
```

Now, we can test some API methods, such as the `/list` operation.

We have already guaranteed that there is an account in the `before` method, so `/list` should give us an array with one entry.

This follows the general pattern for using Mocha to test a REST API method. First, we use SuperTest's `request` object to call the API method and `await` its result. Once we have the result, we use `assert` methods to validate it is what's expected.

Add the following test cases:

```
describe('find user', function() {
  it("find created users", async function() {
    const res = await request.get('/find/me')
      .set('Content-Type', 'application/json')
      .set('Accept', 'application/json')
      .auth(authUser, authKey);
    assert.exists(res.body);
```

```
        assert.isObject(res.body);
        assert.deepEqual(res.body, {
            username: "me", id: "me", provider: "local",
            familyName: "Einarssdottir", givenName: "Ashildr",
            middleName: "",
            emails: [], photos: []
        });
    });
it('fail to find non-existent users', async function() {
    var res;
    try {
        res = await request.get('/find/nonExistentUser')
            .set('Content-Type', 'application/json')
            .set('Accept', 'application/json')
            .auth(authUser, authKey);
    } catch(e) {
        return; // Test is okay in this case
    }
    assert.exists(res.body);
    assert.isObject(res.body);
    assert.deepEqual(res.body, {});
});
});
```

We are checking the /find operation in two ways:

- **Positive test:** Looking for the account we know exists – failure is indicated if the user account is not found
- **Negative test:** Looking for the one we know does not exist – failure is indicated if we receive something other than an error or an empty object

Add the following test case:

```
describe('delete user', function() {
    it('delete nonexistent users', async function() {
        let res;
        try {
            res = await request.delete('/destroy/nonExistentUser')
                .set('Content-Type', 'application/json')
                .set('Accept', 'application/json')
                .auth(authUser, authKey);
        } catch(e) {
            return; // Test is okay in this case
        }
        assert.exists(res);
        assert.exists(res.error);
        assert.notEqual(res.status, 200);
    });
});
```

```
});  
});
```

Finally, we should check the /destroy operation. This operation is already checked the after method, where we destroy a known user account. We also need to perform the negative test and verify its behavior against an account we know does not exist.

The desired behavior is that either an error is thrown or the result shows an HTTP status indicating an error. In fact, the current authentication server code gives a 500 status code, along with some other information.

This gives us enough tests to move forward and automate the test run.

In compose-stack-test-local/docker-compose.yml, we need to inject the test.js script into the svc-userauth-test container. We'll add that here:

```
svc-userauth-test:  
...  
volumes:  
  - type: bind  
    source: ./userauth  
    target: /userauth/test
```

This injects the userauth directory into the container as the /userauth/test directory. As we did previously, we then must get into the container and run the test script.

The next step is creating a package.json file to hold any dependencies and a script to run the test:

```
{  
  "name": "userauth-test",  
  "version": "1.0.0",  
  "description": "Test suite for user authentication server",  
  "scripts": {  
    "test": "cross-env URL_USERS_TEST=http://localhost:5858 mocha  
    test.mjs"  
  },  
  "dependencies": {  
    "chai": "^4.2.0",  
    "mocha": "^7.1.1",  
    "supertest": "^4.0.2",  
    "cross-env": "^7.0.2"  
  }  
}
```

In the dependencies, we list Mocha, Chai, SuperTest, and cross-env. Then, in the `test` script, we run Mocha along with the required environment variable. This should run the tests.

We could use this test suite from our laptop. Because the test directory is injected into the container the tests, we can also run them inside the container. To do so, add the following code to `run.sh`:

```
SVC_USERAUTH=$2
...
docker exec -it -e DEBUG= --workdir /userauth/test ${SVC_USERAUTH} \
    rm -rf node_modules
docker exec -it -e DEBUG= --workdir /userauth/test ${SVC_USERAUTH} \
    npm install
docker exec -it -e DEBUG= --workdir /userauth/test ${SVC_USERAUTH} \
    npm run test
```

This adds a second argument – in this case, the container name for `svc-userauth`. We can then run the test suite, using this script to run them inside the container. The first two commands ensure the installed packages were installed for the operating system in this container, while the last runs the test suite.

Now, if you run the `run.sh` test script, you'll see the required packages get installed. Then, the test suite will be executed.

The result will look like this:

```
$ sh run.sh notes_svc-notes.1.c8ojirrbvv2sfbva91505s3nv notes_svc-
userauth.1.puos4jqocjji47vpcp9nrakmy
...
> userauth-test@1.0.0 test /userauth/test
> cross-env URL_USERS_TEST=http://localhost:5858 mocha test.mjs

  Users Test
    List user
      ✓ list created users
    find user
      ✓ find created users
      ✓ fail to find non-existent users
    delete user
      ✓ delete nonexistent users

  4 passing (312ms)
```

Because `URL_USERS_TEST` can take any URL, we could run the test suite against any instance of the user authentication service. For example, we could test an instance deployed on AWS EC2 from our laptop using a suitable value for `URL_USERS_TEST`.

We're making good progress. We now have test suites for both the Notes and User Authentication services. We have learned how to test a REST service using the REST API. This is different than directly calling internal functions because it is an end-to-end test of the complete system, in the role of a consumer of the service.

Our next task is to automate test results reporting.

Automating test results reporting

It's cool we have automated test execution, and Mocha makes the test results look nice with all those checkmarks. But what if management wants a graph of test failure trends over time? There could be any number of reasons to report test results as data rather than as a user-friendly printout on the console.

For example, tests are often not run on a developer laptop or by a quality team tester, but by automated background systems. The CI/CD model is widely used, in which tests are run by the CI/CD system on every commit to the shared code repository. When fully implemented, if the tests all pass on a particular commit, then the system is automatically deployed to a server, possibly the production servers. In such a circumstance, the user-friendly test result report is not useful, and instead, it must be delivered as data that can be displayed on a CI/CD results dashboard website.

Mocha uses what's called a **Reporter** to report test results. A Mocha Reporter is a module that prints data in whatever format it supports. More information on this can be found on the Mocha website: <https://mochajs.org/#reporters>.

You will find the current list of available reporters like so:

```
# mocha --reporters

dot - dot matrix
doc - html documentation
spec - hierarchical spec list
json - single json object
progress - progress bar
list - spec-style listing
tap - testanything-protocol
...
```

Then, you can use a specific Reporter, like so:

```
root@df3e8a7561a7:/userauth/test# npm run test -- --reporter tap

> userauth-test@1.0.0 test /userauth/test
> cross-env URL_USERS_TEST=http://localhost:5858 mocha test.mjs "--
  reporter" "tap"

1..4
ok 1 Users Test List user list created users
ok 2 Users Test find user find created users
ok 3 Users Test find user fail to find non-existent users
ok 4 Users Test delete user delete nonexistent users
# tests 4
# pass 4
# fail 0
```

In the `npm run script-name` command, we can inject command-line arguments, as we've done here. The `--` token tells npm to append the remainder of its command line to the command that is executed. The effect is as if we had run this:

```
root@df3e8a7561a7:/userauth/test# URL_USERS_TEST=http://localhost:5858
mocha test.mjs "--reporter" "tap"
```

For Mocha, the `--reporter` option selects which Reporter to use. In this case, we selected the TAP reporter, and the output follows that format.

Test Anything Protocol (TAP) is a widely used test results format that increases the possibility of finding higher-level reporting tools. Obviously, the next step would be to save the results into a file somewhere, after mounting a host directory into the container.

In this section, we learned about the test results reporting formats supported by Mocha. This will give you a starting point for collecting long-term results tracking and other useful software quality metrics. Often, software teams rely on quality metrics trends as part of deciding whether a product can be shipped to the public.

In the next section, we'll round off our tour of testing methodologies by learning about a framework for frontend testing.

Frontend headless browser testing with Puppeteer

A big cost area in testing is manual user interface testing. Therefore, a wide range of tools has been developed to automate running tests at the HTTP level. Selenium is a popular tool implemented in Java, for example. In the Node.js world, we have a few interesting choices. The *chai-http* plugin to Chai would let us interact at the HTTP level with the Notes application while staying within the now-familiar Chai environment.

However, in this section, we'll use Puppeteer (<https://github.com/GoogleChrome/puppeteer>). This tool is a high-level Node.js module used to control a headless Chrome or Chromium browser, using the DevTools protocol. This protocol allows tools to instrument, inspect, debug, and profile Chromium or Chrome browser instances. The key result is that we can test the Notes application in a real browser so that we have greater assurance it behaves correctly for users.



The Puppeteer website has extensive documentation that's worth reading: <https://pptr.dev/>.

Puppeteer is meant to be a general-purpose test automation tool and has a strong feature set for that purpose. Because it's easy to make web page screenshots with Puppeteer, it can also be used in a screenshot service.

Because Puppeteer is controlling a real web browser, your user interface tests will be very close to live browser testing, without having to hire a human to do the work. Because it uses a headless version of Chrome, no visible browser window will show on your screen, and tests can be run in the background instead. It can also drive other browsers by using the DevTools protocol.

First, let's set up a directory to work in.

Setting up a Puppeteer-based testing project directory

First, let's set up the directory that we'll install Puppeteer in, as well as the other packages that will be required for this project:

```
$ mkdir test-compose/notesui
$ cd test-compose/notesui
$ npm init
... answer the questions
$ npm install \
  puppeteer@^4.x mocha@^7.x chai@^4.x supertest@^4.x bcrypt@^4.x
  \ cross-env@7.x \
--save
```

This installs not just Puppeteer, but Mocha, Chai, and Supertest. We'll also be using the package.json file to record scripts.

During installation, you'll see that Puppeteer causes Chromium to be downloaded, like so:

```
Downloading Chromium r756035 - 118.4 Mb [=====] 35% 30.4s
```

The Puppeteer package will launch that Chromium instance as needed, managing it as a background process and communicating with it using the DevTools protocol.

The approach we'll follow is to test against the Notes stack we've deployed in the test Docker infrastructure. Therefore, we need to launch that infrastructure:

```
$ cd ..
$ docker stack deploy --compose-file docker-compose.yml notes
... as before
```

Depending on what you need to do, docker-compose build might also be required. In any case, this brings up the test infrastructure and lets you see the running system.

We can use a browser to visit `http://localhost:3000` and so on. Because this system won't contain any users, our test script will have to add a test user so that the test can log in and add notes.

Another item of significance is that tests will be running in an anonymous Chromium instance. Even if we use Chrome as our normal desktop browser, this Chromium instance will have no connection to our normal desktop setup. That's a good thing from a testability standpoint since it means your test results will not be affected by your personal web browser configuration. On the other hand, it means Twitter login testing is not possible, because that Chromium instance does not have a Twitter login session.

With those things in mind, let's write an initial test suite. We'll start with a simple initial test case to prove we can run Puppeteer inside Mocha. Then, we'll test the login and logout functionality, the ability to add notes, and a couple of negative test scenarios. We'll close this section with a discussion on improving testability in HTML applications. Let's get started.

Creating an initial Puppeteer test for the Notes application stack

Our first test goal is to set up the outline of a test suite. We will need to do the following, in order:

1. Add a test user to the user authentication service.
2. Launch the browser.
3. Visit the home page.
4. Verify the home page came up.
5. Close the browser.
6. Delete the test user.

This will establish that we have the ability to interact with the launched infrastructure, start the browser, and see the Notes application. We will continue with the policy and clean up after the test to ensure a clean environment for subsequent test runs and will add, then remove, a test user.

In the `notesui` directory, create a file named `uitest.mjs` containing the following code:

```
import Chai from 'chai';
const assert = Chai.assert;
import supertest from 'supertest';
const request = supertest(process.env.URL_USERS_TEST);
const authUser = 'them';
const authKey = 'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF';
```

```
import { default as bcrypt } from 'bcrypt';
const saltRounds = 10;
import puppeteer from 'puppeteer';

async function hashpass(password) {
  let salt = await bcrypt.genSalt(saltRounds);
  let hashed = await bcrypt.hash(password, salt);
  return hashed;
}
```

This imports and configures the required modules. This includes setting up `bcrypt` support in the same way that is used in the authentication server. We've also copied in the authentication key for the user authentication backend service. As we did for the REST test suite, we will use the `SuperTest` library to add, verify, and remove the test user using the REST API snippets copied from the REST tests.

Add the following test block:

```
describe('Initialize test user', function() {
  it('should successfully add test user', async function() {
    await request.post('/create-user').send({
      username: "testme", password: await hashpass("w0rd"),
      provider: "local",
      familyName: "Einarrsdottir", givenName: "Ashildr",
      middleName: "TEST", emails: [ "md@stolen.test.tardis" ],
      photos: []
    })
    .set('Content-Type', 'application/json')
    .set('Accept', 'application/json')
    .auth(authUser, authKey);
  });
});
```

This adds a user to the authentication service. Refer back and you'll see this is similar to the test case in the REST test suite. If you want a verification phase, there is another test case that calls the `/find/testme` endpoint to verify the result. Since we've already verified the authentication system, we do not need to reverify it here. We just need to ensure we have a known test user we can use for scenarios where the browser must be logged in.

Keep this at the very end of `uitest.mjs`:

```
describe('Destroy test user', function() {
  it('should successfully destroy test user', async function() {
    await request.delete('/destroy/testme')
    .set('Content-Type', 'application/json')
    .set('Accept', 'application/json')
```

```
        .auth(authUser, authKey);
    });
});
```

At the end of the test execution, we should run this to delete the test user. The policy is to clean up after we execute the test. Again, this was copied from the user authentication service test suite. Between those two, add the following:

```
describe('Notes', function() {
  this.timeout(100000);
  let browser;
  let page;

  before(async function() {
    browser = await puppeteer.launch({
      sloMo: 500, headless: false
    });
    page = await browser.newPage();
  });

  it('should visit home page', async function() {
    await page.goto(process.env.NOTES_HOME_URL);
    await page.waitForSelector('a.nav-item[href="/users/login"]');
  });

  // Other test scenarios go here.

  after(async function() {
    await page.close();
    await browser.close();
  });
});
```

Remember that within `describe`, the tests are the `it` blocks. The `before` block is executed before all the `it` blocks, and the `after` block is executed afterward.

In the `before` function, we set up Puppeteer by launching a Puppeteer instance and starting a new Page object. Because `puppeteer.launch` has the `headless` option set to `false`, we'll see a browser window on the screen. This will be useful so we can see what's happening. The `sloMo` option also helps us see what's happening by slowing down the browser interaction. In the `after` function, we call the `close` method on those objects in order to close out the browser. The `puppeteer.launch` method takes an `options` object, with a long list of attributes that are worth learning about.

The `browser` object represents the entire browser instance that the test is being run on. In contrast, the `page` object represents what is essentially the currently open tab in the browser. Most Puppeteer functions execute asynchronously. Therefore, we can use `async` functions and the `await` keywords.

The `timeout` setting is required because it sometimes takes a longish time for the browser instance to launch. We're being generous with the timeout to minimize the risk of spurious test failures.

For the `it` clause, we do a tiny amount of browser interaction. Being a wrapper around a browser tab, the `page` object has methods related to managing an open tab. For example, the `goto` method tells the browser tab to navigate to the given URL. In this case, the URL is the Notes home page, which is passed in as an environment variable.

The `waitForSelector` method is part of a group of methods that wait for certain conditions. These include `waitForFileChooser`, `waitForFunction`, `waitForNavigation`, `waitForRequest`, `waitForResponse`, and `waitForXPath`. These, and the `waitFor` method, all cause Puppeteer to asynchronously wait for a condition to happen in the browser. The purpose of these methods is to give the browser time to respond to some input, such as clicking on a button. In this case, it waits until the web page loading process has an element visible at the given CSS selector. That selector refers to the **Login** button, which will be in the header.

In other words, this test visits the Notes home page and then waits until the **Login** button appears. We could call that a simple smoke test that's quickly executed and determines that the basic functionality is there.

Executing the initial Puppeteer test

We have the beginning of a Puppeteer-driven test suite for the Notes application. We have already launched the test infrastructure using `docker-compose`. To run the test script, add the following to the `scripts` section of the package `.json` file:

```
"test": "cross-env URL_USERS_TEST=http://localhost:5858  
NOTES_HOME_URL=http://localhost:3000 mocha uitest.mjs"
```

The test infrastructure we deployed earlier exposes the user authentication service on port 5858 and the Notes application on port 3000. If you want to test against a different deployment, adjust these URLs appropriately. Before running this, the Docker test infrastructure must be launched, which should have already happened.

Let's try running this initial test suite:

```
$ npm run test

> notesui@1.0.0 test /Users/David/Chapter13/compose-test/notesui
> URL_USERS_TEST=http://localhost:5858
NOTES_HOME_URL=http://localhost:3000 mocha uitest.mjs

  Initialize test user
    ✓ should successfully add test user (125ms)
  Notes
    ✓ should visit home page (1328ms)
  Destroy test user
    ✓ should successfully destroy test user (53ms)

  3 passing (5s)
```

We have successfully created the structure that we can run these tests in. We have set up Puppeteer and the related packages and created one useful test. The primary win is to have a structure to build further tests on top of.

Our next step is to add more tests.

Testing login/logout functionality in Notes

In the previous section, we created the outline within which to test the Notes user interface. We didn't do much testing regarding the application, but we proved that we can test Notes using Puppeteer.

In this section, we'll add an actual test. Namely, we'll test the login and logout functionality. The steps for this are as follows:

1. Log in using the test user identity.
2. Verify that the browser was logged in.
3. Log out.
4. Verify that the browser is logged out.

In `uitest.js`, insert the following test code:

```
describe('should log in and log out correctly', function() {
  this.timeout(100000);

  it('should log in correctly', async function() {
    await page.click('a.nav-item[href="/users/login"]');
    await page.waitForSelector('form[action="/users/login"]');
    await page.type('[name=username]', "testme", {delay: 100});
    await page.type('[name=password]', "w0rd", {delay: 100});
    await page.keyboard.press('Enter');
    await page.waitForNavigation({
      'waitFor': 'domcontentloaded'
    });
  });

  it('should be logged in', async function() {
    assert.isNotNull(await page.$('a[href="/users/logout"]'));
  });

  it('should log out correctly', async function() {
    await page.click('a[href="/users/logout"]');
  });

  it('should be logged out', async function() {
    await page.waitForSelector('a.nav-item[href="/users/login"]');
  });
}

// Other test scenarios go here.
```

This is our test implementation for logging in and out. We have to specify the `timeout` value because it is a new `describe` block.

The `click` method takes a CSS selector, meaning this first click event is sent to the **Login** button. A CSS selector, as the name implies, is similar to or identical to the selectors we'd write in a CSS file. With a CSS selector, we can target specific elements on the page.

To determine the selector to use, look at the HTML for the templates and learn how to describe the element you wish to target. It may be necessary to add ID attributes into the HTML to improve testability.



The Puppeteer documentation refers to the CSS Selectors documentation on the Mozilla Developer Network website: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors.

Clicking on the **Login** button will, of course, cause the **Login** page to appear. To verify this, we wait until the page contains a form that posts to `/users/login`. That form is in `login.hbs`.

The `type` method acts as a user typing text. In this case, the selectors target the `Username` and `Password` fields of the login form. The `delay` option inserts a pause of 100 milliseconds after typing each character. It was noted in testing that sometimes, the text arrived with missing letters, indicating that Puppeteer can type faster than the browser can accept.

The `page.keyboard` object has various methods related to keyboard events. In this case, we're asking to generate the equivalent to pressing *Enter* on the keyboard. Since, at that point, the focus is in the Login form, that will cause the form to be submitted to the Notes application. Alternatively, there is a button on that form, and the test could instead click on the button.

The `waitForNavigation` method has a number of options for waiting on page refreshes to finish. The `selected` option causes a wait until the DOM content of the new page is loaded.

The `$` method searches the DOM for elements matching the selector, returning an array of matching elements. If no elements match, `null` is returned instead. Therefore, this is a way to test whether the application got logged in, by looking to see if the page has a **Logout** button.

To log out, we click on the **Logout** button. Then, to verify the application logged out, we wait for the page to refresh and show a **Login** button:

```
$ npm run test

> notesui@1.0.0 test /Users/David/Chapter13/compose-test/notesui
> URL_USERS_TEST=http://localhost:5858
NOTES_HOME_URL=http://localhost:3000 mocha uitest.mjs

Initialize test user
  ✓ should successfully add test user (188ms)
  ✓ should successfully verify test user exists
```

Notes

```
✓ should visit home page (1713ms)
log in and log out correctly
  ✓ should log in correctly (2154ms)
  ✓ should be logged in
  ✓ should log out correctly (287ms)
  ✓ should be logged out (55ms)

Destroy test user
  ✓ should successfully destroy test user (38ms)
  ✓ should successfully verify test user gone (39ms)

9 passing (7s)
```

With that, our new tests are passing. Notice that the time required to execute some of the tests is rather long. Even longer times were observed while debugging the test, which is why we set long timeouts.

That's good, but of course, there is more to test, such as the ability to add a Note.

Testing the ability to add Notes

We have a test case to verify login/logout functionality. The point of this application is adding notes, so we need to test this feature. As a side effect, we will learn how to verify page content with Puppeteer.

To test this feature, we will need to follow these steps:

1. Log in and verify we are logged in.
2. Click the **Add Note** button to get to the form.
3. Enter the information for a Note.
4. Verify that we are showing the Note and that the content is correct.
5. Click on the **Delete** button and confirm deleting the Note.
6. Verify that we end up on the home page.
7. Log out.

You might be wondering "*Isn't it duplicative to log in again?*" The previous tests focused on login/logout. Surely that could have ended with the browser in the logged-in state? With the browser still logged in, this test would not need to log in again. While that is true, it would leave the login/logout scenario incompletely tested. It would be cleaner for each scenario to be standalone in terms of whether or not the user is logged in. To avoid duplication, let's refactor the test slightly.

In the outermost describe block, add the following two functions:

```
describe('Notes', function() {
  this.timeout(100000);
  let browser;
  let page;

  async function doLogin() {
    await page.click('a.nav-item[href="/users/login"]');
    await page.waitForSelector('form[action="/users/login"]');
    await page.type('[name=username]', "testme", {delay: 150});
    await page.type('[name=password]', "w0rd", {delay: 150});
    await page.keyboard.press('Enter');
    await page.waitForNavigation({
      'waitFor': 'domcontentloaded'
    });
  }

  async function checkLogin() {
    const btnLogout = await page.$('a[href="/users/logout"]');
    assert.isNotNull(btnLogout);
  }
  ...
});
```

This is the same code as the code for the body of the test cases shown previously, but we've moved the code to their own functions. With this change, any test case that wishes to log into the test user can use these functions.

Then, we need to change the login/logout tests to this:

```
describe('log in and log out correctly', function() {
  this.timeout(100000);

  it('should log in correctly', doLogin);
  it('should be logged in', checkLogin);
  ...
});
```

All we've done is move the code that had been here into their own functions. This means we can reuse those functions in other tests, thus avoiding duplicative code.

Add the following code for the Note creation test suite to `uitest.mjs`:

```
describe('allow creating notes', function() {
  this.timeout(100000);

  it('should log in correctly', doLogin);
```

```
it('should be logged in', checkLogin);

it('should go to Add Note form', async function() {
    await page.click('a[href="/notes/add"]');
    await page.waitForSelector('form[action="/notes/save"]');
    await page.type('[name=notekey]', "testkey", {delay: 200});
    await page.type('[name=title]', "Test Note Subject", {delay:
        150});
    await page.type('[name=body]',
        "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
        eiusmod tempor incididunt ut labore et dolore magna aliqua.",
        { delay: 100 });
    await page.click('button[type="submit"]');
});

it('should view newly created Note', async function() {
    await page.waitForSelector('h3#notetitle');
    assert.include(
        await page.$eval('h3#notetitle', el => el.textContent),
        "Test Note Subject"
    );
    assert.include(
        await page.$eval('#notebody', el => el.textContent),
        "Lorem ipsum dolor"
    );
    assert.include(page.url(), '/notes/view');
});

it('should delete newly created Note', async function() {
    assert.isNotNull(await page.$('a#notedestroy'));
    await page.click('a#notedestroy');
    await page.waitForSelector(
        'form[action="/notes/destroy/confirm"]');
    await page.click('button[type="submit"]');
    await page.waitForSelector('#notetitles');
    assert.isNotNull(await page.$('a[href="/users/logout"]'));
    assert.isNotNull(await page.$('a[href="/notes/add"]'));
});

it('should log out', async function() {
    await page.click('a[href="/users/logout"]');
    await page.waitForSelector('a[href="/users/login"]');
});
});
```

These are our test cases for adding and deleting Notes. We start with the `doLogin` and `checkLogin` functions to ensure the browser is logged in.

After clicking on the **Add Note** button and waiting for the browser to show the form in which we enter the Note details, we need to enter text into the form fields. The `page.type` method acts as a user typing on a keyboard and types the given text into the field identified by the selector.

The interesting part comes when we verify the note being shown. After clicking the **Submit** button, the browser is, of course, taken to the page to view the newly created Note. To do this, we use `page.$eval` to retrieve text from certain elements on the screen.

The `page.$eval` method scans the page for matching elements, and for each, it calls the supplied callback function. The callback function is given the element, and in our case, we call the `textContent` method to retrieve the textual form of the element. Then, we're able to use the `assert.include` function to test that the element contains the required text.

The `page.url()` method, as its name suggests, returns the URL currently being viewed. We can test whether that URL contains `/notes/view` to be certain the browser is viewing a note.

To delete the note, we start by verifying that the **Delete** button is on the screen. Of course, this button is there if the user is logged in. Once the button is verified, we click on it and wait for the FORM that confirms that we want to delete the Note. Once it shows up, we can click on the button, after which we are supposed to land on the home page.

Notice that to find the **Delete** button, we need to refer to `a#notedestroy`. As it stands, the template in question does not have that ID anywhere. Because the HTML for the **Delete** button was not set up so that we could easily create a CSS selector, we must edit `views/noteedit.hbs` to change the **Delete** button to this:

```
<a class="btn btn-outline-dark" id="notedestroy"
    href="/notes/destroy?key={{notekey}}"
    role="button">Delete</a>
```

All we did was add the ID attribute. This is an example of improving testability, which we'll discuss later.

A technique we're using is to call `page.$` to query whether the given element is on the page. This method inspects the page, returning an array containing any matching elements. We are simply testing if the return value is non-null because `page.$` returns `null` if there are no matching elements. This makes for an easy way to test if an element is present.

We end this by logging out by clicking on the **Logout** button.

Having created these test cases, we can run the test suite again:

```
$ npm run test

> notesui@1.0.0 test /Users/David/Chapter13/compose-test/notesui
> URL_USERS_TEST=http://localhost:5858
NOTES_HOME_URL=http://localhost:3000 mocha uitest.mjs

Initialize test user
  ✓ should successfully add test user (228ms)
  ✓ should successfully verify test user exists (46ms)

Notes
  ✓ should visit home page (2214ms)
  log in and log out correctly
    ✓ should log in correctly (2567ms)
    ✓ should be logged in
    ✓ should log out correctly (298ms)
    ✓ should be logged out
  allow creating notes
    ✓ should log in correctly (2288ms)
    ✓ should be logged in
    ✓ should go to Add Note form (18221ms)
    ✓ should view newly created Note (39ms)
    ✓ should delete newly created Note (1225ms)

12 passing (1m)
```

We have more passing tests and have made good progress. Notice how one of the test cases took 18 seconds to finish. That's partly because we slowed text entry down to make sure it is correctly received in the browser, and there is a fair amount of text to enter. There was a reason we increased the timeout.

In earlier tests, we had success with negative tests, so let's see if we can find any bugs that way.

Implementing negative tests with Puppeteer

Remember that a negative test is used to purposely invoke scenarios that will fail. The idea is to ensure the application fails correctly, in the expected manner.

We have two scenarios for an easy negative test:

- Attempt to log in using a bad user ID and password
- Access a bad URL

Both of these are easy to implement, so let's see how it works.

Testing login with a bad user ID

A simple way to ensure we have a bad username and password is to generate random text strings for both. An easy way to do that is with the `uuid` package. This package is about generating Universal Unique IDs (that is, UUIDs), and one of the modes of using the package simply generates a unique random string. That's all we need for this test; it is a guarantee that the string will be unique.

To make this crystal clear, by using a unique random string, we ensure that we don't accidentally use a username that might be in the database. Therefore, we will be certain of supplying an unknown username when trying to log in.

In `uitest.mjs`, add the following to the imports:

```
import { v4 as uuidv4 } from 'uuid';
```

There are several methods supported by the `uuid` package, and the `v4` method is what generates random strings.

Then, add the following scenario:

```
describe('reject unknown user', function() {
  this.timeout(100000);

  it('should fail to log in unknown user correctly', async function()
  {
    assert.isNull(await page.$('a[href="/users/login"]'));
    await page.click('a.nav-item[href="/users/login"]');
    await page.waitForSelector('form[action="/users/login"]');
    await page.type('[name=username]', uuidv4(), {delay: 150});
    await page.type('[name=password]',
      await hashpass(uuidv4()), {delay: 150});
    await page.keyboard.press('Enter');
    await page.waitForSelector('form[action="/users/login"]');
    assert.isNull(await page.$('a[href="/users/login"]'));
    assert.isNull(await page.$('form[action="/users/login"]'));
  });
});
```

This starts with the login scenario. Instead of a fixed username and password, we instead use the results of calling `uuidv4()`, or the random UUID string.

This does the login action, and then we wait for the resulting page. In trying this manually, we learn that it simply returns us to the login screen and that there is no additional message. Therefore, the test looks for the login form and ensures there is a **Login** button. Between the two, we are certain the user is not logged in.

We did not find a code error with this test, but there is a user experience error: namely, the fact that, for a failed login attempt, we simply show the login form and do not provide a message (that is, *unknown username or password*), which leads to a bad user experience. The user is left feeling confused over what just happened. So, let's put that on our backlog to fix.

Testing a response to a bad URL

Our next negative test is to try a bad URL in Notes. We coded Notes to return a 404 status code, which means the page or resource was not found. The test is to ask the browser to visit the bad URL, then verify that the result uses the correct error message.

Add the following test case:

```
describe('reject unknown URL', function() {
  this.timeout(100000);

  it('should fail on unknown URL correctly', async function() {
    let u = new URL(process.env.NOTES_HOME_URL);
    u.pathname = '/bad-unknown-url';
    let response = await page.goto(u.toString());
    await page.waitForSelector('header.page-header');
    assert.equal(response.status(), 404);
    assert.include(
      await page.$eval('h1', el => el.textContent),
      "Not Found"
    );
    assert.include(
      await page.$eval('h2', el => el.textContent),
      "404"
    );
  });
});
```

This computes the bad URL by taking the URL for the home page (`NOTES_HOME_URL`) and setting the `pathname` portion of the URL to `/bad-unknown-url`. Since there is no route in Notes for this path, we're certain to get an error. If we wanted more certainty, it seems we could use the `uuidv4()` function to make the URL random.

Calling `page.goto()` simply gets the browser to go to the requested URL. For the subsequent page, we wait until a page with a `header` element shows up. Because this page doesn't have much on it, the `header` element is the best choice for determining when we have the subsequent page.

To check the 404 status code, we call `response.status()`, which is the status code that's received in the HTTP response. Then, we call `page.$eval` to get a couple of items from the page and make sure they contain the text that's expected.

In this case, we did not find any code problems, but we did find another user experience problem. The error page is downright ugly and user-unfriendly. We know the user experience team will scream about this, so add it to your backlog to do something to improve this page.

In this section, we wrapped up test development by creating a couple of negative tests. While this didn't result in finding code bugs, we found a pair of user experience problems. We know this will result in an unpleasant discussion with the user experience team, so we've proactively added a task to the backlog to fix those pages. But we also learned about being on the lookout for any kind of problem that crops up along the way. It's well-known that the lowest cost of fixing a problem is the issues found by the development or testing team. The cost of fixing problems goes up tremendously when it is the user community reporting the problems.

Before we wrap up this chapter, we need to talk a little more in-depth about testability.

Improving testability in the Notes UI

While the Notes application displays well in the browser, how do we write test software to distinguish one page from another? As we saw in this section, the UI test often performed an action that caused a page refresh and had to wait for the next page to appear. This means our test must be able to inspect the page and work out whether the browser is displaying the correct page. An incorrect page is itself a bug in the application. Once the test determines it is the correct page, it can then validate the data on the page.

The bottom line is a requirement stating that each HTML element must be easily addressable using a CSS selector.

While in most cases it is easy to code a CSS selector for every element, in a few cases, this is difficult. The **Software Quality Engineering (SQE)** manager has requested our assistance. At stake is the testing budget, which will be stretched further the more the SQE team can automate their tests.

All that's necessary is to add a few `id` or `class` attributes to HTML elements to improve testability. With a few identifiers and a commitment to maintaining those identifiers, the SQE team can write repeatable test scripts to validate the application.

We have already seen one example of this: the **Delete** button in `views/noteview.hbs`. It proved impossible to write a CSS selector for that button, so we added an `ID` attribute that let us write the test.

In general, *testability* is about adding things to an API or user interface for the benefit of software quality testers. For an HTML user interface, that means making sure test scripts can locate any element in the HTML DOM. And as we've seen, the `id` and `class` attributes go a long way to satisfying that need.

In this section, we learned about user interface testing as a form of functional testing. We used Puppeteer, a framework for driving a headless Chromium browser instance, as the vehicle for testing the Notes user interface. We learned how to automate user interface actions and how to verify that the web pages that showed up matched with their correct behavior. That included test scenarios covering login, logout, adding notes, and logging in with a bad user ID. While this didn't discover any outright failures, watching the user interaction told us of some usability problems with Notes.

With that, we are ready to close out this chapter.

Summary

We covered a lot of territory in this chapter and looked at three distinct areas of testing: unit testing, REST API testing, and UI functional tests. Ensuring that an application is well tested is an important step on the road to software success. A team that does not follow good testing practices is often bogged down with fixing regression after regression.

First, we talked about the potential simplicity of simply using the assert module for testing. While the test frameworks, such as Mocha, provide great features, we can go a long way with a simple script.

There is a place for test frameworks, such as Mocha, if only to regularize our test cases and to produce test results reports. We used Mocha and Chai for this, and these tools were quite successful. We even found a couple of bugs with a small test suite.

When starting down the unit testing road, one design consideration is mocking out dependencies. But it's not always a good use of our time to replace every dependency with a mock version. As a result, we ran our tests against a live database, but with test data.

To ease the administrative burden of running tests, we used Docker to automate setting up and tearing down the test infrastructure. Just as Docker was useful in automating the deployment of the Notes application, it's also useful in automating test infrastructure deployment.

Finally, we were able to test the Notes web user interface in a real web browser. We can't trust that unit testing will find every bug; some bugs will only show up in the web browser.

In this book, we've covered the full life cycle of Node.js development, from concept, through various stages of development, to deployment and testing. This will give you a strong foundation from which to start developing Node.js applications.

In the next chapter, we'll explore another critical area – security. We'll start by using HTTPS to encrypt and authenticate user access to Notes. We'll use several Node.js packages to limit the chances of security intrusions.

14

Security in Node.js Applications

We're coming to the end of this journey of learning Node.js. But there is one important topic left to discuss: **security**. The security of your applications is very important. Do you want to get into the news because your application is the greatest thing since Twitter, or do you want to be known for a massive cybersecurity incident launched through your website?

Cybersecurity officials around the world have for years clamored for greater security on the internet. Security holes in things as innocent as internet-connected security cameras have been weaponized by miscreants into vast botnets and are used to bludgeon websites or commit other mayhem. In other cases, rampant identity theft from security intrusions are a financial threat to us all. Almost every day, the news includes more revelations of cybersecurity problems.

We've mentioned this issue several times in this book. Starting in Chapter 10, *Deploying Node.js Applications on Linux*, we discussed the need to segment the deployment of Notes to present internal barriers against invasion, and specifically to keep the user database isolated in a protected container. The more layers of security you put around critical systems, the less likely it is that attackers can get in. While Notes is a toy application, we can use it to learn about implementing web application security.

Security shouldn't be an afterthought, just as testing should not be an afterthought. Both are incredibly important, if only to keep your company from getting in the news for the wrong reasons.

In this chapter, we will cover the following topics:

- Implementing HTTPS/SSL on AWS ECS for an Express application
- Using the Helmet library to implement headers for Content Security Policy, DNS Prefetch Control, Frame Options, Strict Transport Security, and mitigating XSS attacks
- Preventing cross-site request forgery attacks against forms
- SQL injection attacks
- Pre-deployment scanning for packages with known vulnerabilities
- Reviewing security facilities available on AWS



For general advice, the Express team has an excellent security resource page at <https://expressjs.com/en/advanced/best-practice-security.html>.

If you haven't yet done so, duplicate the `Chapter 13, Unit Testing and Functional Testing`, source tree, which you may have called `chap13`, to make a `Security` source tree, which you can call `chap14`.

By the end of this chapter, you will have experienced the details of provisioning SSL certificates, using them to implement an HTTPS reverse-proxy. Following that, you will read about several tools to improve the security of Node.js web applications. This should give you a foundation in web application security.

Let's start with implementing HTTPS support for the deployed Notes application.

Implementing HTTPS in Docker for deployed Node.js applications

The current best practice is that every website must be accessed with HTTPS. Gone are the days when it was okay to transmit unencrypted information over the internet. That old model is susceptible to problems such as man-in-the-middle attacks, and other threats.

Using SSL and HTTPS means that connections over the internet are authenticated and encrypted. The encryption is good enough to keep out all but the most advanced of snoops, and the authentication means we are assured the website is what it says it is. HTTPS uses the HTTP protocol but is encrypted using **SSL**, or **Secure Sockets Layers**. Implementing HTTPS requires getting an SSL certificate and implementing HTTPS support in the web server or web application.

Given a suitable SSL certificate, Node.js applications can easily implement HTTPS because a small amount of code gives us an HTTPS server. But there's another route that offers an additional benefit. NGINX is a well-regarded web server, and proxy server, that is extremely mature and feature-filled. We can use it to implement the HTTPS connection, and at the same time gain another layer of shielding between potential miscreants and the Notes application.

We have already deployed Notes using Docker swarm on an AWS EC2 cluster. Using NGINX is a simple matter of adding another container to the swarm, configured with the tools required to provision SSL certificates. For that purpose, we will use a Docker container that combines NGINX with a Let's Encrypt client program, and scripting to automate certificate renewal. Let's Encrypt is a non-profit operating an excellent service for free SSL certificates. Using their command-line tools, we can provision and otherwise manage SSL certificates as needed.

In this section, we will do the following:

1. Configure a domain name to point to our swarm
2. Incorporate a Docker container containing NGINX, Cron, and Certbot (one of the Let's Encrypt client tools)
3. Implement automated processes in that container for managing certificate renewal
4. Configure NGINX to listen on port 443 (HTTPS) alongside port 80 (HTTP)
5. Configure the Twitter application to support the website on HTTPS

This may seem like a lot of work, but every task is simple. Let's get started.

Assigning a domain name for an application deployed on AWS EC2

The Notes application is deployed using a Docker swarm built on AWS EC2 instances. One of those instances has a public IP address and a domain name assigned by AWS. It is best to assign a domain name to the EC2 instance because the name assigned by AWS is not only user-unfriendly, but will change the next time you redeploy the cluster. Giving the EC2 instance a domain name requires having a registered domain name, adding an A record listing its IP address, and updating the A record every time the EC2 IP address changes.

What does it mean to add an A record? The **domain name system (DNS)** is what lets us use a name such as `geekwisdom.net` for a website rather than the IP address, `216.239.38.21`. In the DNS protocol, there are several types of *records* that can be associated with domain name entries in the system. For this project, we need to only concern ourselves with one of those record types, the A record, for recording IP addresses for domain names. A web browser that's been told to visit any domain looks up the A record for that domain and uses that IP address to send HTTP(S) requests for website content.

The specific method to add an A record to the DNS entries of a domain varies considerably from one domain registrar to another. For example, one registrar (Pair Domains) has this screen:

Host Name	IP Address
notes.geekwisdom.net	35.163.196.76

In the dashboard for a specific domain, there might be a section for adding new DNS records. In this registrar, a dropdown lets you choose among the record types. Select the A record type, then for your domain name enter the IP address in the right-hand box, and in the left-hand box enter the subdomain name. In this case, we are creating a subdomain, `notes.geekwisdom.net`, so we can deploy a test site without disturbing the main site hosted on that domain. This also lets us avoid the expense of registering a new domain for this project.

As soon as you click the **ADD RECORD** button, the A record will be published. Since it usually takes some time for DNS records to propagate, you might not be able to visit the domain name right away. If this takes more than a couple of hours, you might have done something wrong.

Once the A record is successfully deployed, your users will be able to visit the Notes application at a nice domain like `notes.geekwisdom.net`.

Note that the IP address will change every time the EC2 instances are redeployed. If you redeploy the EC2 instances, you will need to update the A record for the new address.

In this section, we have learned about assigning a domain name to the EC2 instance. This will make it easier for our users to access Notes, while also letting us provision an HTTPS/SSL certificate.

Adding the domain name means updating the Twitter application configuration so that Twitter knows about the domain name.

Updating the Twitter application

Twitter needs to know which URLs are valid for our application. So far, we've told Twitter about test URLs on our laptop. We have Notes on a live domain, we need to tell Twitter about this.

We've already done this several times, so you already know what to do. Head to `developers.twitter.com`, logging in with your Twitter account, and go to the Apps dashboard. Edit the application related to your Notes instance, and add your domain name to the list of URLs.

We will be implementing both HTTP and HTTPS for the Notes application, and therefore Notes will have both `http://` and `https://` URLs. This means you must not only add the HTTP URLs to the Twitter configuration site, but also the HTTPS URLs.

In the `compose-stack/docker-compose.yml` file, the `TWITTER_CALLBACK_HOST` environment variable in the `svc-notes` configuration must also be updated with the domain.

We now have both a domain name associated with the EC2 cluster, and we've informed Twitter of the domain name. We should be able to redeploy Notes to the swarm and be able to use it with the domain name. That includes being able to log in using Twitter, creating and deleting notes, and so forth. At this point, you cannot put an HTTPS URL into `TWITTER_CALLBACK_HOST` because we've not implemented HTTPS support.

These steps prepared the way for implementing HTTPS on Notes using Let's Encrypt. But first, let's examine how Let's Encrypt works so we can better implement it for Notes.

Planning how to use Let's Encrypt

Like every HTTPS/SSL certificate provider, Let's Encrypt is required to be certain that you own the domain for which you're requesting the certificate. Successfully using Let's Encrypt requires successful validation before any SSL certificates are issued. Once a domain is registered with Let's Encrypt, the registration must be renewed at least every 90 days, because that's the expiry time for their SSL certificates. Domain registration, and certificate renewal, are therefore the two primary tasks we must accomplish.

In this section, we'll discuss how the registration and renewal features work. The goal is gaining an overview of how we'll manage an HTTPS service for any domain we plan to use.

Let's Encrypt supports an API and there are several client applications for this API. Certbot is the recommended user interface for Let's Encrypt requests. It is easily installed on a variety of operating systems. For example, it is available through the Debian/Ubuntu package management system.



For Let's Encrypt documentation, see <https://letsencrypt.org/docs/>.

For Certbot documentation, see <https://certbot.eff.org/docs/intro.html>.

Validated domain ownership is a core feature of HTTPS, making it a core requirement for any SSL certificate supplier to be certain it is handing out SSL certificates correctly. Let's Encrypt has several validation strategies, and in this project, we'll focus on one, the HTTP-01 challenge.

The HTTP-01 challenge involves the Let's Encrypt service making a request to a URL such as `http://<YOUR_DOMAIN>/ .well-known/acme-challenge/<TOKEN>`. The `<TOKEN>` is a coded string supplied by Let's Encrypt, which the Certbot tool will write as a file in a directory. Our task is then to somehow allow the Let's Encrypt servers to retrieve that file using this URL.

Once Certbot successfully registers the domain with Let's Encrypt, it receives a pair of PEM files comprising the SSL certificate. Certbot tracks various administrative details, and the SSL certificates, in a directory, by default `/etc/letsencrypt`. The SSL certificate in turn must be used to implement the HTTPS server for Notes.

Let's Encrypt SSL certificates expire in 90 days, and we must create an automated administrative task to renew the certificates. Certbot is also used for certificate renewal, by running `certbot renew`. This command looks at the domains registered on this server, and for any that require renewal it reruns the validation process. Therefore the directory required for the HTTP-01 challenge must remain enabled.

With SSL certificates in hand, we must configure some an HTTP server instance to use those certificates to implement HTTPS. It's very possible to configure the `svc-notes` service to handle HTTPS on its own. In the Node.js runtime is an HTTPS server object that could handle this requirement. It would be a small rewrite in `notes/app.mjs` to accommodate SSL certificates to implement HTTPS, as well as the HTTP-01 challenge.

But there is another possible approach. Web servers such as NGINX are very mature, robust, well tested, and, most importantly, support HTTPS. We can use NGINX to handle the HTTPS connection, and use what's called a *reverse proxy* to pass along the traffic to `svc-notes` as HTTP. That is, NGINX would be configured to accept inbound HTTPS traffic, converting it to HTTP traffic to send to `svc-notes`.

Beyond the security goal of implementing HTTPS, this has an additional advantage of using a well-regarded web server (NGINX) to act as a shield against certain kinds of attacks.

Having looked over the Let's Encrypt documentation, we have a handle on how to proceed. There is a Docker container available that handles everything we need to do with NGINX and Let's Encrypt. In the next section, we'll learn how to integrate that container with the Notes stack, and implement HTTPS.

Using NGINX and Let's Encrypt in Docker to implement HTTPS for Notes

We just discussed how to implement HTTPS for Notes using Let's Encrypt. The approach we'll take is to use a pre-baked Docker container, Cronginx (<https://hub.docker.com/r/robogeek/cronginx>), which includes NGINX, Certbot (a Let's Encrypt client), and a Cron server with a Cron job for managing SSL certificate renewal. This will simply require adding another container to the Notes stack, a little bit of configuration, and running a command to register our domain with Let's Encrypt.

Before starting this section, make sure you have set aside a domain name that we will use in this project.

In the Cronginx container, Cron is used for managing a background task to renew SSL certificates. Yes, Cron, the server Linux/Unix administrators have used for decades for managing background tasks.

The NGINX configuration will both handle the HTTP-01 challenge and use a reverse proxy for the HTTPS connection. A *proxy server* acts as an intermediary; it receives requests from clients and uses other services to satisfy those requests. A *reverse proxy* is a kind of proxy server that retrieves resources from one or more other servers, while making it look like the resource came from the proxy server. In this case, we will configure NGINX to access the Notes service at `http://svc-notes:3000`, while making the appearance that the Notes service is hosted by the NGINX proxy.

If you don't know how to configure NGINX, don't worry because we'll show exactly what to do, and it's relatively simple.

Adding the Cronginx container to support HTTPS on Notes

We've determined that adding HTTPS support requires the addition of another container to the Notes stack. This container will handle the HTTPS connection and incorporate tools for managing SSL certificates provisioned from Let's Encrypt.

In the `compose-stack` directory, edit `docker-compose.yml` like so:

```
services:  
  ...  
  svc-notes:  
    ...  
    # ports:
```

```
# - "80:3000"
...
environment:
  TWITTER_CALLBACK_HOST: "http://YOUR-DOMAIN"
...
cronginx:
  image: robogeek/cronginx
  container_name: cronginx
  deploy:
    replicas: 1
    placement:
      constraints:
        - "node.labels.type==public"
  networks:
    - frontnet
  ports:
    - 80:80
    - 443:443
  dns:
    - 8.8.8.8
    - 9.9.9.9
  restart: always
  volumes:
    - type: bind
      source: /home/ubuntu/etc-letsencrypt
      target: /etc/letsencrypt
    - type: bind
      source: /home/ubuntu/webroots
      target: /webroots
    - type: bind
      source: /home/ubuntu/nginx-conf-d
      target: /etc/nginx/conf.d
```

Because the `svc-notes` container will not be handling inbound traffic, we start by disabling its `ports` tag. This has the effect of ensuring it does not export any ports to the public. Instead, notice that in the `cronginx` container we export both port 80 (HTTP) and port 443 (HTTPS). That container will take over interfacing with the public internet.

Another change on `svc-notes` is to set the `TWITTER_CALLBACK_HOST` environment variable. Set this to the domain name you've chosen. Remember that correctly setting this variable is required for successful login using Twitter. Until we finish implementing HTTPS, this should have an HTTP URL.

The deploy tag for Cronginx is the same as for `svc-notes`. In theory, because `svc-notes` is no longer interacting with the public it could be redeployed to an EC2 instance on the private network. Because both are attached to `frontnet`, either will be able to access the other with a simple domain name reference, which we'll see in the configuration file.

This container uses the same DNS configuration, because Certbot needs to be able to reach the Let's Encrypt servers to do its work.

The final item of interest is the volume mounts. In the previous section, we discussed certain directories that must be mounted into this container. As with the database containers, the purpose is to persist the data in those directories while letting us destroy and recreate the Cronginx container as needed. Each directory is mounted from `/home/ubuntu` because that's the directory that is available on the EC2 instances. The three directories are as follows:

- `/etc/letsencrypt`: As discussed earlier, Certbot uses this directory to track administrative information about domains being managed on the server. It also stores the SSL certificates in this directory.
- `/webroots`: This directory will be used in satisfying the HTTP-01 request to `the http://<YOUR_DOMAIN>/.well-known/acme-challenge/<TOKEN>` URL.
- `/etc/nginx/conf.d`: This directory holds the NGINX configuration files for each domain we'll handle using this Cronginx instance.

For NGINX configuration, there is a default config file at `/etc/nginx/nginx.conf`. That file automatically includes any configuration file in `/etc/nginx/conf.d`, within an `http` context. What that means is each such file should have one or more `server` declarations. It won't be necessary to go deeper into learning about NGINX since the config files we will use are very straightforward.

We will be examining NGINX configuration files. If you need to learn more about these files, the primary documentation is at <https://nginx.org/en/docs/>.



Further documentation for the commercial NGINX Plus product is at <https://www.nginx.com/resources/admin-guide/>.

The NXING website has a *Getting Started* section with many useful recipes at <https://www.nginx.com/resources/wiki/start/>.

It will be a useful convention to follow to have one file in the `/etc/nginx/conf.d` directory for each domain you are hosting. That means, in this project, you will have one domain, and therefore you'll store one file in the directory named `YOUR-DOMAIN.conf`. For the example domain we configured earlier, that file would be `notes.geekwisdom.net.conf`.

Creating an NGINX configuration to support registering domains with Let's Encrypt

At this point, you have selected a domain you will use for Notes. To register a domain with Let's Encrypt, we need a web server configured to satisfy requests to the `http://<YOUR_DOMAIN>/.well-known/acme-challenge/<TOKEN>` URL, and where the corresponding directory is writable by Certbot. All the necessary elements are contained in the Cronginx container.

What we need to do is create an NGINX configuration file suitable for handling registration, then run the shell script supplied inside Cronginx. After registration is handled, there will be another NGINX configuration file that's suitable for HTTPS. We'll go over that in a later section.

Create a file for your domain named `initial-YOUR-DOMAIN.conf`, named this way because it's the initial configuration file for the domain. It will contain this:

```
# HTTP — redirect all traffic to HTTPS
server {
    listen 80;
    # listen [::]:80 default_server ipv6only=on;

    # Here put the domain name the server is to be known as.
    server_name YOUR-DOMAIN www.YOUR-DOMAIN;
    access_log /var/log/YOUR-DOMAIN.access.log main;
    error_log /var/log/YOUR-DOMAIN.error.log debug;

    # This is for handling the ACME challenge. Substitute the
    # domain name here.
    location /.well-known/ {
        root /webroots/YOUR-DOMAIN/;
    }

    # Use this to proxy for another service
    location / {
        proxy_pass http://svc-notes:3000/;
    }
}
```

As we said, the NGINX configuration files are relatively simple. This declares a server, in this case listening to port 80 (HTTP). It is easy to turn on IPv6 support if you wish.

The `server_name` field tells NGINX which domain name to handle. The `access_log` and `error_log` fields, as the name implies, specify where to send logging output.

The `location` blocks describe how to handle sections of the URL space for the domain. In the first, it says that HTTP-01 challenges on the `/well-known` URL are handled by reading files from `/webroots/YOUR-DOMAIN`. We've already seen that directory referenced in the `docker-compose.yml` file.

The second `location` block describes the reverse proxy configuration. In this case, we configure it to run an HTTP proxy to the `svc-notes` container at port 3000. That corresponds to the configuration in the `docker-compose.yml` file.

That's the configuration file, but we need to do a little work before we can deploy it to the swarm.

Adding the required directories on the EC2 host

We've identified three directories to use with Cronginx. Remember that each of the EC2 hosts is configured by a shell script we supply in the `user_data` field in the Terraform files. That script installs Docker and performs another setup. Therefore, we should use that script to create the three directories.

In `terraform-swarm`, edit `ec2-public.tf` and make this change:

```
resource "aws_instance" "public" {
  ...
  user_data = join("\n", [
    ...
    // Make directories required for croninx container
    "mkdir /home/ubuntu/etc-letsencrypt",
    "mkdir /home/ubuntu/webroots",
    "mkdir /home/ubuntu/nginx-conf-d"
  ]);
}
```

There is an existing shell script that performs the Docker setup. These three lines are appended to that script and create the directories.

With this in place, we can redeploy the EC2 cluster, and the directories will be there ready to be used.

Deploying the EC2 cluster and Docker swarm

Assuming that the EC2 cluster is currently not deployed, we can set it up as we did in Chapter 12, *Deploying a Docker Swarm to AWS EC2 with Terraform*. In `terraform-swarm`, run this command:

```
$ terraform apply
```

By now you will have done this several times and know what to do. Wait for it to finish deploying, record the IP addresses and other data, then initialize the swarm cluster and set up remote control access so you can run Docker commands on your laptop.

A very important task is to take the IP address and go to your DNS registrar and update the A record for the domain with the new IP address.

We need to copy the NGINX configuration file into `/home/ubuntu/nginx-conf-d`, so let's do so as follows:

```
$ ssh ubuntu@PUBLIC-IP-ADDRESS sudo chown ubuntu nginx-conf-d
$ scp initial-YOUR-DOMAIN.conf \
    ubuntu@PUBLIC-IP-ADDRESS:/home/ubuntu/nginx-conf-d/
    YOUR-DOMAIN.conf
```

The `chown` command is required because when Terraform created that directory it became owned by the `root` user. It needs to be owned by the `ubuntu` user for the `scp` command to work.

At this point make sure that, in `compose-swarm/docker-compose.yml`, the `TWITTER_CALLBACK_HOST` environment variable for `svc-notes` is set to the HTTP URL (`http://YOUR-DOMAIN`) rather than the HTTPS URL. Obviously you have not yet provisioned HTTPS and can only use the HTTP domain.

With those things set up, we can run this:

```
$ printf '....' | docker secret create TWITTER_CONSUMER_SECRET --
xgfp14f7grcx33e7hn3pjme9
$ printf '....' | docker secret create TWITTER_CONSUMER_KEY --
1xen2h4cjige0uonxnyyg8icq

$ docker stack deploy --with-registry-auth \
```

```
--compose-file docker-compose.yml notes
...
Creating network notes_frontnet
Creating network notes_authnet
Creating network notes_svcnet
Creating service notes_db-notes
Creating service notes_svc-notes
Creating service notes_redis
Creating service notes_cronginx
Creating service notes_db-userauth
Creating service notes_svc-userauth
```

This adds the required secrets to the swarm, and then deploys the Notes stack. After a few moments, the services should all show as having launched. Notice that Cronginx is one of the services.

Once it's fully launched, you should be able to use Notes as always, but using the domain you configured. You should even be able to log in using Twitter.

Registering a domain with Let's Encrypt

We have just deployed the Notes stack on the AWS EC2 infrastructure. A part of this new deployment is the Cronginx container with which we'll handle HTTPS configuration.

We have Notes deployed on the swarm, with the croninx container acting as an HTTP proxy. Inside that container came pre-installed the Certbot tool and a script (`register.sh`) to assist with registering domains. We must run `register.sh` inside the croninx container, and once the domain is registered we will need to upload a new NGINX configuration file.

Starting a shell inside the croninx container can be this easy:

```
$ docker ps
.... look for container name for croninx
$ docker exec -it notes_croninx.1.CODE-STRING bash
root@d4a81204cca4:/scripts# ls
register.sh renew.sh
```

You see there is a file named `register.sh` containing the following:

```
#!/bin/sh
mkdir -p /webroots/$1/.well-known/acme-challenge
certbot certonly --webroot -w /webroots/$1 -d $1
```

This script is designed to both create the required directory in `/webroots`, and to use Certbot to register the domain and provision the SSL certificates. Refer to the configuration file and you'll see how the `/webroots` directory is used.

The `certbot certonly` command only retrieves SSL certificates and does not install them anywhere. What that means is it does not directly integrate with any server, but simply stashes the certificates in a directory. That directory is within the `/etc/letsencrypt` hierarchy.

The `--webroot` option means that we're running in cooperation with an existing web server. It must be configured to serve the `/.well-known/acme-challenge` files from the directory specified with the `-w` option, which is the `/webroots/YOUR-DOMAIN` directory we just discussed. The `-d` option is the domain name to be registered.

In short, `register.sh` fits with the configuration file we created.

The script is executed like so:

```
root@d4a81204cca4:/scripts# sh -x register.sh notes.geekwisdom.net
+ mkdir -p /webroots/notes.geekwisdom.net/.well-known/acme-challenge
+ certbot certonly --webroot -w /webroots/notes.geekwisdom.net -d
notes.geekwisdom.net
Saving debug log to /var/log/letsencrypt/letsencrypt.log
Plugins selected: Authenticator webroot, Installer None
Enter email address (used for urgent renewal and security notices)
(Enter 'c' to
cancel): ...
```

We run the shell script using `sh -x register.sh` and supply our chosen domain name as the first argument. Notice that it creates the `/webroots` directory, which is required for the Let's Encrypt validation. It then runs `certbot certonly`, and the tool starts asking questions required for registering with the service.

The registration process ends with this message:

```
Obtaining a new certificate
Performing the following challenges:
http-01 challenge for notes.geekwisdom.net
Using the webroot path /webroots/notes.geekwisdom.net for all
unmatched domains.
Waiting for verification...
Cleaning up challenges
```

IMPORTANT NOTES:

- Congratulations! Your certificate and chain have been saved at:
`/etc/letsencrypt/live/notes.geekwisdom.net/fullchain.pem`
Your key file has been saved at:
`/etc/letsencrypt/live/notes.geekwisdom.net/privkey.pem`
Your cert will expire on 2020-09-23. To obtain a new or tweaked
version of this certificate in the future, simply run certbot
again. To non-interactively renew *all* of your certificates, run
`"certbot renew"`
- Your account credentials have been saved in your Certbot
configuration directory at `/etc/letsencrypt`. You should make a
secure backup of this folder now. This configuration directory will
also contain certificates and private keys obtained by Certbot so
making regular backups of this folder is ideal.

The key data is the pathnames for the two PEM files that make up the SSL certificate. It also tells you to run `certbot renew` every so often to renew the certificates. We already took care of that by installing the Cron job.

As they say, it is important to persist this directory elsewhere. We took the first step by storing it outside the container, letting us destroy and recreate the container at will. But what about when it's time to destroy and recreate the EC2 instances? Place a task on your backlog to set up a backup procedure, and then during EC2 cluster initialization install this directory from the backup.

Now that our domain is registered with Let's Encrypt, let's change the NGINX configuration to support HTTPS.

Implementing an NGINX HTTPS configuration using Let's Encrypt certificates

Alright, we're getting so close we can taste the encryption. We have deployed NGINX plus Let's Encrypt tools into the Notes application stack. We've verified that the HTTP-only NGINX configuration works correctly. And we've used Certbot to provision SSL certificates for HTTPS from Let's Encrypt. That makes it time to rewrite the NGINX configuration to support HTTPS and to deploy that config to the Notes stack.

In `compose-stack/cronginx` create a new file, `YOUR-DOMAIN.conf`, for example `notes.geekwisdom.net.conf`. The previous file had a prefix, `initial`, because it served us for the initial phase of implementing HTTPS. Now that the domain is registered with Let's Encrypt, we need a different configuration file:

```
# HTTP — redirect all traffic to HTTPS
server {
    listen 80;
    # listen [::]:80 default_server ipv6only=on;

    # Here put the domain name the server is to be known as.
    server_name YOUR-DOMAIN www.YOUR-DOMAIN;
    access_log /var/log/YOUR-DOMAIN.access.log main;
    error_log /var/log/YOUR-DOMAIN.error.log debug;

    # This is for handling the ACME challenge. Substitute the
    # domain name here.
    location /.well-known/ {
        root /webroots/YOUR-DOMAIN/;
    }

    # Use this to force a redirect to the SSL/HTTPS site
    return 301 https://$host$request_uri;
}
```

This reconfigures the HTTP server to do permanent redirects to the HTTPS site. When an HTTP request results in a **301** status code, that is a permanent redirect. Any redirect tells web browsers to visit a URL provided in the redirect. There are two kinds of redirects, temporary and permanent, and the **301** code makes this a permanent redirect. For permanent redirects, the browser is supposed to remember the redirect and apply it in the future. In this case, the redirect URL is computed to be the request URL, rewritten to use the HTTPS protocol.

Therefore our users will silently be sent to the HTTPS version of Notes, with no further effort on our part.

To implement the HTTPS server, add this to the config file:

```
# HTTPS service
server { # simple reverse-proxy
    # Enable HTTP/2
    listen 443 ssl http2;
    # listen [::]:443 ssl http2;

    # Substitute here the domain name for the site
    server_name YOUR-DOMAIN www.YOUR-DOMAIN;
```

```
access_log /var/log/YOUR-DOMAIN.access.log main;
error_log /var/log/YOUR-DOMAIN.error.log debug;

# Use the Let's Encrypt certificates
# Substitute in the domain name
ssl_certificate /etc/letsencrypt/live/YOUR-DOMAIN/fullchain.pem;
ssl_certificate_key /etc/letsencrypt/live/YOUR-DOMAIN/privkey.pem;

# Replication of the ACME challenge handler. Substitute
# the domain name.
location /.well-known/ {
    root /webroots/YOUR-DOMAIN/;
}

# See:
# https://stackoverflow.com/questions/29043879/socket-io-with-nginx
location ^~ /socket.io/ {
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_set_header X-NginX-Proxy false;

    proxy_pass http://svc-notes:3000;
    proxy_redirect off;

    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
}

# Use this for proxying to a backend service
# The HTTPS session is terminated at this Proxy.
# The back end service will see a simple HTTP service.
location / {
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-NginX-Proxy true;
    proxy_pass http://svc-notes:3000/;
    proxy_ssl_session_reuse off;
    proxy_set_header Host $http_host;
    proxy_cache_bypass $http_upgrade;
    proxy_redirect off;
}
}
```

This is an HTTPS server implementation in NGINX. There are many similarities to the HTTP server declaration, but with a couple of HTTPS – specific items. It listens on port 443, the standard port for HTTPS, and tells NGINX to use SSL. It has the same configuration for the server name and logging.

The next segment tells NGINX the location of the SSL certificates. Simply replace this with the pathname that Certbot gave you.

The next segment handles the /.well-known URL for future validation requests with Let's Encrypt. Both the HTTP and HTTPS server definitions have been configured to handle this URL from the same directory. We don't know whether Let's Encrypt will request validation through the HTTP or HTTPS URL, so we might as well support this on both servers.

The next segment is a proxy server to handle the /socket.io URL. This requires specific settings because Socket.IO must negotiate an upgrade from HTTP/1.1 to WebSocket. Otherwise, an error is printed in the JavaScript console, and the Socket.IO features will not work. For more information, see the URL shown in the code.

The last segment is a reverse proxy set up to proxy HTTPS traffic to an HTTP backend server. In this case, the backend server is the Notes application running on port 3000.

Having created a new configuration file, we can upload it to the notes-public EC2 instance like so:

```
$ scp YOUR-DOMAIN.conf \
  ubuntu@52.32.117.130:/home/ubuntu/nginx-conf-d/YOUR-DOMAIN.conf
```

The next question is how do we restart the NGINX server so it reads the new configuration file? One way is to send a SIGHUP signal to the NGINX process, causing it to reload the configuration:

```
$ docker exec -it notes_cronnginx.1.8c2in59gz7b4g2asxfxgd1y3q bash
root@31a813dad28c:/scripts# kill -HUP `cat /var/run/nginx.pid`
```

The nginx.pid file contains the process ID of the NGINX process. Many background services on Unix/Linux systems store the process ID in such a file. This command sends the SIGHUP signal to that process, and NGINX is written to reread its configuration upon receiving that signal. SIGHUP is one of the standard Unix/Linux *signals*, and is commonly used to cause background processes to reload their configuration like this. For more information, see the `signal(2)` man page.

However, using Docker commands we can do this:

```
$ docker service update --force notes_croninx
notes_croninx
overall progress: 1 out of 1 tasks
1/1: running
verify: Service converged
```

That will kill the existing container and start a new one.

Instead of that rosy success message, you might get this instead:

```
service update paused: update paused due to failure or early
termination of task flueg3xg8aclciq05r1o2bk1w
```

This says that Docker swarm saw that the container exited, and it was therefore unable to restart the service.

It is easy to make mistakes in NGINX configuration files. First take a careful look at the configuration to see what might be amiss. The next stage of diagnosis is to look at the NGINX logs. We can do that with the `docker logs` command, but we need to know the container name. Because the container has exited, we must run this:

```
$ docker ps -a
```

The `-a` option causes `docker ps` to return information about every container, even the ones that are not currently running. With the container name in hand, we can run this:

```
$ docker logs notes_croninx.1.bytadzur7fyj0c3xtwokpcrv0
2020/06/25 18:36:18 [emerg] 8#8: unknown directive "Use" in
/etc/nginx/conf.d/YOUR-DOMAIN.conf:26
```

And indeed, the issue is a syntax error, and it even helpfully tells you the line number.

Once you have successfully restarted the `croninx` service, visit the Notes service you've deployed and verify that it is in HTTPS mode.

In this section, we successfully deployed HTTPS support for the Notes application stack on our AWS EC2 based Docker swarm. We used the files Docker container created in the previous section and deployed the updated Notes Stack to the swarm. We then ran Certbot to register our domain with Let's Encrypt. And we rewrote the NGINX configuration to support HTTPS.

Our next task is to verify the HTTPS configuration is working correctly.

Testing HTTPS support for the Notes application

We have done ad hoc testing, and more formal testing, of Notes all through this book. Therefore you know what to do to ensure Notes is working in this new environment. But there are a couple of HTTPS-specific things to check.

In your browser, head to the domain name where you've hosted the application. If all went well, you will be greeted by the application, and it will have redirected to the HTTPS port automatically.

So that we humans know that a website is on HTTPS, most browsers show a *lock* icon in the location bar.

You should be able to click on that lock icon, and the browser will show a dialog giving information about the certificate. The certificate will verify that this is indeed the correct domain, and will also show the certificate was issued by Let's Encrypt via the **Let's Encrypt Authority X3**.

You should be able to browse around the entire application and still see the lock icon.

You should be on the lookout for *mixed content* warnings. These will appear in the JavaScript console and occur when some content on an HTTPS-loaded page is loaded using an HTTP URL. The mixed content scenario is less secure, and therefore browsers issue warnings to the user. Messages might appear in the JavaScript console inside the browser. If you have followed the instructions in this book correctly you will not see this message.

Finally, head to the Qualys SSL Labs test page for SSL implementations. This service will examine your website, especially the SSL certificates, and give you a score. To examine your score, see <https://www.ssllabs.com/ssltest/>.

Having completed this task, you may want to bring down the AWS EC2 cluster. Before doing so, it's good form to de-register the domain from Let's Encrypt. That's also a simple matter of running Certbot with the right command:

```
$ docker ps
...
$ docker exec -it notes_croninginx.1.1gz1bi8cvr2c0gapuvibegkrn bash
root@f896d97f30d5:/scripts#
root@f896d97f30d5:/scripts# certbot delete --domain YOUR-DOMAIN
...
```

As before, we run `docker ps` to find out the exact container name. With that name, we start a command shell inside the container. The actual act is simple, we just run `certbot delete` and specify the domain name.

Certbot doesn't just go ahead and delete the registration. Instead, it asks you to verify that's what you want to do, then it deletes the registration.

In this section, we have finished implementing HTTPS support for Notes by learning how to test that it is implemented correctly.

We've accomplished a redesign of the Notes application stack using a custom NGINX-based container to implement HTTPS support. This approach can be used for any service deployment, where an NGINX instance is used as the frontend to any kind of backend service.

But we have other security fish to fry. Using HTTPS solves only part of the security problem. In the next section, we'll look at Helmet, a tool for Express applications to set many security options in the HTTP headers.

Using Helmet for across-the-board security in Express applications

While it was useful to implement HTTPS, that's not the end of implementing security measures. It's hardly the beginning of security, for that matter. The browser makers working with the standards organizations have defined several mechanisms for telling the browser what security measures to take. In this section, we will go over some of those mechanisms, and how to implement them using Helmet.

Helmet (<https://www.npmjs.com/package/helmet>) is, as the development team says, not a security silver bullet (do Helmet's authors think we're trying to protect against vampires?). Instead, it is a toolkit for setting various security headers and taking other protective measures in Node.js applications. It integrates with several packages that can be either used independently or through Helmet.

Using Helmet is largely a matter of importing the library into `node_modules`, making a few configuration settings, and integrating it with Express.

In the notes directory, install the package like so:

```
$ npm install helmet --save
```

Then add this to notes/app.mjs:

```
import helmet from 'helmet';
...
const app = express();
export default app;

app.use(helmet());
```

That's enough for most applications. Using Helmet out of the box provides a reasonable set of default security options. We could be done with this section right now, except that it's useful to examine closely what Helmet does, and its options.

Helmet is actually a cluster of 12 modules for applying several security techniques. Each can be individually enabled or disabled, and many have configuration settings to make. One option is instead of using that last line, to initialize and configure the sub-modules individually. That's what we'll do in the following sections.

Using Helmet to set the Content-Security-Policy header

The **Content-Security-Policy (CSP)** header can help to protect against injected malicious JavaScript and other file types.

We would be remiss to not point out a glaring problem with services such as the Notes application. Our users could enter any code they like, and an improperly behaving application will simply display that code. Such applications can be a vector for JavaScript injection attacks among other things.

To try this out, edit a note and enter something like this:

```
<script src="https://pirates.den/malicious.js"></script>
```

Click the **Save** button, and you'll see this code displayed as text. A dangerous version of Notes would instead insert the `<script>` tag in the notes view page so that the malicious JavaScript would be loaded and cause a problem for our visitors. Instead, the `<script>` tag is encoded as safe HTML so it simply shows up as text on the screen. We didn't do anything special for that behavior, Handlebars did that for us.

Actually, it's a little more interesting. If we look at the Handlebars documentation, <http://handlebarsjs.com/expressions.html>, we learn about this distinction:

```
 {{encodedAsHtml}}  
 {{notEncodedAsHtml}}
```

In Handlebars, a value appearing in a template using two curly braces (`{{encoded}}`) is encoded using HTML coding. For the previous example, the angle bracket is encoded as `<` and so on for display, rendering that JavaScript code as neutral text rather than as HTML elements. If instead, you use three curly braces (`{{{{notEncoded}}}}`), the value is not encoded and is instead presented as is. The malicious JavaScript would be executed in your visitor's browser, causing problems for your users.

We can see this problem by changing `views/noteview.hbs` to use raw HTML output:

```
 {{#if note}}<div id="notebody">{{{ note.body }}}</div>{{/if}}
```

We do not recommend doing this except as an experiment to see what happens. The effect is, as we just said, to allow our users to enter HTML code and have it displayed as is. If Notes were to behave this way, any note could potentially hold malicious JavaScript snippets or other malware.

Let's return to Helmet's support for the Content-Security-Policy header. With this header, we instruct the web browser the scope from which it can download certain types of content. Specifically, it lets us declare which domains the browser can download JavaScript, CSS, or Font files from, and which domains the browser is allowed to connect to for services.

This header, therefore, solves the named issue, namely our users entering malicious JavaScript code. But it also handles a similar risk of a malicious actor breaking in and modifying the templates to include malicious JavaScript code. In both cases, telling the browser a specific list of allowed domains means references to JavaScript from malicious sites will be blocked. That malicious JavaScript that's loaded from `pirates.den` won't run.

To see the documentation for this Helmet module, see <https://helmetjs.github.io/docs/csp/>.

There is a long list of options. For instance, you can cause the browser to report any violations back to your server, in which case you'll need to implement a route handler for `/report-violation`. This snippet is sufficient for Notes:

```
app.use(helmet.contentSecurityPolicy({
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'", "'unsafe-inline'"],
    styleSrc: ["'self'", 'fonts.googleapis.com'],
    fontSrc: ["'self'", 'fonts.gstatic.com'],
    connectSrc: [ "'self'", 'wss://notes.geekwisdom.net' ]
  }
}));
```

For better or for worse, the Notes application implements one security best practice—all CSS and JavaScript files are loaded from the same server as the application. Therefore, for the most part, we can use the `'self'` policy. There are several exceptions:

- `scriptSrc`: Defines where we are allowed to load JavaScript. We do use inline JavaScript in `noteview.hbs` and `index.hbs`, which must be allowed.
- `styleSrc, fontSrc`: We're loading CSS files from both the local server and from Google Fonts.
- `connectSrc`: The WebSockets channel used by Socket.IO is declared here.

To develop this, we can open the JavaScript console or Chrome DevTools while browsing the website. Errors will show up listing any domains of failed download attempts. Simply add such domains to the configuration object.

Making the ContentSecurityPolicy configurable

Obviously, the `ContentSecurityPolicy` settings shown here should be configurable. If nothing else the setting for `connectSrc` must be, because it can cause a problem that prevents Socket.IO from working. As shown here, the `connectSrc` setting includes the URL `wss://notes.geekwisdom.net`. The `wss` protocol here refers to WebSockets and is designed to allow Socket.IO to work while Notes is hosted on `notes.geekwisdom.net`. But what about when we want to host it on a different domain?

To experiment with this problem, change the hard coded string to a different domain name then redeploy it to your server. In the JavaScript console in your browser you will get an error like this:

```
Refused to connect to
wss://notes.geekwisdom.net/socket.io/?EIO=3&transport=websocket&sid=x-
WiqH-g6uKIqoNqAAPA because it does not appear in the connect-src
directive of the Content Security Policy.
```

What's happened is that the statically defined constant was no longer compatible with the domain where Notes was deployed. You had reconfigured this setting to limit connections to a different domain, such as `notes.newdomain.xyz`, but the service was still hosted on the existing domain, such as `notes.geekwisdom.net`. The browser no longer believed it was safe to connect to `notes.geekwisdom.net` because your configuration said to trust only `notes.newdomain.xyz`.

The best solution is to make this a configurable setting by declaring another environment variable that can be set to customize behavior.

In `app.mjs`, change the `contentSecurityPolicy` section to this:

```
const csp_connect_src = [ "'self'" ];
if (typeof process.env.CSP_CONNECT_SRC_URL === 'string'
  && process.env.CSP_CONNECT_SRC_URL !== '') {
  csp_connect_src.push(process.env.CSP_CONNECT_SRC_URL);
}
app.use(helmet.contentSecurityPolicy({
  directives: {
    defaultSrc: ["'self'",],
    scriptSrc: ["'self'", "'unsafe-inline'", ],
    styleSrc: ["'self'", 'fonts.googleapis.com'],
    fontSrc: ["'self'", 'fonts.gstatic.com'],
    connectSrc: csp_connect_src
  }
}));
```

This lets us define an environment variable, `CSP_CONNECT_SRC_URL`, which will supply a URL to be added into the array passed to the `connectSrc` parameter. Otherwise, the `connectSrc` setting will be limited to `"'self'"`.

Then in `compose-swarm/docker-compose.yml`, we can declare that variable like so:

```
services:
  ...

```

```
svc-notes:  
...  
environment:  
...  
CSP_CONNECT_SRC_URL: "wss://notes.geekwisdom.net"  
...
```

We can now set that in the configuration, changing it as needed.

After rerunning the `docker stack deploy` command, the error message will go away and Socket.IO features will start to work.

In this section, we learned about the potential for a site to send malicious scripts to browsers. Sites that accept user-supplied content, such as Notes, can be a vector for malware. By using this header, we are able to notify the web browser which domains to trust when visiting this website, which will then block any malicious content added by malicious third parties.

Next, let's learn about preventing excess DNS queries.

Using Helmet to set the X-DNS-Prefetch-Control header

DNS Prefetch is a nicety implemented by some browsers where the browser will preemptively make DNS requests for domains referred to by a given page. If a page has links to other websites, it will make DNS requests for those domains so that the local DNS cache is pre-filled. This is nice for users because it improves browser performance, but it is also a privacy intrusion and can make it look like the person visited websites they did not visit. For documentation, see <https://helmetjs.github.io/docs/dns-prefetch-control>.

Set the DNS prefetch control with the following:

```
app.use(helmet.dnsPrefetchControl({ allow: false })); // or true
```

In this case, we learned about preventing the browser from making premature DNS queries. The risk is that excess DNS queries give a false impression of which websites someone has visited.

Let's next look at how to control which browser features can be enabled.

Using Helmet to control enabled browser features using the Feature-Policy header

Web browsers nowadays have a long list of features that can be enabled, such as vibrating a phone, or turning on the camera or microphone, or reading the accelerometer. These features are interesting and very useful in some cases, but can be used maliciously. The Feature-Policy header lets us notify the web browser about which features to allow to be enabled, or to deny enabling.

For Notes we don't need any of those features, though some look intriguing as future possibilities. For instance, we could pivot to taking on Instagram if we allowed people to upload photos, maybe? In any case, this configuration is very strict:

```
app.use(helmet.featurePolicy({
  features: {
    accelerometer: ["'none'"],
    ambientLightSensor: ["'none'"],
    autoplay: ["'none'"],
    camera: ["'none'"],
    encryptedMedia: ["'self'"],
    fullscreen: ["'self'"],
    geolocation: ["'none'"],
    gyroscope: ["'none'"],
    vibrate: ["'none'"],
    payment: ["'none'"],
    syncXhr: ["'none'"]
  }
}));
```

To enable a feature, either set it to '`'self'`' to allow the website to turn on the feature, or a domain name of a third-party website to allow to enable that feature. For example, enabling the payment feature might require adding '`'paypal.com'`' or some other payment processor.

In this section, we have learned about allowing the enabling or disabling of browser features.

In the next section, let's learn about preventing clickjacking.

Using Helmet to set the X-Frame-Options header

Clickjacking has nothing to do with carjacking but instead is an ingenious technique for getting folks to click on something malicious. The attack uses an invisible `<iframe>`, containing malicious code, positioned on top of a thing that looks enticing to click on. The user would then be enticed into clicking on the malicious thing.

The `frameguard` module for Helmet will set a header instructing the browser on how to treat an `<iframe>`. For documentation, see <https://helmetjs.github.io/docs/frameguard/>.

```
app.use(helmet.frameguard({ action: 'deny' }));
```

This setting controls which domains are allowed to put this page into an `<iframe>`. Using `deny`, as shown here, prevents all sites from embedding this content using an `<iframe>`. Using `sameorigin` allows the site to embed its own content. We can also list a single domain name to be allowed to embed this content.

In this section, you have learned about preventing our content from being embedded into another website using `<iframe>`.

Now let's learn about hiding the fact that Notes is powered by Express.

Using Helmet to remove the X-Powered-By header

The `X-Powered-By` header can give malicious actors a clue about the software stack in use, informing them of attack algorithms that are likely to succeed. The `Hide Powered-By` submodule for Helmet simply removes that header.

Express can disable this feature on its own:

```
app.disable('x-powered-by')
```

Or you can use Helmet to do so:

```
app.use(helmet.hidePoweredBy())
```

Another option is to masquerade as some other stack like so:

```
app.use(helmet.hidePoweredBy({ setTo: 'Drupal 5.7.0' }))
```

There's nothing like throwing the miscreants off the scent.

We've learned how to let your Express application go incognito to avoid giving miscreants clues about how to break in. Let's next learn about declaring a preference for HTTPS.

Improving HTTPS with Strict Transport Security

Having implemented HTTPS support, we aren't completely done. As we said earlier, it is best for our users to use the HTTPS version of Notes. In our AWS EC2 deployment, we forced the user to use HTTPS with a redirect. But in some cases we cannot do that, and instead must try to encourage the users to visit the HTTPS site over the HTTP site.

The Strict Transport Security header notifies the browser that it should use the HTTPS version of the site. Since that's simply a notification, it's also necessary to implement a redirect from the HTTP to HTTPS version of Notes.

We set Strict-Transport-Security like so:

```
const sixtyDaysInSeconds = 5184000 // 60 * 24 * 60 * 60
app.use(helmet.hsts({
  maxAge: sixtyDaysInSeconds
});
```

This tells the browser to stick with the HTTPS version of the site for the next 60 days, and never visit the HTTP version.

And, as long as we're on this issue, let's learn about `express-force-ssl`, which is another way to implement a redirect so the users use HTTPS. After adding a dependency to that package in `package.json`, add this in `app.mjs`:

```
import forceSSL from 'express-force-ssl';
...
app.use(forceSSL);
app.use(bodyParser.json());
```

With this package installed, the users don't have to be encouraged to use HTTPS because we're silently forcing them to do so.

With our deployment on AWS EC2, using this module will cause problems. Because HTTPS is handled in the load balancer, the Notes app does not know the visitor is using HTTPS. Instead, Notes sees an HTTP connection, and if `forceSSL` were in use it would then force a redirect to the HTTPS site. But because Notes does not see the HTTPS session at all, it only sees HTTP requests to which `forceSSL` will always respond with a redirect.

These settings are not useful in all circumstances. Your context may require these settings, but for a context like our deployment on AWS EC2 it is simply not needed. For the sites where this is useful, we have learned about notifying the web browser to use the HTTPS version of our website, and how to force a redirect to the HTTPS site.

Let's next learn about **cross-site-scripting (XSS)** attacks.

Mitigating XSS attacks with Helmet

XSS attacks attempt to inject JavaScript code into website output. With malicious code injected into another website, the attacker can access information they otherwise could not retrieve, or cause other sorts of mischief. The X-XSS-Protection header prevents certain XSS attacks, but not all of them, because there are so many types of XSS attacks:

```
app.use(helmet.xssFilter());
```

This causes an X-XSS-Protection header to be sent specifying `1; mode=block`. This mode tells the browser to look for JavaScript in the request URL that also matches JavaScript on the page, and it then blocks that code. This is only one type of XSS attack, and therefore this is of limited usefulness. But it is still useful to have this enabled.

In this section, we've learned about using Helmet to enable a wide variety of security protections in web browsers. With these settings, our application can work with the browser to avoid a wide variety of attacks, and therefore make our site significantly safer.

But with this, we have exhausted what Helmet provides. In the next section, we'll learn about another package that prevents cross-site request forgery attacks.

Addressing Cross-Site Request Forgery (CSRF) attacks

CSRF attacks are similar to XSS attacks in that both occur across multiple sites. In a CSRF attack, malicious software forges a bogus request on another site. To prevent such an attack, CSRF tokens are generated for each page view. The tokens are to be included as hidden values in HTML FORMs and then checked when the FORM is submitted. A mismatch on the tokens causes the request to be denied.

The `csurf` package is designed to be used with Express <https://www.npmjs.com/package/csrf>. In the `notes` directory, run this:

```
$ npm install csurf --save
```

This installs the `csurf` package, recording the dependency in `package.json`.

Then install the middleware like so:

```
import csrf from 'csurf';
...
app.use(cookieParser());
app.use(csrf({ cookie: true }));
```

The `csurf` middleware must be installed following the `cookieParser` middleware.

Next, for every page that includes a FORM, we must generate and send a token with the page. That requires two things, in the `res.render` call we generate the token, sending the token with other data for the page, and then in the view template we include the token as a hidden INPUT on any form in the page. We're going to be touching on several files here, so let's get started.

In `routes/notes.mjs`, add the following as a parameter to the `res.render` call for the `/add`, `/edit`, `/view`, and `/destroy` routes:

```
csrfToken: req.csrfToken()
```

This generates the CSRF token, ensuring it is sent along with other data to the template. Likewise, do the same for the `/login` route in `routes/users.mjs`. Our next task is to ensure the corresponding templates render the token as a hidden INPUT.

In `views/noteedit.hbs` and `views/notedestroy.hbs`, add the following:

```
{#{if user}
  <input type="hidden" name="_csrf" value="{{csrfToken}}">
  ...
{#/if}}
```

This is a hidden INPUT, and whenever the FORM containing this is submitted this value will be carried along with the FORM parameters.

The result is that code on the server generates a token that is added to each FORM. By adding the token to FORMs, we ensure it is sent back to the server on FORM submission. Other software on the server can then match the received token to the tokens that have been sent. Any mismatched token will cause the request to be rejected.

In `views/login.hbs`, make the same addition but adding it inside the FORM like so:

```
<form method='POST' action='/users/login'>
  <input type="hidden" name="_csrf" value="{{csrfToken}}">
  ...
</form>
```

In `views/noteview.hbs`, there's a form for submitting comments. Make this change:

```
<form id="submit-comment" class="well" data-async data-
  target="#rating-modal"
  action="/notes/make-comment" method="POST">
  <input type="hidden" name="_csrf" value="{{csrfToken}}">
  ...
</form>
```

In every case, we are adding a hidden INPUT field. These fields are not visible to the user and are therefore useful for carrying a wide variety of data that will be useful to receive on the server. We've already used hidden INPUT fields in Notes, such as in `noteedit.hbs` for the `doCreate` flag.

This `<input>` tag renders the CSRF token into the FORM. When the FORM is submitted, the `csurf` middleware checks it for the correctness and rejects any that do not match.

In this section, we have learned how to stop an important type of attack, CSRF.

Denying SQL injection attacks

SQL injection is another large class of security exploits, where the attacker puts SQL commands into input data. See <https://www.xkcd.com/327/> for an example.

The best practice for avoiding this problem is to use parameterized database queries, allowing the database driver to prevent SQL injections simply by correctly encoding all SQL parameters. For example, we do this in the SQLite3 model:

```
db.get("SELECT * FROM notes WHERE notekey = ?",
       [ key ] ...);
```

This uses a parameterized string, and the value for `key` is encoded and inserted in the place of the question mark. Most database drivers have a similar feature, and they already know how to encode values into query strings. Even if a miscreant got some SQL into the value of `key`, because the driver correctly encodes the contents of `key` the worst that will result is an SQL error message. That automatically renders inert any attempted SQL injection attack.

Contrast this with an alternative we could have written:

```
db.get(`SELECT * FROM notes WHERE notekey = ${key}` , ...);
```

The template strings feature of ES6 is very tempting to use everywhere. But it is not appropriate in all circumstances. In this case, the database query parameter would not be screened nor encoded, and if a miscreant can get a custom string to that query it could cause havoc in the database.

In this section, we learned about SQL injection attacks. We learned that the best defense against this sort of attack is the coding practice all coders should follow anyway, namely to use parameterized query methods offered by the database driver.

In the next section, we will learn about an effort in the Node.js community to screen packages for vulnerabilities.

Scanning for known vulnerabilities in Node.js packages

Built-in to the npm command-line tool is a command, `npm audit`, for reporting known vulnerabilities in the dependencies of your application. To support this command is a team of people, and software, who scan packages added to the npm registry. Every third-party package used by your application is a potential security hole.

It's not just that a query against the application might trigger buggy code, whether in your code or third-party packages. In some cases, packages that explicitly cause harm have been added to the npm registry.

Therefore the security audits of packages in the npm registry are extremely helpful to every Node.js developer.

The audit command consults the vulnerability data collected by the auditing team and tells you about vulnerabilities in packages your application uses.

When running `npm install`, the output might include a message like this:

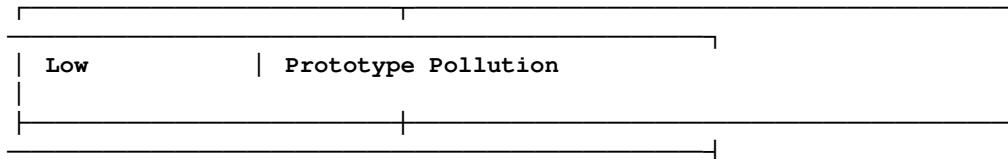
```
found 8 vulnerabilities (7 low, 1 moderate)
  run `npm audit fix` to fix them, or `npm audit` for details
```

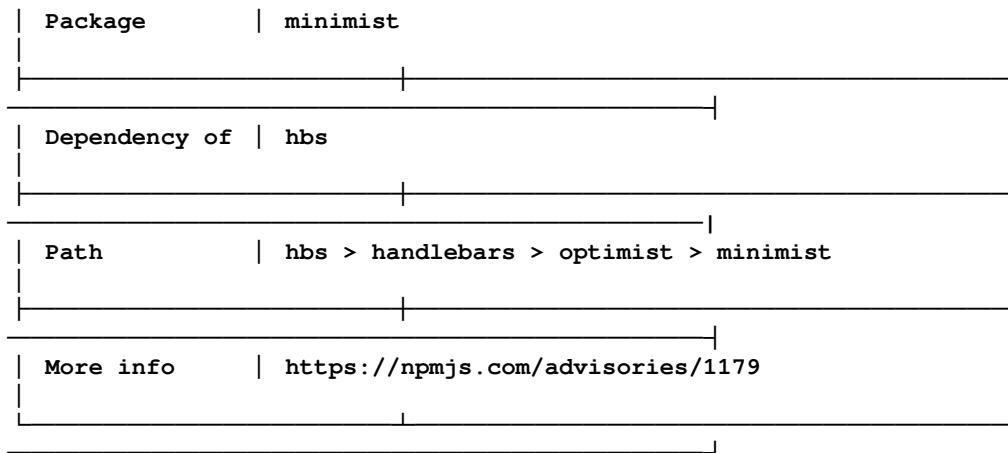
This tells us there are eight known vulnerabilities among the packages currently installed. Each vulnerability is assigned a criticality on this scale (<https://docs.npmjs.com/about-audit-reports>):

- *Critical*: Address immediately
- *High*: Address as quickly as possible
- *Moderate*: Address as time allows
- *Low*: Address at your discretion

In this case, running `npm audit` tells us that every one of the low-priority issues is in the `minimist` package. For example, the report includes this:

```
# Run npm install hbs@4.1.1 to resolve 1 vulnerability
```





In this case, `minimist` is reported because `hbs` uses `handlebars`, which uses `optimist`, which uses `minimist`. There are six more instances where `minimist` is used by some package that's used by another package that our application is using.

In this case, we're given a recommendation, to upgrade to `hbs@4.1.1`, because that release results in depending on the correct version of `minimist`.

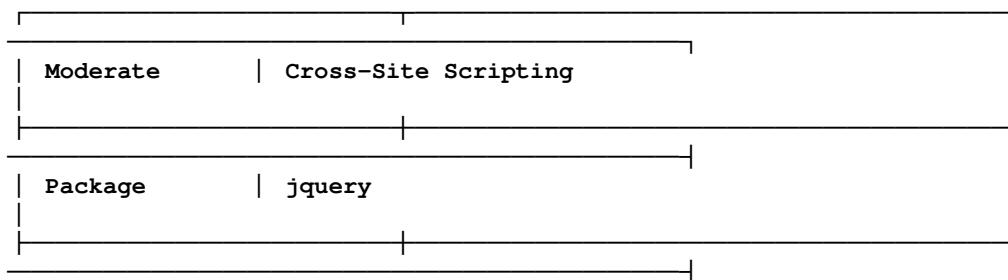
In another case, the chain of dependencies is this:

```
sqlite3 > node-pre-gyp > rc > minimist
```

In this case, no recommended fix is available because none of these packages have released a new version that depends on the correct version of `minimist`. The recommended solution for this case is to file issues with each corresponding package team requesting they update their dependencies to the later release of the offending package.

In the last case, it is our application that directly depends on the vulnerable package:

```
# Run npm update jquery --depth 1 to resolve 1 vulnerability
```



Dependency of	jquery
Path	jquery
More info	https://npmjs.com/advisories/1518

Therefore it is our responsibility to fix this problem because it is in our code. The good news is that this particular package is not executed on the server side since jQuery is a client-side library that just so happens to be distributed through the npm repository.

The first step is to read the advisory to learn what the issue is. That way, we can evaluate for ourselves how serious this is, and what we must do to correctly fix the problem.

What's not recommended is to blindly update to a later package release just because you're told to do so. What if the later release is incompatible with your application? The best practice is to test that the update does not break your code. You may need to develop tests that illustrate the vulnerability. That way, you can verify that updating the package dependency fixes the problem.

In this case, the advisory says that jQuery releases before 3.5.0 have an XSS vulnerability. We are using jQuery in Notes because it is required by Bootstrap, and on the day we read the Bootstrap documentation we were told to use a much earlier jQuery release. Today, the Bootstrap documentation says to use jQuery 3.5.1. That tells us the Bootstrap team has already tested against jQuery 3.5.1, and we are therefore safe to go ahead with updating the dependency.

In this section, we have learned about the security vulnerability report we can get from the npm command-line tool. Unfortunately for Yarn users, it appears that Yarn doesn't support this command. In any case, this is a valuable resource for being warned about known security issues.

In the next section, we'll learn about the best practices for cookie management in Express applications.

Using good cookie practices

Some nutritionists say eating too many sweets, such as cookies, is bad for your health. Web cookies, however, are widely used for many purposes including recording whether a browser is logged in or not. One common use is for cookies to store session data to aid in knowing whether someone is logged in or not.

In the Notes application, we're already following the good practices described in the Express security guidelines:

- We're using an Express session cookie name different from the default shown in the documentation.
- The Express session cookie secret is not the default shown in the documentation.
- We use the `express-session` middleware, which only stores a session ID in the cookie, rather than the whole session data object.

Taken together, an attacker can't exploit any known vulnerability that relies on the default values for these items. While it is convenient that many software products have default values, such as passwords, those defaults could be security vulnerabilities. For example, the default Raspberry Pi login/password is *pi* and *raspberry*. While that's cute, any Raspbian-based IoT device that's left with the default login/password is susceptible to attack.

But there is more customization we can do to the cookie used with `express-session`. That package has a few options available for improving security.

See <https://www.npmjs.com/package/express-session>, and then consider this change to the configuration:

```
app.use(session({  
    store: sessionStore,  
    secret: sessionSecret,  
    resave: true,  
    saveUninitialized: true,  
    name: sessionCookieName,  
    secure: true,  
    maxAge: 2 * 60 * 60 * 1000 // 2 hours  
}));
```

These are additional attributes that look useful. The `secure` attribute requires that cookies be sent ONLY over HTTPS connections. This ensures the cookie data is encrypted by HTTPS encryption. The `maxAge` attribute sets an amount of time that cookies are valid, expressed in milliseconds.

Cookies are an extremely useful tool in web browsers, even if there is a lot of over-hyped worry about what websites do with cookies. At the same time, it is possible to misuse cookies and create security problems. In this section, we learned how to mitigate risks with the session cookie.

In the next section, we'll review the best practices for AWS ECS deployment.

Hardening the AWS EC2 deployment

There is an issue left over from Chapter 12, *Deploying a Docker Swarm to AWS EC2 with Terraform*, which is the security group configuration for the EC2 instances. We configured the EC2 instances with permissive security groups, and it is better for them to be strictly defined. We rightly described that, at the time, as not the best practice, and promised to fix the issue later. This is where we do so.

In AWS, remember that a security group describes a *firewall* that allows or disallows traffic based on the IP port and IP address. This tool exists so we can decrease the potential attack surface miscreants have to gain illicit access to our systems.

For the `ec2-public-sg` security group, edit `ec2-public.tf` and change it to this:

```
resource "aws_security_group" "ec2-public-sg" {
  name = "${var.project_name}-public-sg"
  description = "allow inbound access to the EC2 instance"
  vpc_id = aws_vpc.notes.id

  ingress {
    description = "SSH"
    protocol = "TCP"
    from_port = 22
    to_port = 22
    cidr_blocks = [ "0.0.0.0/0" ]
  }

  ingress {
    description = "HTTP"
    protocol = "TCP"
    from_port = 80
    to_port = 80
    cidr_blocks = [ "0.0.0.0/0" ]
  }

  ingress {
    description = "HTTPS"
    protocol = "TCP"
    from_port = 443
    to_port = 443
    cidr_blocks = [ "0.0.0.0/0" ]
  }
}
```

```
protocol = "TCP"
from_port = 443
to_port = 443
cidr_blocks = [ "0.0.0.0/0" ]
}

ingress {
  description = "Redis"
  protocol = "TCP"
  from_port = 6379
  to_port = 6379
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "Docker swarm management"
  from_port = 2377
  to_port = 2377
  protocol = "tcp"
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "Docker container network discovery"
  from_port = 7946
  to_port = 7946
  protocol = "tcp"
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "Docker container network discovery"
  from_port = 7946
  to_port = 7946
  protocol = "udp"
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "Docker overlay network"
  from_port = 4789
  to_port = 4789
  protocol = "udp"
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

egress {
  description = "Docker swarm (udp)"
```

```
    from_port = 0
    to_port = 0
    protocol = "udp"
    cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

egress {
    protocol = "-1"
    from_port = 0
    to_port = 0
    cidr_blocks = [ "0.0.0.0/0" ]
}
}
```

This declares many specific network ports used for specific protocols. Each rule names the protocol in the `description` attribute. The `protocol` attribute says whether it is a UDP or TCP protocol. Remember that TCP is a stream-oriented protocol that ensures packets are delivered, and UDP, by contrast, is a packet-oriented protocol that does not ensure delivery. Each has characteristics making them suitable for different purposes.

Something missing is an `ingress` rule for port 3306, the MySQL port. That's because the `notes-public` server will not host a MySQL server based on the placement constraints.

Another thing to note is which rules allow traffic from public IP addresses, and which limit traffic to IP addresses inside the VPC. Many of these ports are used in support of the Docker swarm, and therefore do not need to communicate anywhere but other hosts on the VPC.

An issue to ponder is whether the SSH port should be left open to the entire internet. If you, or your team, only SSH into the VPC from a specific network, such as an office network, then this setting could list that network. And because the `cidr_blocks` attribute takes an array, it's possible to configure a list of networks, such as a company with several offices each with their own office network.

In `ec2-private.tf`, we must make a similar change to `ec2-private-sg`:

```
resource "aws_security_group" "ec2-private-sg" {
    name = "${var.project_name}-private-sg"
    description = "allow inbound access to the EC2 instance"
    vpc_id = aws_vpc.notes.id

    ingress {
        description = "SSH"
        protocol = "TCP"
```

```
from_port = 22
to_port = 22
cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "HTTP"
  protocol = "TCP"
  from_port = 80
  to_port = 80
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "MySQL"
  protocol = "TCP"
  from_port = 3306
  to_port = 3306
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "Redis"
  protocol = "TCP"
  from_port = 6379
  to_port = 6379
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "Docker swarm management"
  from_port = 2377
  to_port = 2377
  protocol = "tcp"
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "Docker container network discovery"
  from_port = 7946
  to_port = 7946
  protocol = "tcp"
  cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
  description = "Docker container network discovery"
  from_port = 7946
```

```
    to_port = 7946
    protocol = "udp"
    cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

ingress {
    description = "Docker overlay network"
    from_port = 4789
    to_port = 4789
    protocol = "udp"
    cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

egress {
    description = "Docker swarm (udp)"
    from_port = 0
    to_port = 0
    protocol = "udp"
    cidr_blocks = [ aws_vpc.notes.cidr_block ]
}

egress {
    protocol = "-1"
    from_port = 0
    to_port = 0
    cidr_blocks = [ "0.0.0.0/0" ]
}
}
```

This is largely the same but for some specific differences. First, because the private EC2 instances can have MySQL databases, we have declared a rule for port 3306. Second, all but one of the rules restrict traffic to IP addresses inside the VPC.

Between these two security group definitions, we have strictly limited the attack surface of the EC2 instances. This will throw certain barriers in the path of any miscreants attempting to intrude on the Notes service.

While we've implemented several security best practices for the Notes service, there is always more that can be done. In the next section, we'll discuss where to learn more.

AWS EC2 security best practices

At the outset of designing the Notes application stack deployment, we described a security model that should result in a highly secure deployment. Are we the kind of security experts that can design a secure deployment infrastructure on the back of a napkin? Probably not. But the team at AWS does employ engineers with security expertise. When we turned to AWS EC2 for deployment, we learned it offered a wide range of security tools we hadn't considered in the original plan, and we ended up with a different deployment model.

In this section, let's review what we did and also review some additional tools available on AWS.

The AWS **Virtual Private Cloud (VPC)** contains many ways to implement security features, and we used a few of them:

- *Security Groups* act as a firewall with strict controls over the traffic that can enter or leave the things protected by a Security Group. Security Groups are attached to every infrastructure element we used, and in most cases, we configured them to allow only the absolutely necessary traffic.
- We ensured the database instances were created within the VPC, rather than hosted on the public internet. This hides the databases from public access.

While we did not implement the originally envisioned segmentation, there are enough barriers surrounding Notes that it should be relatively safe.

In reviewing the AWS VPC security documentation, there are a few other facilities that are worth exploring.



Security in AWS Virtual Private Cloud: <https://docs.aws.amazon.com/vpc/latest/userguide/security.html>.

In this section, you've had a chance to review the security of the application that was deployed to AWS ECS. While we did a fairly good job, there is more that can be done to exploit tools offered by AWS to beef up the internal security of the application.

With that, it's time to close out this chapter.

Summary

In this chapter, we've covered an extremely important topic, application security. Thanks to the hard work of the Node.js and Express communities, we've been able to tighten the security simply by adding a few bits of code here and there to configure security modules.

We first enabled HTTPS because it is now a best practice, and has positive security gains for our users. With HTTPS, the browser session is authenticated to positively identify the website. It also protects against man-in-the-middle security attacks, and encrypts communications for transmission across the internet, preventing most snooping.

The `helmet` package provides a suite of tools to set security headers that instruct web browsers on how to treat our content. These settings prevent or mitigate whole classes of security bugs. With the `csurf` package, we're able to prevent **cross-site request forgery (CSRF)** attacks.

These few steps are a good start for securing the Notes application. But you should not stop here because there is a never-ending set of security issues to fix. None of us can neglect the security of the applications we deploy.

Over the course of this book, the journey has been about learning the major life cycle steps required to develop and deploy a Node.js web application. This started from the basics of using Node.js, proceeded to an application concept to develop, and from there we covered every stage of developing, testing, and deploying that application.

Throughout the book, we've learned how advanced JavaScript features such as `async` functions and ES6 modules are used in Node.js applications. To store our data, we learned how to use several database engines, and a methodology to make it easy to switch between engines.

Mobile-first development is extremely important in today's environment, and to fulfill that goal, we learned how to use the Bootstrap framework.

Real-time communication is expected on a wide variety of websites because advanced JavaScript capabilities mean we can now offer more interactive services in our web applications. To fulfill that goal, we learned how to use the Socket.IO real-time communications framework.

Deploying application services to cloud hosting is widely used, both for simplifying the system setup and to scale services to meet the demands of our user base. To fulfill that goal, we learned to use Docker, and then we learned how to deploy Docker services to AWS ECS using Terraform. We not only used Docker for production deployment but for deploying a test infrastructure, within which we can run unit tests and functional tests.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Full-Stack React Projects - Second Edition

Shama Hoque

ISBN: 978-1-83921-541-4

- Extend a basic MERN-based application to build a variety of applications
- Add real-time communication capabilities with Socket.IO
- Implement data visualization features for React applications using Victory
- Develop media streaming applications using MongoDB GridFS
- Improve SEO for your MERN apps by implementing server-side rendering with data
- Implement user authentication and authorization using JSON web tokens
- Set up and use React 360 to develop user interfaces with VR capabilities
- Make your MERN stack applications reliable and scalable with industry best practices



Webpack 5 Up and Running

Tom Owens

ISBN: 978-1-78995-440-1

- Get to grips with Webpack bundle configuration and set options
- Optimize your JavaScript projects by using code-splitting techniques
- Efficiently handle dependencies in complex web apps
- Break down complex problems into simple ones using advanced debugging procedures
- Master version migration and deployment hurdles
- Effectively deploy the Webpack application using Babel

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

*

*BSD

Node.js, installing from package management systems 37

A

absolute module identifiers 101

access control list (ACL) 530

administrator-privileged PowerShell

opening, on Windows 39

Advanced RISC Machine (ARM) 547

algorithmic refactoring 170, 172, 173

Amazon Machine Image (AMI) 547

Amazon Web Services (AWS)

EC2 instance, launching on 552, 553

gateway, configuring 537, 539, 540

infrastructure, deploying with Terraform to 540, 542, 543, 545

signing up with 518, 519

subnet resources, configuring 537, 539, 540

URL 518

assert module 610, 611

async arrow functions

writing 56, 57

async functions

error-first-callback, converting to 58

exploring, in Express router functions 188, 189, 190, 191

integrating, with Express router functions 193, 194, 195

reference link 189

Atom

URL 51

authentication best practices

reference link 363

authentication tokens

storing 363, 365

AuthNet

exploring 485, 486, 487

automating test results

reporting 644, 645

availability zones (AZs) 529

AWS account

services, searching 519

way, searching around 520

AWS authentication credentials

used, for setting up AWS CLI 520, 521, 522

AWS CLI commands

environment variables 563, 564

AWS CLI

configuring 518, 519

setting up, with AWS authentication

credentials 520, 521, 522

AWS EC2 deployment

best practices 703, 705, 707

AWS EC2 key-pair file

handling 554

AWS EC2 security

best practices 708

AWS EC2

Docker Swarm cluster, setting up on 545, 546

domain name, assigning for application

deployed 668

AWS ECR

Docker images, pushing to 564, 566, 568, 569

AWS infrastructure

creating, with Terraform 531, 532, 533, 534

overview 528, 529, 530, 531

overview, to be deployed 529

AWS Virtual Private Cloud, security

reference link 708

- AWS Virtual Private Cluster
configuring, with Terraform 534, 535, 536, 537
- ## B
- Babel
 JavaScript experimental features, using with 66, 68, 69, 70
 URL 66
- backend service
 creating 170
- base URL 140
- Base
 URL 237
- bcrypt documentation
 reference link 374
- BIOS configuration
 reference link 461
- Bluebird
 reference link 189
- body-parser module
 reference link 156
- Bootstrapjumbotron component 357
- Bootstrap
 reference link 253
- Bootswatch
 URL 258
- breakpoint 231
- bugs
 fixing, by package dependencies update 119
- business logic 196
- ## C
- capture-console package
 reference link 270
- central processing unit (CPU) 529
- Certbot
 reference link 670
- Chai tool 612
- class inheritance 133, 134
- Classless Inter-Domain Routing (CIDR) 529
- CLI documentation
 reference link 109
- Clickjacking 693
- comma-separated values (CSV) 521
- command-line interface (CLI) 323
- command-line programs, installation
 local, installing 122
- command-line programs
 installing, globally 122
- command-line tool
 creating, to administer user authentication server 323, 325, 326
 creating, to test user authentication server 323, 325, 326
- Commander
 reference link 323
- CommonJS module system
 URL 76
- CommonJS modules
 ES6 modules, using with import() function 85, 86
 implementation details, hiding in 87, 88
 using, with ES6 modules 83, 84
- Community Edition (CE) 550
- compiler 66
- computationally intensive code 168, 169, 170
- connect-redis
 reference link 512
- Content-Security-Policy (CSP) header
 about 687
 setting up, with Helmet 687, 688, 689
- ContentSecurityPolicy
 configuring 689, 690
- cookie management
 best practices 702
- cookie-parser module
 reference link 156
- core modules 101
- create, read, update, and delete (CRUD) 335, 410
- Create, Read, Update, and Delete/Destroy (CRUD) model 196
- Cronginx container
 adding, to HTTPS on Notes 672, 674
- Cronginx
 reference link 672
- cross-env package
 reference link 152

cross-env tool
 reference link 613

Cross-Site Request Forgery (CSRF) attacks
 addressing 696, 697

cross-site-scripting (XSS) attacks
 about 695
 mitigating, with Helmet 695

cross-var package
 URL 563

CSS Grids
 about 238
 URL 238

CSS Selectors
 reference link 653

CSS-Tricks
 URL 231

csurf package
 reference link 696

Cursor class
 reference link 308

customized Bootstrap
 building 253, 254, 255, 257, 258
 third-party custom Bootstrap themes, using
 258, 259, 260

D

data model
 for storing messages 405, 407, 408, 409,
 410

data serialization language 293

data storage model, Notes application
 data hiding, in ES-2015 class definitions 205,
 206, 207
 implementing 203, 205

data storage
 asynchronous code 262

database connections
 closing, on process exits 283, 284

debug package
 reference link 267

debug tracing
 enabling, in Socket.IO code 399

deep import module identifier
 overriding 103

deep import module specifiers
 using 102

deep import path 102

deployed Node.js application
 HTTPS, implementing in Docker 666

deployed Notes application
 testing 600

directories
 adding, on EC2 host 676

DNS Prefetch
 about 691
 reference link 691

Docker bridge network 470

Docker Compose
 about 459
 reference link 498, 500
 used, for building Notes application 502, 503,
 504, 506
 used, for deploying Notes stack 498, 500,
 501
 used, for executing Notes application 502,
 503, 505, 506
 used, for managing multiple containers 497,
 498

Docker engine 459

Docker Hub
 reference link 464

Docker image
 process, defining to build 564, 566, 568, 569
 pushing, to AWS ECR 564, 566, 568, 569

Docker orchestrator 497

Docker stack file
 creating, for deployment to Docker Swarm
 569, 570
 creating, from Notes Docker compose file
 571, 572, 573

Docker Swarm cluster
 features 546
 setting up, on AWS EC2 545, 546
 URL 546

Docker Swarm hosted
 remote control access, setting up on EC2
 instance 558, 559, 560, 561

Docker Swarm
 container placement across 574, 575, 576
 data persistence 580, 581, 582

Docker swarm
 deploying 677, 678

Docker Swarm
 Docker stack file, creating for deployment to 569, 570

EC2 instances, configuring and connecting to 583, 584, 586, 587

EC2 instances, provisioning 582

MongoDB, setting up 635, 636, 637, 638

Notes stack file, deploying to 594

Notes stack, deploying to 590, 592, 594, 596

Notes stack, preparing to deploy to 595, 596

secrets, configuring 576, 577, 578, 579

semi-automatic initialization, implementing 587, 588, 589, 590

testing 554, 555, 556, 557

tests, executing 632, 633, 634, 635

using, to deploy test infrastructure 628, 629, 630, 631

using, to manage test infrastructure 627

Docker
 authentication service, setting up 464, 465

 configuring 547, 548, 550

 HTTPS, implementing for deployed Node.js application 667

 installation links 460

 installation, requisites 460

 installing, on laptop 460, 461

 installing, on Windows/macOS 461

 Let's Encrypt, using to implement HTTPS for Notes 672

 NGINX, using to implement HTTPS for Notes 672

 using 462, 463, 464

Dockerfiles
 reference link 479

domain name system (DNS) 11, 668

Domain Name System (DNS) 536

domain name
 assigning, for application deployed on AWS EC2 668, 669

domain
 registering, to create NGINX configuration with Let's Encrypt 675, 676

 registering, with Let's Encrypt 678, 679, 680

dotenv package
 about 363

 reference link 363

E

EC2 cluster
 deploying 677, 678

EC2 host
 directories, adding on 676

EC2 instance
 adding 547, 548, 550

 configuring, and connecting to Docker Swarm 583, 584, 586, 587

 launching, on AWS 551, 553

 provisioning, for full Docker Swarm 582

 remote control access, setting up to Docker Swarm hosted 558, 559, 560, 561

EC2 key-pair
 creating 527, 528

ECMAScript 2015 (ES-2015) 342

ECMAScript
 used, for advancing Node.js 62, 64, 65, 66

ECR repositories
 setting up, for Notes Docker images 561, 563

Elastic IP (EIP) 538

encrypted password
 implementing, in Notes application 379, 380

environment variables, setting in Windows
 cmd.exe command line
 setting, in Windows cmd.exe command line 151

environment variables
 setting, in Windows cmd.exe command line 152

error handling 161, 162

error-first-callback
 converting, to async functions 58

 converting, to Promise paradigm 58

ES2015 (ES6) Modules, using with Babel 6
 reference link 90

ES2015 multiline strings 142, 143

ES2015 template strings 142, 143

ES2015/2016/2017/2018 JavaScript code

deploying 30
ES6 import statement 94
ES6 modules
 dynamic importing 275, 276, 277, 278
 implementation details, hiding in 87, 88
 missing `__dirname` variable, computing 82, 83
 objects, injecting 81, 82
 reference link 92
 supporting, on Node.js versions 90, 91
 using, from CommonJS modules with `import()` 86
 using, from CommonJS modules with `import()` function 85
 using, with CommonJS modules 83, 84
ES6/ES-2015 module format
 examining 77, 80, 81
esm module
 reference link 91
esm package
 reference link 91
EventEmitter
 class 135, 136
 theory 137, 138
 used, for receiving events 132
 used, for sending events 132
events
 receiving, with EventEmitter 132, 133
 sending, with EventEmitter 132, 133
Express application
 creating, to compute Fibonacci numbers 162, 163, 164, 165, 166, 167, 170
 designing, in MVC paradigm 196
 Helmet, using for across-the-board security 686
 REST backend service, calling from 176, 177
 theming 222, 223, 224
 used, for implementing REST server 177, 179
Express router functions
 async functions 188, 189, 190, 191
 async functions, integrating with 193, 194, 195
 error handling 191, 192
 Promises 188, 189, 190, 191, 192, 193
Express session stores
 reference link 510
express-session cookie
 reference link 702
Express
 default application 153, 154, 155
 environment variables, setting in Windows
 cmd.exe command line 151, 152
 error handling 161, 162
 middleware functions 156, 158
 reference link 147, 153
 request handlers, contrasting from middleware functions 159, 160
 Socket.IO, initializing 386, 387, 388
 using 147, 149, 151
ExpressJS Wiki
 reference link 146
Extended Page Tables (EPT) 461

F

Feather Icons
 URL 242

Feature-Policy header
 used, for controlling enabled browser features with Helmet 692

Fibonacci application
 about 153
 refactoring, to call REST service 181, 182, 184

Fibonacci numbers
 computationally intensive code 168, 169
 computing, to create Express application 162, 163, 164, 165, 166, 167, 170
 Node.js event loop 168, 169

file module 92, 93, 94

filesystem
 ES6 modules, dynamically importing 275, 276, 277, 278
 Notes application, executing 278
 Notes, storing 271, 274, 275

Flexbox
 about 238
 URL 238

Foundation
 URL 237

frameguard module

reference link 693
frontend headless browser
testing, with Puppeteer 646
FrontNet
creating, for Notes application 487, 488
fs-extra
reference link 273

G

General Purpose I/O (GPIO) 13
generator functions
reference link 194
Git, for Windows
reference link 49
glue service 11
grave accent 142

H

Handlebars
reference link 688
Helmet
Content-Security-Policy (CSP) header, setting
up 687, 688, 689
reference 688
reference link 686
used, for mitigating XSS attacks 695
used, for setting up X-DNS-Prefetch-Control
header 691
using, for across-the-board security in
Express application 686
using, to control enabled browser features
with Feature-Policy header 692
using, to remove X-Powered-By header 693
using, to setting up X-Frame-Options header
693
helpful documentation
accessing 109
hexy program 60
Homebrew
Node.js, installing on macOS with Homebrew
36, 37
URL 36
HTTP Client requests
creating 174, 176
HTTP server application

about 138, 140, 141, 142
ES2015 multiline strings 142, 143
ES2015 template strings 142, 143
HTTP Sniffer
used, for listening to HTTP conversation 144
HTTPS
Cronginx container, adding on Notes 672,
674
implementing, for Notes with Let's Encrypt in
Docker 672
implementing, for Notes with NGINX in
Docker 672
implementing, in Docker for deployed Node.js
application 666
improving, with Strict Transport Security 694
testing, for Notes application 685, 686

I

IAM user
account, creating 523, 524, 525, 526
group, creating 523
groups, creating 524, 525, 526
roles, creating 523, 524, 525, 526
import features
used, for finding Node.js module 92
used, for loading Node.js module 92
import statement
used, for loading Node.js module 105, 106,
107
import() function
used, for loading Node.js module 105, 106,
107
installed module
comparing, with installed package 96
installed package
comparing, with installed module 96
finding, in file system 97, 98
multiple versions, handling 99
searching 100, 101
inter-user chat, for Notes application
data model, for storing messages 405, 407,
408, 409, 410
implementing 404
messages, adding to Notes router 410
note view template, modifying for messages

413
internet gateway 529
Internet Protocol version 4 (IPv4) 529
internet relay chat (IRC) 11
internet service provider (ISP) 530
iteration protocol
reference link 194

J

JavaScript classes 133, 134
JavaScript experimental features
using, with Babel 66, 68, 69
JavaScript Object Notation (JSON) 343, 524
JavaScript
about 15
advances, embracing 28, 29, 30
jQuery
reference link 233
JSON module
using 88, 89

K

Kota
reference link 146

L

Let's Encrypt Authority X3 685
Let's Encrypt
certificates, used for implementing NGINX
HTTPS configuration 680, 681, 683, 684
reference link 670
usage, planning 670, 671
used, for creating NGINX configuration to
register domains 675, 676
used, for registering domain 678, 679, 680
using, in Docker to implement HTTPS for
Notes 672
LevelDB datastore
used, for storing Notes 279, 280, 282, 283
Linux deployment, for Node.js services
about 427, 428, 429
deployed user authentication service, testing
438
failure to launch Multipass instances on
Windows, handling 431, 432

Multipass, installing 429, 431
script execution in PowerShell, on Windows
440
server, provisioning for Notes service 441,
442, 443
server, provisioning for user authentication
service 433, 434, 435, 437

Linux

Node.js, installing from package management
systems 37
log rotation 263
logging
about 262
request, with morgan package 264, 266
login functionality
ability, testing to add Notes 655, 656, 657,
659
testing, in Notes 652, 653, 654, 655
login support, for Notes application
executing, with user authentication 358, 359,
360
implementing 339
login and logout routing functions,
implementing 346
login and logout routing functions,
incorporating 344, 347, 348
login/logout, modifying in app.js 349, 350
login/logout, modifying in routes/index.mjs
351
login/logout, modifying in routes/notes.mjs
352, 353
template changes, viewing for login/logout
354, 355, 356, 357
user authentication REST API, accessing
340, 341, 342, 343
logout functionality
ability, testing to add Notes 655, 656, 657,
659
testing, in Notes 652, 653, 654, 655
long term support (LTS) 50
Loopback
reference link 184
LSB-style init script
reference link 428

M

- macOS
 - developer tools, installing 42
 - Node.js, installing with Homebrew 36, 37
 - used, for installing Node.js with MacPorts 35
 - MacPorts
 - Node.js, installing on macOS with 35
 - URL 35
 - maxiservices
 - developing, with Node.js 31
 - memory management unit (MMU) 461
 - messages
 - debugging 267, 268, 269
 - microservices
 - advantages 32
 - developing, with Node.js 31
 - Microsoft Visual Studio Code
 - URL 51
 - middleware functions
 - request handlers, contrasting 159, 160
 - middleware
 - reference link 157
 - mime module
 - reference link 102
 - mobile-first design, for Notes application
 - about 238, 239
 - add/edit note form, cleaning up 249, 251
 - Bootstrap grid foundation, laying 239, 240, 241
 - delete-note window, cleaning up 251, 252
 - icon libraries, using 242, 243
 - note viewing experience, cleaning up 248, 249
 - Notes list, enhancing on front page 245, 246
 - responsive page header navigation bar 243, 245
 - responsive page structure 241, 242
 - visual appeal, enhancing 242, 243
 - mobile-first paradigm 230
 - Mocha tool
 - about 612
 - URL 613
 - Model, View, and Controller (MVC)
 - Express application, designing 196, 197
 - Model-View-Controller (MVC) 165
- module identifiers
 - absolute module identifiers 101
 - core modules 101
 - relative elative module identifiers 101
 - reviewing 101, 102
 - top-level module identifiers 101
 - MongoDB (or MySQL), Express, Angular, and Node.js (MEAN) 303
 - MongoDB database
 - used, for executing Notes application 309, 310
 - MongoDB model
 - for Notes application 304, 307, 308
 - MongoDB Node.js driver
 - reference link 305
 - MongoDB
 - Notes, storing 303
 - Notes, testing 635
 - reference link 635
 - setting up, in Docker Swarm 635, 636, 637, 638
 - URL 303
 - Mongoose
 - URL 303
 - morgan package
 - used, for logging request 264, 266
 - Morgan request logger
 - reference link 156
 - multi-factor authentication (MFA) 518
 - Multipass
 - installing 429, 430, 431
 - PM2 setup, scripting 450, 451, 453, 454
 - multiple Notes service instances
 - used, for testing session management 508, 509
 - Multipurpose Internet Mail Extensions (MIME) 161
 - MySQL container
 - creating, for authentication service 472, 473, 474, 475
 - creating, for Notes application 488, 489
 - launching, in Docker 465, 466, 467, 468
 - securing 476, 477, 478, 479

N

- named pipe 202
- NAT gateway 530
- native code modules
 - installing, requisites 48, 49
- negative test 641
 - implementing, with Puppeteer 659
 - login, with bad user ID 660, 661
 - response, testing to bad URL 661, 662
- Netwide Assembler (NASM)
 - about 45
 - URL 45
- network address translation (NAT) 529
- NGINX configuration
 - creating, to register domains with Let's Encrypt 675, 676
 - reference link 674
- NGINX HTTPS configuration
 - implementing, with Let's Encrypt certificates 680, 681, 683, 684
- NGINX Plus product
 - reference link 674
- NGINX
 - using, in Docker to implement HTTPS for Notes 672
- Node green website
 - URL 63
- node-webkit (NW.js) 13
- Node.js commands
 - running 52
 - testing 52
- Node.js core modules
 - about 94, 95
 - reference link 94
- Node.js distribution
 - installing, from nodejs.org 39, 40
- Node.js Docker image
 - reference link 480
- Node.js documentation
 - reference link 92
- Node.js event loop 168, 169, 170
- Node.js instances
 - installing, with nvm 45, 47
- Node.js module
 - CommonJS modules, using with ES6 modules 83, 84
 - deep import module specifiers, using 102
 - defining 73
 - directory structure, using 95
 - ES6 modules, supporting on Node.js versions 90, 91
 - ES6 modules, using with CommonJS modules 83, 84
 - ES6/ES-2015 module format, examining 77, 80, 81
 - file module 92, 93, 94
 - finding, with require and import features 92
 - format, examining 73, 74, 75, 76, 77
 - implementation details, hiding with encapsulation in CommonJS and ES6 modules 87, 88
 - installed package, comparing with installed module 96
 - installed package, finding in file system 97, 98
 - installed package, searching 100, 101
 - JSON module, using 88, 89
 - loading, with import statement 105, 106, 107
 - loading, with import() function 105, 106, 107
 - loading, with require and import features 92
 - loading, with require function 105, 106, 107
 - module identifiers, reviewing 101, 102
 - multiple versions, handling of installed package 99
 - pathnames, reviewing 101, 102
 - project directory structure, example 103, 105
 - reference link 285
- Node.js MongoDB driver
 - reference link 304
- Node.js package management system 108
- Node.js package manager
 - using 60, 61
- Node.js package
 - initializing, with npm init 109, 110
 - known vulnerabilities, scanning 699, 701
- Node.js packaged binaries
 - executing, with npx 62
- Node.js processes
 - managing, by setting up PM2 448

Node.js project
initializing, with npm init 109, 110

Node.js services
Linux deployment 427, 428, 429
Multipass, installing 430
script execution in PowerShell, on Windows 441
server, provisioning for user authentication service 436

Node.js version compatibility
declaring 127

Node.js versions
ES6 modules, supporting 90, 91
policy 50, 51
selection, for using 50, 51

Node.js's command-line tools
using 52

Node.js, installing from source on POSIX-like systems
developer tools, installing on macOS 42
prerequisites, installing 41

Node.js
about 17
advancing, with ECMAScript 62, 64, 65, 66
asynchronous requests 20, 21
asynchronous-programming model 16
build tools 12
capabilities 11, 12
complexity, handling 19, 20
debuggers, selecting 51, 52
desktop applications 13
editors, selecting 51, 52
environmental impact 27
event-driven architecture 17, 18, 19
event-driven model 16
installing, from source for all POSIX-like systems 43, 44
installing, from source on POSIX-like systems 40, 41
installing, from source on Windows 45
installing, in WSL 38
installing, on *BSD from package management systems 37
installing, on Linux from package management systems 37

installing, on macOS with Homebrew 36, 37
installing, on macOS with MacPorts 35
installing, on Windows from package management systems 37
installing, with package managers 35

Internet of things (IoT) 13
maxiservices, developing 31
microservice architecture 16
microservices, developing 31
mobile applications 13
need for 12, 13, 14
overhead costs 27
overview 10, 11
performance 22, 23, 24
popularity 14, 15
scalability disaster 24, 26, 27
server utilization 27
system requisites 34, 35
URL 59
utilization 22, 23, 24
web UI testing 12

nodejs.org
Node.js distribution, installing from 39, 40
URL 39

note view template, for messages
composing 413, 415, 416
deleting 420
displaying 417, 420
executing 421, 422
modifying 413
passing 421, 422

Notes application 215

Notes application stack
correct launch, verifying 597, 599
database services, launch failure diagnosing 599, 600
inability to log in, diagnosing with Twitter credentials 603, 604

instances, scaling 604, 605, 607
logging in, with regular account 601, 602
Puppeteer test, creating 649
scaling, Redis used 507
used, for creating Puppeteer test 648, 650, 651

Notes application

app.mjs 199, 200, 202
architecture 426
building, with Docker Compose 502, 503, 504, 506
commenting 405
creating 197
 data storage model, implementing 203, 205
 deployment considerations 427
 dockerizing 489, 490, 491, 492, 493, 494, 496
 encrypted password, implementing 379, 380
 executing 381
 executing, with Docker Compose 502, 503, 505
 executing, with filesystem 278, 279
 executing, with MongoDB database 309, 310
 executing, with Sequelize 301, 302
 executing, with SQLite3 292
 FrontNet, creating 487, 488
 generated router module, rewriting as ES6 module 198, 199
 home page, creating 209, 210, 211
 in-memory datastore, implementing 208
 inter-user chat, implementing 405
 issues 228, 229
 login support, implementing 339
 MongoDB model, using 304, 307, 308
 multiple instances, executing 224, 225, 226
 MySQL container, creating 488, 489
 new note, adding 213, 214, 216, 217
 note, editing 219, 220
 notes, deleting 220, 221, 222
 notes, viewing 217, 218, 219
 Sequelize model, creating 297, 298, 301
 Twitter Bootstrap, adding 234
 Twitter Bootstrap, adding to 236
 Twitter Bootstrap, using 232
 Twitter login support, implementing 360
 used, for testing HTTPS 685, 686
Notes Docker compose file
 Docker stack file, creating from 571, 572, 573
Notes Docker images
 ECR repositories, setting up 561, 563
Notes model
 Chai tool 613
Mocha tool 612
test case 615
 test case, creating 614, 615
 test case, executing 616, 617
 test failures, diagnosing 623, 624, 625
 test suite 613
 testing 612
 testing, against MongoDB 626, 627
 testing, against MySQL 626, 627
 tests, adding 618, 619, 620, 621, 622
Notes router
 messages, adding to 410
Notes service
 server, provisioning 441, 442, 443
Notes stack file
 deploying, to Docker Swarm 594
Notes stack
 deploying, to Docker Swarm 590, 592, 594, 595, 596
 deploying, with Docker Compose file 498, 500, 501
Notes
 about 452
 adding, to test login functionality ability 655, 656, 658, 659
 adding, to test logout functionality ability 655, 656, 658, 659
 Cronginx container, adding to HTTPS 672, 674
 Let's Encrypt, using in Docker to implement HTTPS 672
 login functionality, testing 652, 653, 654, 655
 logout functionality, testing 652, 653, 654, 655
 NGINX, using in Docker to implement HTTPS 672
notes
 storing, in MongoDB 303
 storing, in SQL with SQLite3 285
 storing, with LevelDB datastore 279, 280, 282, 283
Notes
 testing, against MongoDB 635
NotesStore classes
 refactoring, to emit events 389, 391, 392

npm package
format 108, 109
global module installation, avoiding 117
installing 113
installing, by version number 114, 115
installing, from npm repository 115
installing, globally 116
publishing 128
searching 111
searching, with `package.json` fields 112, 113

npm-path module
URL 124

npm-run-all tool
reference link 613

npm
command-line programs, installation 121
dependencies, maintaining 117, 118
Node.js version compatibility, declaring 127
outdated packages, updating 125
package dependency version numbers,
specifying 120, 121
tasks, automating with scripts in `package.json`
126, 127
using 60, 61, 108

npx
reference link 62
used, for executing Node.js packaged
binaries 62

nvm
installing, on Windows 47, 48
used, for installing Node.js instances 45, 47

NXING
reference link 675

O

Object-Relational Mapping (ORM)
about 293, 314
notes, storing with Sequelize 293
operating system (OS) 427

P

package dependencies
maintaining, with npm 117, 118
updating, via fixing bugs 119, 120

`package.json` dependencies
automatically, updating 119

PaperCSS
URL 237

password encryption
adding, to user information service 375, 376,
377, 378

PATH variable
configuring, on Windows 123
configuring, to handle locally installed
commands 122, 123
modifications, avoiding 124, 125

pathnames
reviewing 101, 102

PayPal's blog post
reference link 14

persistent background processes
PM2 setup, integrating as 454, 456

Picnic CSS
URL 237

PM2 setup
for managing Node.js processes 448
integrating, as persistent background
processes 454, 456
scripting, on Multipass 450, 451, 453, 454

PM2
reference link 448
using 448, 450

Popper.js package
reference link 233

Popper.js
reference link 233

positive test 641

POSIX-like systems
Node.js, installing from source 40, 41
Node.js, installing from source for all 43, 44

PowerShell package
reference link 562

predefined script names
reference link 126

Privacy Enhanced Mail (PEM) 553

`process.argv` array
reference link 324

`process.stderr` streams
capturing 269

`process.stdout` streams

capturing 269
project directory structure
example 103, 105
Promise chain 191
Promise objects
fulfilled state 190
pending state 190
rejected state 190
Promise paradigm
error-first-callback, converting to 58
Promises
exploring, in Express router functions 188, 189, 190, 191
publishing packages
reference link 128
Puppeteer-based testing
project directory, setting up 647, 648
Puppeteer
reference link 646
test, creating for Notes application stack 648, 649, 650, 651
test, executing 651, 652
used, for implementing negative test 659
used, for testing frontend headless browser 646
Pure.css
URL 237
PuTTY Private Key (PPK) 553
pyramid of doom problem 189
Python, for Windows
reference link 49

R

read-eval-print loop (REPL) 11
real-time updates, on Notes home page
about 389
debug tracing, enabling in Socket.IO code 399
executing 398, 399
execution, while viewing note 404
home page, modifying 394, 395, 396
layout template, modifying 394, 395, 396
managing 392, 393, 394
note view template, modifying 402
notes, viewing 400

NotesStore classes, refactoring to emit events 389, 391
Socket.IO client, adding 396, 397
Redis server
Express/Passport session data, storing 509, 511, 512
Redis
Socket.IO messages, distributing 512, 514
used, for scaling Notes application stack 507
relative module identifiers 101
remote control access
setting up, to Docker Swarm hosted on EC2 instance 558, 559, 560, 561
Reporter
about 644
reference link 644
Representational State Transfer (REST)
modules and frameworks 184
request handlers
contrasting, from middleware functions 159, 160
request routing 140
require features
used, for finding Node.js module 92
used, for loading Node.js module 92
require function
used, for loading Node.js module 105, 106, 107
responsive 230
responsive web design techniques 230
REST backend service
calling, from Express application 176, 177
testing 638, 639, 640, 642, 643, 644
REST server
creating, for user information 319, 322
implementing, with Express application 177, 178
REST service
Fibonacci application, refactoring 181, 182, 183, 184
Restify
reference link 184, 314
REX-Ray project
URL 581
route parameter 159

S

script, with Node.js
 executing 54, 55
 inline async arrow functions, writing 56, 57

secrets and passwords
 securing 374, 375

Secure Shell (SSH) 527

Secure Sockets Layers (SSL) 667

semantic versioning (SemVer) 321

Sequelize
 configuring 294, 296, 297
 database, connecting 294, 296, 297
 model, creating for Notes application 297, 298, 300
 reference link 293
 used, for executing Notes application 301, 302

server, with Node.js
 launching 59

server-side JavaScript 13, 14

session store 373

Session Store implementation
 reference link 345

Shoelace
 URL 237

single EC2 instance
 single-node Docker Swarm, deploying on 547

single-node Docker Swarm
 deploying, on single EC2 instance 547

Socket.IO code
 debug tracing, enabling 399

Socket.IO documentation
 reference link 513

Socket.IO messages
 distributing, with Redis 512, 514

Socket.IO
 about 384, 385
 initializing, with Express 386, 387, 388
 URL 385

SQL injection attacks
 denying 698

SQL
 notes, storing with SQLite3 285

SQLite3

database schema 285, 286, 287

model code 287, 289, 290, 291

reference link 285

used, for executing Notes application 291, 292

used, for storing notes in SQL 285

SSL certificates
 reference link 685

static file web server
 reference link 156

Strategy modules 339

Strict Transport Security
 used, for improving HTTPS 694

system facility 270

T

TC-39 committee
 URL 63

Technical Steering Committee (TSC) 73

Terraform
 used, for configuring AWS Virtual Private Cluster 534, 535, 536, 537
 used, for creating AWS infrastructure 531, 532, 533, 534
 used, for deploying infrastructure to AWS 540, 542, 543, 545

Test Anything Protocol (TAP) 645

test infrastructure
 deploying, with Docker Swarm 628, 629, 630, 631
 managing, with Docker Swarm 627

testability
 improving, in Notes UI 662

testing methodologies
 assert module 610, 611

top-level module identifiers 101

Transmission Control Protocol (TCP) 558

transpiler 66

Transport Layer Security (TLS) 546

Twenty Twelve
 reference link 230

Twitter application, registering
 reference link 361

Twitter application
 updating 669

- Twitter authentication
adjusting, to work on server 446, 447
- Twitter Bootstrap
adding, to Notes application 234, 236
alternative layout frameworks 237
reference link 233
setting up 232, 234
using, on Notes application 232
- Twitter brand assets
reference link 370
- Twitter credentials
used, for diagnosing inability to log in 603, 604
- Twitter login support, for Notes application
application, registering 361, 362
implementing 360
TwitterStrategy, implementing 365, 367, 368, 370, 372, 373, 374
- Twitter sign-up process
information, implementing 362
- TwitterStrategy
implementing 365, 367, 370, 371, 373, 374
- type guard 272
- TypeScript 31
- ## U
- uncaught errors
capturing 262
- uncaught exceptions
capturing 270, 271
- unhandled rejected Promises
capturing 270, 271
- Uniform Resource Locator (URL) 321, 525
- universally unique identifier (UUID) 374
- user authentication service, setting up in Docker
about 464, 465
architecture, defining 469, 471
Authnet, exploring 485, 486, 487
Docker containers 468, 469
dockerizing 479
MySQL container, creating 472, 473, 474, 475
MySQL container, launching 465, 466, 467, 468
- user authentication service
- Docker container, building 482, 483, 484
Docker container, executing 482, 483, 484
Dockerfile, creating 480, 481
dockerizing 479
server, provisioning 433, 434, 435, 436, 437
- user information database
user, creating 327, 329, 331
- user information microservice
creating 313, 315
- user information model
developing 315, 318
- user information service
password encryption, adding 375, 376, 377, 378
user data, reading 331, 334
user information, updating 334
user record, deleting 336
user's password, checking 337, 339
- user profile
reference link 317
- ## V
- V8 16
- Virtual Private Cloud (VPC) 708
- Visual Studio build tools
reference link 49
- ## W
- web application frameworks 146, 147
- Web Hypertext Application Technology Working Group (WHATWG) 341
- Windows cmd.exe command line
environment variables, setting 151, 152
- Windows Subsystem for Linux (WSL)
about 38, 558
Node.js, installing 38
reference link 38
- Windows
administrator-privileged PowerShell, opening on 39
Node.js, installing from package management systems 37
Node.js, installing from source 45
nvm, installing on 47, 48
PATH variable, configuring on 123

script execution in PowerShell 440, 441
World Wide Web (WWW) 176

X

X-DNS-Prefetch-Control header
 setting up, with Helmet 691
X-Frame-Options header
 setting up, with Helmet 693
X-Powered-By header

removing, with Helmet 693

Y

YAML Ain't Markup Language (YAML)
 about 317
 reference link 293
Yarn package management system 128, 129
Yarn
 URL 61, 129