

41

42



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Node.js Web Development

Third Edition

Create real-time server-side applications with this practical,
step-by-step guide

David Herron

[PACKT] open source*
PUBLISHING
community experience distilled

Node.js Web Development

Third Edition

Create real-time server-side applications with this practical, step-by-step guide

David Herron



open source 
community experience distilled

BIRMINGHAM - MUMBAI

Node.js Web Development

Third Edition

Copyright © 2016 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: August 2011

Second published: July 2013

Third edition: June 2016

Production reference: 1220616

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78588-150-3

www.packtpub.com

Credits

Author

David Herron

Project Coordinator

Sanchita Mandal

Reviewer

Nicholas Duffy

Proofreader

Safis Editing

Commissioning Editor

Amarabha Banerjee

Indexer

Mariammal Chettiar

Acquisition Editor

Larissa Pinto

Graphics

Disha Haria

Content Development Editor

Samantha Gonsalves

Production Coordinator

Manu Joseph

Technical Editor

Vivek Pala

Cover Work

Manu Joseph

Copy Editor

Pranjali Chury

About the Author

David Herron has worked as a software engineer in Silicon Valley for over 20 years. This includes companies both tiny and large. He has worked on a wide variety of projects, from an X.400 e-mail server and client application to assisting with the launch of the OpenJDK project (open source Java rocks), to Yahoo's Node.js application-hosting platform (Mojito and Manhattan), and applications to monitor solar power array performance.

While a staff engineer at Sun Microsystems, David worked as the architect of the Java SE Quality Engineering team where he focused on test automation tools, including co-developing the AWT Robot class. He was involved in open source activities related to Java, including the OpenJDK project.

Before Sun, he worked for VXtreme on software which eventually became Windows Media Player when Microsoft bought the company. At Mainsoft, David worked on a library that allowed developers to recompile Windows applications on Unix, and then participated in porting Internet Explorer to Unix. At The Wollongong Group, he worked on both e-mail client and server software and was part of several IETF working groups improving e-mail-related protocols.

David is interested in electric vehicles, world energy supplies, climate change, and environmental issues, and he is a co-founder of Transition Silicon Valley. As an online journalist, he writes about electric cars and other green technology on LongTailPipe.com after having written for PlugInCars.com. He runs a large electric vehicle discussion website on VisForVoltage.org, and he blogs about other topics, including Node.js, Drupal, and Doctor Who on DavidHerron.com. Using Node.js, he developed a Content Management System that produces static HTML websites or EPUB3 eBooks, AkashaCMS ([akashacms . com](http://akashacms.com)).

There are many people I am grateful to.

I wish to thank my mother, Evelyn, for, well everything; my father, Jim; my sister, Patti; and my brother, Ken. What would life be without all of you?

I wish to thank my girlfriend, Maggie, for being there and encouraging me, her belief in me, her wisdom, and humor. May we have many more years of this.

I wish to thank Dr. Ken Kubota of the University of Kentucky for believing in me and giving me my first job in computing. It was six years of learning not just the art of computer system maintenance, but so much more.

I wish to thank my former employers, University of Kentucky Mathematical Sciences Department, The Wollongong Group, MainSoft, VXtreme, Sun Microsystems, Yahoo!, Recargo, Laplace Systems, and all the people I worked with in each company.

I am grateful to Ryan Dahl, the creator of Node.js, and the current Node.js core team members. They have the rare combination of wisdom and vision needed to create such a joy-filled fluid software development platform. Some platforms are just plain hard to work with, but not this one.

About the Reviewer

Nicholas Duffy has had a wide-ranging career, holding positions from analyst to business intelligence architect, to software engineer, and even golf professional. He has a passion for all things data and software engineering, specializing in data warehouse architecture, Python, and Node.js. He is a frequent contributor to open source projects and is, unfortunately, also a lifelong New York Mets fan.

You can read more about Nicholas' interests on this blog at
<https://medium.com/@duffn> or contact him via GitHub at @duffn.

I'd like to thank my wife, Anne, and boys, Jack and Chuck, for their never ending support in whatever endeavor.

www.PacktPub.com

eBooks, discount offers, and more

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.PacktPub.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.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface	vii
Chapter 1: About Node.js	1
The capabilities of Node.js	3
Server-side JavaScript	3
Why should you use Node.js?	4
Popularity	4
JavaScript at all levels of the stack	4
Leveraging Google's investment in V8	5
Leaner asynchronous event-driven model	5
Microservice architecture	5
The Node.js is stronger for having survived a major schism and hostile fork	5
Performance and utilization	8
Is Node.js a cancerous scalability disaster?	9
Server utilization, the bottom line, and green web hosting	11
Node.js, the microservice architecture, and easily testable systems	12
Node.js and the Twelve-Factor app model	13
Summary	13
Chapter 2: Setting up Node.js	15
System requirements	15
Installing Node.js using package managers	16
Installing on Mac OS X with MacPorts	16
Installing on Mac OS X with Homebrew	17
Installing on Linux, *BSD, or Windows from package management systems	17
Installing the Node.js distribution from nodejs.org	18

Installing from source on POSIX-like systems	19
Installing prerequisites	19
Native code modules and node-gyp	20
Installing developer tools on Mac OS X	21
Installing from source for all POSIX-like systems	22
Installing development instances with nvm	23
Node.js versions policy and what to use	25
Running and testing commands	26
Node.js's command-line tools	26
Running a simple script with Node.js	28
Launching a server with Node.js	29
NPM – the Node.js package manager	30
Node.js and ECMAScript 6 (ES-2015, ES-2016, and so on)	31
Using Babel to use experimental JavaScript features	33
Summary	35
Chapter 3: Node.js Modules	37
Defining a module	37
Node.js module format	39
File modules	39
Demonstrating module-level encapsulation	40
Directories as modules	41
Node.js's algorithm for require (module)	42
Module identifiers and path names	42
An example application directory structure	44
npm – the Node.js package management system	45
The npm package format	45
Finding npm packages	47
Other npm commands	49
Installing an npm package	49
Initializing a new npm package	50
Maintaining package dependencies with npm	51
Fixing bugs by updating package dependencies	52
Declaring Node.js version compatibility	53
Updating outdated packages you've installed	53
Installing packages from outside the npm repository	54
Publishing an npm package	54
Package version numbers	55
A quick note about CommonJS	57
Summary	57

Chapter 4: HTTP Servers and Clients – A Web Application's First Steps	59
Sending and receiving events with EventEmitters	60
The EventEmitter theory	61
HTTP server applications	62
ES-2015 multiline and template strings	65
HTTP Sniffer – listening to the HTTP conversation	67
Web application frameworks	69
Getting started with Express	70
Walking through the default Express application	74
The Express middleware	76
Middleware and request paths	77
Error handling	79
Calculating the Fibonacci sequence with an Express application	79
Computationally intensive code and the Node.js event loop	84
Algorithmic refactoring	86
Making HTTP Client requests	88
Calling a REST backend service from an Express application	91
Implementing a simple REST server with Express	91
Refactoring the Fibonacci application for REST	95
Some RESTful modules and frameworks	97
Summary	97
Chapter 5: Your First Express Application	99
ES-2015 Promises and Express router functions	99
Promises and error handling	101
Flattening our asynchronous code	102
Additional tools	103
Express and the MVC paradigm	104
Creating the Notes application	104
Your first Notes model	106
The Notes home page	108
Adding a new note – create	112
Viewing notes – read	116
Editing an existing note – update	117
Deleting notes – destroy	119
Theming your Express application	121
Scaling up – running multiple Notes instances	123
Summary	125

Chapter 6: Implementing the Mobile-First Paradigm	127
Problem – the Notes app isn't mobile friendly	128
Mobile-first paradigm	129
Using Twitter Bootstrap on the Notes application	131
Setting it up	132
Adding Bootstrap to application templates	134
Mobile-first design for the Notes application	136
Laying the Bootstrap grid foundation	136
Improving the notes list on the front page	140
Breadcrumbs for the page header	141
Cleaning up the add/edit note form	144
Building a customized Bootstrap	146
Bootstrap customizers	148
Summary	149
Chapter 7: Data Storage and Retrieval	151
Data storage and asynchronous code	152
Logging	152
Request logging with Morgan	153
Debugging messages	155
Capturing stdout and stderr	155
Uncaught exceptions	156
Storing notes in the filesystem	157
Storing notes with the LevelUP data store	163
Storing notes in SQL with SQLite3	167
SQLite3 database scheme	167
SQLite3 model code	169
Running Notes with SQLite3	173
Storing notes the ORM way with Sequelize	174
Sequelize model for the Notes application	175
Configuring a Sequelize database connection	179
Running the Notes application with Sequelize	180
Storing notes in MongoDB	182
MongoDB model for the Notes application	183
Running the Notes application with MongoDB	187
Summary	188
Chapter 8: Multiuser Authentication the Microservice Way	189
Creating a user information microservice	190
User information model	192
A REST server for user information	197
Scripts to test and administer the User Authentication server	204

Login support for the Notes application	207
Accessing the user authentication REST API	207
Login and logout routing functions	211
Login/logout changes to app.js	214
Login/logout changes in routes/index.js	216
Login/logout changes required in routes/notes.js	216
View template changes supporting login/logout	218
Running the Notes application with user authentication	221
Twitter login support for the Notes application	224
Registering an application with Twitter	224
Implementing TwitterStrategy	225
The Notes application stack	230
Summary	231
Chapter 9: Dynamic Interaction between Client and Server with Socket.IO	233
Introducing Socket.IO	235
Initializing Socket.IO with Express	235
Real time updates on the Notes home page	239
The Notes model as an EventEmitter class	239
Real-time changes in the Notes home page	242
Changing the home page template	244
Running Notes with real-time home page updates	245
Real-time action while viewing notes	245
Changing the note view template for real-time action	247
Running Notes with real-time updates while viewing a note	249
Inter-user chat and commenting for Notes	249
Data model for storing messages	249
Adding messages to the Notes router	252
Changing the note view template for messages	254
Using a Modal window to compose messages	254
Sending, displaying, and deleting messages	256
Running Notes and passing messages	260
Other applications of Modal windows	261
Summary	262
Chapter 10: Deploying Node.js Applications	263
Notes application architecture	264
Traditional Linux Node.js service deployment	265
Prerequisite – provisioning the databases	266
Installing Node.js on Ubuntu	268
Setting up Notes and User Authentication on the server	268
Setting up PM2 to manage Node.js processes	272
Twitter support for the hosted Notes app	275

Node.js microservice deployment with Docker	276
Installing Docker on your laptop	278
Starting Docker using Docker Toolbox and Docker Machine	279
Starting Docker with Docker for Windows/Mac	280
Kicking the tires of Docker	281
Creating the AuthNet for the User Authentication service	282
MySQL for the Authentication service	282
Dockerizing the Authentication service	286
Putting Authnet together	288
Creating FrontNet for the Notes application	291
MySQL for the Notes application	291
Dockerizing the Notes application	292
Putting FrontNet together	294
Configuring remote access on Docker for Windows or Mac	296
Configuring remote access in VirtualBox on Docker toolbox	297
Exploring the Docker Toolbox VirtualBoxMachine	298
Controlling the location of MySQL data volumes	299
Deploying to the cloud with Docker compose	302
Docker compose files	302
Running the Notes application with Docker Compose	306
Deploying to cloud hosting with Docker Compose	307
Summary	312
Chapter 11: Unit Testing	313
Testing asynchronous code	313
Assert – the simplest testing methodology	315
Testing a model	316
Mocha and Chai the chosen test tools	316
Notes model test suite	317
Configuring and running tests	320
More tests for the Notes model	321
Testing database models	323
Using Docker to manage test database servers	327
Docker Compose to orchestrate test infrastructure	327
Package.json scripts for Dockerized test infrastructure	331
Executing tests under Docker Compose	332
Testing REST backend services	334
Frontend headless browser testing with CasperJS	337
Setup	338
Improving testability in Notes UI	339
CasperJS test script for Notes	340
Running the UI test with CasperJS	343
Summary	344
Index	347

Preface

Welcome to the world of software development on the Node.js platform. This is an up-and-coming software platform that liberates JavaScript from the web browser, allowing us to reuse our JavaScript skills for general software development on a large range of systems. It runs atop the ultra-fast JavaScript engine at the heart of Google's Chrome browser, V8, and adds a fast and robust library of asynchronous network I/O modules. The Node.js community have developed a dizzyingly large body of third-party modules for nearly every conceivable purpose. While the primary focus of Node.js is high performance highly-scalable web applications, it is seeing widespread use in Internet of Things (IoT) applications, microservice development, asset build workflow for frontend engineers, and even in desktop applications like the Atom editor.

Microservices are one of the brightest ideas in computing today, and Node.js is right there as one of the best platforms for microservice development. This is doubly true when combined with Docker.

In just a few years, Node.js has gone from being a brand new "will anyone adopt it" technology to a major force in software development. It is now widely used in companies big and small, and the MEAN Stack (MongoDB, Express, AngularJS, and Node.js) has become a leading application model.

The Node.js platform was developed by Ryan Dahl in 2009 after a couple years of experimenting with web server component development in Ruby and other languages. His goal was to create an event-oriented system with a low-overhead architecture. This led Dahl toward an asynchronous single-thread system, as opposed to a more traditional thread-based architecture.

This model was chosen for simplicity, under the theory that threaded systems are notoriously difficult to develop and debug for lower overhead and for speed. Node.js's goal is to provide "an easy way to build scalable network servers." The design is similar to, and influenced by, other systems, such as Event Machine (Ruby) and the Twisted framework (Python).

JavaScript was chosen as the language because anonymous functions and other language elements provide an excellent base to implement asynchronous computation. Event handler functions are often coded in-line as anonymous functions. The Node.js runtime is ingeniously designed to support asynchronous I/O operations.

Now that ECMA Script 2016 is on the scene, new features, such as arrow functions and Promises, are coming to JavaScript engines, including Node.js. These powerful new capabilities will change JavaScript programming for the better. We now have a path toward taming some of the difficulties with asynchronous coding. The Promise object gives us a way to organize asynchronously-executed procedures in a semi-linear sequence, rather than the pyramid-shaped structures of callback functions.

The result is a platform that allows developers to not only succinctly write code of great power but to have a lot of fun while doing so.

Having JavaScript on both the server and the client (browser) lets us implement a vision dating back to the earliest days of the World Wide Web. Java's proponents first showed us dynamic stuff, powered by Java, running inside a web page. With Java on both client and server side, developers were supposed to reach nirvana. Java did not achieve success promoted by Sun Microsystems. Instead, it is JavaScript that is quickly advancing to implement that vision in a big way.

With Node.js, we have JavaScript on both the client and the server. While this probably won't help developers reach nirvana, our work is made much easier. We can use common code, common data structures, and more team members speak the same code language.

This book, *Node.js Web Development, Third Edition*, focuses on building web applications using Node.js. We assume that you have some knowledge of JavaScript and maybe even have server-side development experience. We will take a tour through the important concepts in order to understand Node.js programming.

To do so, we'll develop several applications, including a Note-taking application that will take several chapters to implement data storage with several database engines, user authentication, including OAuth2 against Twitter, real-time communications between users, and server deployment, including Docker. Along the way, we'll explore leading application development best practices, distributing heavy workloads to backend servers, and implementing REST microservices for multitiered architectures.

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, and why JavaScript should be liberated from the browser.

Chapter 2, Setting up Node.js, goes over setting up a Node.js developer environment. This includes installing Node.js on Windows, Mac OS X, and Linux, the command-line tools using ECMA Script 2015 features in Node.js, and the npm package management system.

Chapter 3, Node.js Modules, explores the module as the unit of modularity of Node.js applications. We dive deep into understanding and developing Node.js modules and using npm to maintain our dependency list.

Chapter 4, HTTP Servers and Clients – A Web Applications First Steps, starts exploring web development with Node.js. We develop several small webserver and client applications in Node.js. We use the Fibonacci algorithm to explore the effects of heavy-weight long-running computations on a Node.js application, as well as several mitigation strategies. This gives us our first exposure to REST-based service development.

Chapter 5, Your First Express Application, begins several chapters of developing a note-taking application. The first step is getting a basic application running.

Chapter 6, Implementing the Mobile-First Paradigm, uses the Bootstrap framework to implement responsive web design. Supporting mobile devices from the outset is a driving paradigm in modern software development.

Chapter 7, Data Storage and Retrieval, ensures that we don't lose our notes when restarting the application. We explore several database engines, and a method to make it easy to switch between them at will.

Chapter 8, Multiuser Authentication the Microservice Way, adds user authentication to the note-taking application. It can be used by both logged in and anonymous users with different capabilities for each. Authentication is supported against both a local user registry and using OAuth2 against Twitter.

Chapter 9, Dynamic Interaction between Client and Server with Socket.IO, lets our users talk to each other in real time. JavaScript code will be written in both browser and server, with Socket.IO providing the plumbing needed for real-time event exchange. Users will see notes change as they're edited by other users, or they will read and write notes that others can see.

Chapter 10, Deploying Node.js Applications, helps us understand Node.js application deployment. We look at both traditional Linux service deployment using an `/etc/init` script and using Docker to easily deploy an infrastructure of two databases and two Node.js services.

Chapter 11, Unit Testing, takes a look at three test development models: Unit Testing, REST testing, and functional testing. In addition to the Mocha and Chai frameworks, we use CasperJS to run automated tests in a headless browser component. Docker is used to facilitate test infrastructure deployment.

What you need for this book

The basic requirement is to install Node.js and to have a programmer-oriented text editor. We will show you how to install everything that you need, all of which is open source software, easily downloaded from the web. The most important tool is the one between your ears.

The examples here were tested using Node.js v5.x and ECMA Script 2015 features are widely used.

Some chapters require the database engines, MySQL and MongoDB.

While Node.js supports cross-platform software development, some of the third-party modules require compilation from source code. This means that one must have C/C++ compiler tools and Python installed. The details are covered in *Chapter 2, Setting up Node.js*.

While this book is about developing web applications, it does not require you to have a web server. Node.js provides its own web server stack.

Who this book is for

This book is written for any software engineer who wants the adventure that comes with a new software platform embodying a new programming paradigm.

Server-side engineers may find the concepts behind Node.js refreshing, giving you a different perspective on web application development. JavaScript is a powerful language, and Node.js's asynchronous nature plays to its strengths.

Developers experienced with JavaScript in the browser may find it fun to bring that knowledge to new territory.

We assume that you already know how to write software and have an understanding of modern programming languages such as JavaScript.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The implementation of this is in `views/pageHeader.ejs`."

A block of code is set as follows:

```
router.get('/auth/twitter/callback',
  passport.authenticate('twitter', { successRedirect: '/',
    failureRedirect: '/users/login' }));
```

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

```
<a class="btn btn-primary" href="/users/login">Log in</a>
<a class="btn btn-primary" href="/users/auth/twitter">Log in with Twitter</a>
<% } %>
```

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

```
$ npm start
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "You now have both **Log Out** and **ADD Note** buttons."



Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

About 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, asynchronous programming, built around JavaScript anonymous functions, and a single execution thread event-driven architecture.

While only a few years old, Node.js has quickly grown in prominence to where it's playing a significant role. Companies, small and large, 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 model is very different from common application server platforms using threads. The claim is that with the single-thread event-driven architecture, 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 are seeing it as a compelling alternative to the traditional Java, PHP, Python, Ruby on Rails, and so on, approach to building web applications.

At its heart, it is a standalone JavaScript engine with extensions making it suitable for general purpose programming and with 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. There are web application frameworks and application servers written in Node.js, however. In the few years that Node.js has been available, it's quickly gained a significant role, fulfilling the prediction that it could potentially supplant other web application stacks.

It is implemented around a non-blocking I/O event loop and a layer of file and network I/O libraries, all built on top of the V8 JavaScript engine (from the Chrome web browser). At the time of writing this, Microsoft had just proposed a patch to allow Node.js to utilize the ChakraCore JavaScript engine (from the Edge web browser). The theoretical possibility of hosting the Node.js API on top of a different JavaScript engine may come true, in the due course of time. Visit <https://github.com/nodejs/node-chakracore> to take a look at the project.

The I/O library is general enough to implement any sort of server implementing any TCP or UDP protocol, whether it's DNS, HTTP, IRC, or FTP. While it supports developing servers or clients for any network protocol, its biggest use case is in regular websites 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 be easily done with the Socket.IO library for Node.js.

A particularly intriguing combination is deploying small services using Docker into cloud hosting infrastructure. A large application can be divided into what's now called microservices and easily deployed at scale using Docker. The result fits agile project management methods since each microservice can be easily managed by a small team which collaborates at the boundary of their individual API.

This book will give you an introduction to Node.js. We presume the following:

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

We will cover the following topics in this chapter:

- An introduction to 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

We will dive right into developing working applications and recognize that often the best way to learn is by rummaging around in working code.

The capabilities of Node.js

Node.js is a platform for writing JavaScript applications outside web browsers. This is not the JavaScript we are familiar with in web browsers! For example, there is no DOM built into Node.js, nor any other browser capability.

Beyond its native ability to execute JavaScript, the bundled modules provide capabilities of this sort:

- Command-line tools (in shell script style)
- An interactive-TTY style of program (REPL which stands for Read-Eval-Print Loop)
- 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 and HTTPS 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.

In other words, it's very easy to write an HTTP server in Node.js, but the typical web application developer doesn't need to work at that level of detail. For example, PHP coders assume that Apache is already there, and that they don't have to implement the HTTP server portion of the stack. The Node.js community has developed a wide range of web application frameworks such as Express, allowing developers to quickly configure an HTTP server that provides all of the basics we've come to expect—sessions, cookies, serving static files, logging, and so on—thus letting developers focus on their business logic.

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 browsers?".

Back in the dawn of the web age, the tools for writing web applications were at a fledgling stage. Some 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. In other words, JavaScript outside the browser is not a new thing, even if it is uncommon.

Why should you use Node.js?

Among the many available web application development platforms, why should you chose Node.js? There are many stacks to choose from; What is it about Node.js that makes it rise above the others? We will see in the following sections.

Popularity

Node.js is quickly becoming a popular development platform with adoption from plenty of big and small players. One of those is PayPal, who are replacing their incumbent Java-based system with one written in Node.js. For PayPal's blog post about this, visit <https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/>. Other large Node.js adopters include Walmart's online e-commerce platform, LinkedIn, and eBay.

Since we shouldn't just follow the crowd, let's look at technical reasons to adopt Node.js.

JavaScript at all levels of the stack

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 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 the hype and we ended up with JavaScript as the principle in-browser client-side language, rather than Java. With Node.js we may finally be 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 wins:

- The same programming staff can work on both ends of the wire
- Code can be migrated between server and client more easily

- Common data formats (JSON) exist between server and client
- Common software tools exist for server and client
- Common testing or quality reporting tools for server and client
- When writing web applications, view templates can be used on both sides

The JavaScript language is very popular due to its ubiquity in web browsers. It compares favorably against 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. The competition to make the best web browser leads Google to keep on improving V8. As a result, Node.js programmers automatically win as each V8 iteration ratchets up performance and capabilities.

The Node.js community may change things to utilize any JavaScript engine, in case another one ends up surpassing V8.

Leaner asynchronous event-driven model

We'll get into this later. The Node.js architecture, a single execution thread and a fast JavaScript engine, has less overhead than thread-based architectures.

Microservice architecture

A new hotness in software development is the microservice idea. Node.js is an excellent platform for implementing microservices. We'll get into this later.

The Node.js is stronger for having survived 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 who wanted to incorporate several features and change who's in control of making decisions. What resulted is a merge of the Node.js and *io.js* repositories, an independent Node.js foundation to run the show, and the community is working together to move forward in a common direction.

Threaded versus 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. That's a nice thing to say, but what's the rationale for the statement?

The normal application server model uses blocking I/O to retrieve data, and it uses threads for concurrency. Blocking I/O causes threads to wait, causing a churn between threads as they are forced to wait on I/O while the application server handles requests. Threads add complexity to the application server as well as server overhead.

Node.js has a single execution thread with no waiting on I/O or context switching. Instead, there is an event loop looking for events and dispatching them to handler functions. The paradigm is that any operation that would block or otherwise take time to complete must use the asynchronous model. These functions are to be given an anonymous function to act as a handler callback, or else (with the advent of ES2015 promises), the function would return a Promise. The handler function, or promise, is invoked when the operation is complete. In the meantime, control returns to the event loop, which continues dispatching events.

To help us wrap our heads around this, Ryan Dahl, the creator of Node.js, (in his *Cinco de Node* presentation) asked us what happens while executing a line of code like this:

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

Of course, the program pauses at that point while the database layer sends the query to the database, which determines the result and returns the data. Depending on the query, that pause can be quite long. Well, a few milliseconds, which is an eon in computer time. This pause is bad because while the entire thread is idling, another request might come in and need to be handled. This is where a thread-based server architecture would need to make a thread context switch. The more outstanding connections to the server, the greater the number of thread context switches. Context switching is not free because more threads requires more memory for per-thread state and more time for the CPU to spend on thread management overhead.

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

Using threads to implement concurrency often comes with admonitions like these: *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 the 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 the tendency to create frameworks such as `java.util.concurrent` to tame the complexity of threaded concurrency, but some might argue that papering over complexity does not make things simpler.

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, and simpler to implement.

Ryan Dahl points to the relative access time of objects to understand the need for asynchronous I/O. Objects in memory are more quickly accessed (on 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 zillions 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.

In Node.js, the query discussed previously will read as follows:

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

Or if written with an ES2015 Promise:

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

This code performs the same query written earlier. The difference is that the query result is not the result of the function call, but it is provided to a callback function that will be called later. The order of execution is not one line after another, but it is instead determined by the order of callback function execution.

Once the call to the `query` function finishes, control will return almost immediately to the event loop, which goes on to servicing other requests. One of those requests will be the response to the query, which invokes the callback function.

Commonly, web pages bring together data from dozens of sources. Each one has a query and response as discussed earlier. Using asynchronous queries, each one can happen in parallel, where the page construction function can fire off dozens of queries – no waiting, each with their own callback – and then go back to the event loop, invoking the callbacks as each is done. Because it's in parallel, the data can be collected much more quickly than if these queries were done synchronously one at a time. Now, the reader on the web browser is happier because the page loads more quickly.

Performance and utilization

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

One benchmark going around is this 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 one 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` Hello World response.

Ryan Dahl (Node.js's original author) showed a simple benchmark (http://nodejs.org/cinco_de_node.pdf) that returned a 1-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 4 megabytes memory, while Node.js peaked at 64 megabytes.

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%. In another presentation (<http://www.slideshare.net/FabianFrankDe/yahoo-scale-nodejs>), he discussed how Yahoo!Axis is running on Manhattan + Mojito and the value of being able to use the same language (JavaScript) and framework (YUI/YQL) on both frontend and backend.

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 let them move from 30 servers down to three, 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, choosing 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/>.

Mikito Takada blogged about benchmarking and performance improvements in a *48 hour hackathon* application (<http://blog.mixu.net/2011/01/17/performance-benchmarking-the-node-js-backend-of-our-48h-product-wehearvoices-net/>) he built comparing Node.js with what he claims is a similar application written with Django (a web application framework for Python). The unoptimized Node.js version is quite a bit slower (in response time) than the Django version but a few optimizations (MySQL connection pooling, caching, and so on) made drastic performance improvements handily beating out Django.

Is Node.js a cancerous scalability disaster?

In October 2011, software developer and blogger Ted Dziuba wrote an infamous blog post (since pulled from his blog) claiming that Node.js is a cancer, calling it a "scalability disaster". The example he showed for proof is a CPU-bound implementation of the Fibonacci sequence algorithm. While his argument was flawed, he raised a valid point that Node.js application developers have to consider – where do you put the heavy computational tasks?

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

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

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

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 mathematics functions:

```
var http = require('http');  
var url = require('url');  
  
var fibonacci = // as above  
  
http.createServer(function (req, res) {  
    var urlP = url.parse(req.url, true);  
    var fibo;  
    res.writeHead(200, {'Content-Type': 'text/plain'});  
    if (urlP.query['n']) {  
        fibo = fibonacci(urlP.query['n']);  
        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');
```

If you call this from the request handler in a Node.js HTTP server, for sufficiently large values of n (for example, 40), the server becomes completely unresponsive because the event loop is not running, as this function is grinding through the calculation.

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 a solution. The possible solutions include rewriting the algorithm to work with the event loop or to foist computationally expensive calculations to a backend server.

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

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

Dziuba's valid point wasn't expressed well in his blog post, and it was somewhat lost in the flames following that post. Namely, that while Node.js is a great platform for I/O-bound applications, it isn't a good platform for computationally intensive ones.

Server utilization, the bottom line, and green web hosting

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 can let your organization do more with less.

Roughly speaking, the more servers you buy, the greater the cost, and the greater the environmental impact. There's a whole field of expertise around reducing cost 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 lower environmental impact.

Intel's paper, *Increasing Data Center Efficiency with Server Power Measurements* (http://download.intelintel.com/it/pdf/Server_Power_Measurement_final.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 (datacenter efficiency, datacenter density, and storage density) can decrease costs and environmental impact. But you can destroy those 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 datacenter efficiency with an efficient software stack.

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.

Node.js, the microservice architecture, and easily testable systems

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 appserver). 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, such as for easier testing
- It's easier to switch technology stack choices

Where does Node.js fit 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

Node.js and the Twelve-Factor app model

Throughout this book, we'll call out aspects of the Twelve-Factor application model, and ways to implement those ideas in Node.js. This model is published on <http://12factor.net>, and it is a set of guidelines for application deployment in the modern cloud computing era.

The guidelines are straightforward, and once you read them, they seem like pure common sense. As a best practice, the Twelve-Factor model is a compelling strategy for delivering the kind of fluid self-contained cloud deployed applications called for by our current computing environment.

Summary

You learned a lot in this chapter. Specifically, you saw that JavaScript has a life outside web browsers and you learned about the difference between asynchronous and blocking I/O. We then covered the attributes of Node.js and where it fits in the overall web application platform market and threaded versus asynchronous software. Lastly, we saw the advantages of fast event-driven asynchronous I/O, coupled with a language with great support for anonymous closures.

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

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 setting up a Node.js environment, so let's get started.

2

Setting up Node.js

Before getting started with using Node.js, you must set up your development environment. In the following chapters, we'll use this for development and for non-production deployment.

In this chapter, we will cover the following topics:

- How to install Node.js from source and prepackaged binaries on Linux, Mac, or Windows
- How to install the NPM package manager and some popular tools
- The Node.js module system

So let's get on with it.

System requirements

Node.js runs on POSIX-like operating systems, various UNIX derivatives (Solaris, for example) or workalikes (Linux, Mac OS X, and so on), as well as on Microsoft Windows thanks to extensive assistance from Microsoft. It can run on machines both large and small, including the tiny ARM devices such as the Raspberry Pi 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 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. If you plan to use encryption in your networking code, you will also need the OpenSSL cryptographic library. The modern UNIX derivatives almost certainly come with these, and Node.js's configure script, used when installing from source, will detect their presence. If you need to install them, Python is available at <http://python.org> and OpenSSL is available at <http://openssl.org>.

Installing Node.js using package managers

The preferred method for installing Node.js, now, is to use the versions available in package managers, such as `apt-get`, or Macports. Package managers simplify your life 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 this first.

Installing on Mac OS X with MacPorts

The MacPorts project (<http://www.macports.org/>) has for years been packaging a long list of open source software packages for Mac OS X, and they have packaged Node.js. After you have installed MacPorts using the installer on their website, installing Node.js is pretty much this simple:

```
$ sudo port search nodejs npm
nodejs @4.4.3 (devel, net)
    Evented I/O for V8 JavaScript

nodejs-devel @5.10.0 (devel, net)
    Evented I/O for V8 JavaScript

Found 2 ports.

--
npm @2.15.2 (devel)
    node package manager

npm-devel @3.8.6 (devel)
    node package manager
$ sudo port install nodejs npm
.. long log of downloading and installing prerequisites and Node
$ which node
/opt/local/bin/node
$ node --version
v4.4.3
```

Installing on Mac OS X with Homebrew

Homebrew is another open source software package manager for Mac OS X, 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 Homebrew is correctly set up, use the following:

```
$ brew search node  
leafnode  node
```

Then, install it this way:

```
$ brew install node  
==> Downloading https://homebrew.bintray.com/bottles/node-5.10.1.el_ _  
capitan.bottle.tar.gz  
#####
100.0%  
==> Pouring node-5.10.1.el_capitan.bottle.tar.gz  
==> Caveats  
Please note by default only English locale support is provided. If you  
need  
full locale support you should:  
`brew reinstall node --with-full-icu`  
  
Bash completion has been installed to:  
/usr/local/etc/bash_completion.d  
==> Summary  
bin  /usr/local/Cellar/node/5.10.1: 3,663 files, 35.7M
```

Once installed this way, the Node.js command can be run as follows:

```
$ node --version  
v5.5.0
```

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

Node.js is now available through most of the 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, Mac OS X, and even Windows. Visit <https://nodejs.org/en/download/package-manager/> for more information.

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

```
# curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -
# sudo apt-get install -y nodejs
# sudo apt-get install -y build-essential
```

In earlier Node.js releases, a Launchpad PPA maintained by Chris Lea was used instead. That PPA has transitioned to become Nodesource. Visit the Node.js website for further documentation.

Installing the Node.js distribution from nodejs.org

The <https://nodejs.org/en/> website offers built-in binaries for Windows, Mac OS X, 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 preferable to use the package management system. That's because you'll find it easier to stay up-to-date with the latest version. Some will prefer to install a binary.

Simply go to <https://nodejs.org/en/> and you'll see something like the following screenshot. Click on the **DOWNLOADS** link in the header to look at all possible downloads:



For Mac OS X, the installer is a PKG file giving the typical installation process. For Windows, the installer simply takes you through the typical Install Wizard process.

Once finished with the installer, you have a command-line tool with which you can run Node.js programs.

Installing from source on POSIX-like systems

Installing the prepackaged Node.js distributions is currently the preferred installation method. However, installing Node.js from 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 the high-level view, let's get our hands dirty mucking around in some build scripts. The general process follows the usual `configure`, `make`, and `make install` routine that you may already have 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 the README contained within the source distribution at <https://github.com/nodejs/node/blob/master/README.md>.



Installing prerequisites

As noted in the preceding section, there are three prerequisites: a C compiler, Python, and the OpenSSL libraries. The Node.js installation process checks for their presence and will fail if the C compiler or Python is not present. The specific method of installing these is dependent on your operating system

These commands will check for their presence:

```
$ cc --version
Apple LLVM version 7.0.2 (clang-700.1.81)
Target: x86_64-apple-darwin15.3.0
Thread model: posix
```

```
$ python  
Python 2.7.11 (default, Jan  8 2016, 22:23:13)  
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>>
```

Native code modules and node-gyp

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, and build the requisite .node files.

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`.

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. You'll see in the output a `node-gyp` execution, followed by many lines of text obviously related to compiling C/C++ files.

The `node-gyp` tool is a cross-platform command-line tool written in Node.js for compiling native addon 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.

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

Its GitHub repository contains documentation at <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, as well as developing native code modules.

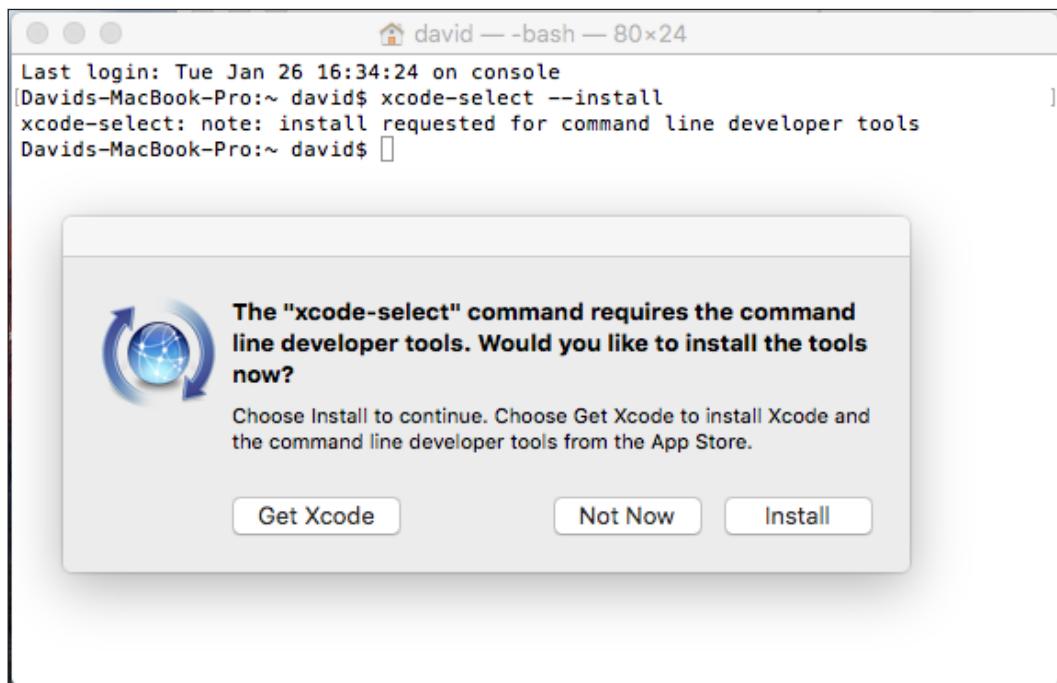
Installing developer tools on Mac OS X

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

You start with Xcode, which is available, for free through the Mac App Store. Simply search for Xcode and click on the Get button. Once you have Xcode installed, open a Command Prompt 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/>.

Installing from source for all POSIX-like systems

First, download the source from <http://nodejs.org/download>. One way to do this is with your browser and another way is as follows, substituting your preferred version:

```
$ mkdir src  
$ cd src  
$ wget https://nodejs.org/dist/v4.2.6/node-v4.2.6.tar.gz  
$ tar xvzf node-v4.2.6.tar.gz  
$ cd node-v4.2.6
```

The next step is to configure the source so that it can be built. It is done with the typical sort of configure script and you can see its long list of options by running the following:

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

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

```
$ ./configure --prefix=$HOME/node/4.2.6  
..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.

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, it will print a message about something that needs to be fixed. Once the configure script is satisfied, you can go 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 into a system-wide directory, do 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=$HOME/node/4.2.6/bin:${PATH}' >> ~/.bashrc  
$ . ~/.bashrc
```

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

```
$ echo 'setenv PATH $HOME/node/4.2.6/bin:${PATH}' >> ~/.cshrc  
$ source ~/.cshrc
```

This should result in some directories like this:

```
$ ls ~/node/4.2.6/  
bin      include      lib      share  
$ ls ~/node/4.2.6/bin
```

Installing development instances with nvm

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

If you noticed during the instructions discussed earlier, the `-prefix` option was used in a way that directly supports installing several Node.js versions side by side in the same directory:

```
$ ./configure -prefix=$HOME/node/4.2.7
```

And:

```
$ ./configure -prefix=/usr/local/node/4.2.7
```

This initial step determines the install directory. Clearly when version 4.2.7 or version 5.5.3 or whichever version is released, you can change the `install` prefix to have the new version installed side by side with previous versions.

To switch between Node.js versions is simply a matter of changing the PATH variable (on POSIX systems), as follows:

```
$ export PATH=/usr/local/node/4.2.7/bin:${PATH}
```

It starts to be a little tedious to maintain 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 install. Also, the command shown to change your PATH is not quite optimal. Inventive programmers have created several version managers to simplify setting up not only Node.js but also NPM and providing commands to change your PATH the smart way:

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

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

For example, with nvm, you can run commands like these:

```
$ nvm ls
  v0.10.40
  v0.12.7
  v4.0.0
  v4.1.1
  v4.1.2
  v4.2.0
  v5.1.0
  v5.3.0
  v5.5.0
->      system
node -> stable (-> v5.5.0) (default)
stable -> 5.5 (-> v5.5.0) (default)
$ nvm use v5
Now using node v5.5.0 (npm v3.3.12)
$ node --version
v5.5.0
$ nvm use v4.2
Now using node v4.2.0 (npm v2.14.7)
$ node --version
v4.2.0
$ nvm install v5.4
#####
100.0%
```

```
Checksums empty  
Now using node v5.4.1 (npm v3.3.12)  
$ node --version  
v5.4.1  
$ /usr/bin/node --version  
v0.10.40
```

This demonstrates that you can have a system-wide Node.js installed and keep multiple private Node.js versions and switch between them as needed. When new Node.js versions are released, they are simple to install with nvm even if the official packaged version for your OS doesn't immediately update.

Node.js versions policy and what to use

We just threw around so many different Node.js version numbers in the previous section that you may have become confused over which version to use. This book is targeting Node.js version 5.x, and it's expected that everything we'll cover is compatible with Node.js 5.x and any subsequent release.

Starting with Node.js 4.x, the Node.js team is following a dual-track approach. The even numbered releases (4.x, 6.x, and so on) are what they're calling **Long Term Support (LTS)**, while the odd numbered releases (5.x, 7.x, and so on) are where current new feature development occurs. While the development branch is kept stable, the LTS releases are more positioned as being more appropriate for production use.

At the time of writing this, Node.js 4.x is the current LTS release; Node.js 6.x was just released and will eventually become the LTS release. Refer to <https://github.com/nodejs/LTS/>.

It's likely that everything shown in this book will work on 4.x, but that's not guaranteed.

A major impact of each new Node.js release, beyond the usual performance improvements and bug fixes, is bringing in the latest V8 JavaScript engine release. In turn, this means that we can use more and more of the ES-2015 features as the V8 team progresses towards full ES-2015 and ES-2016 adoption.

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 breaking changes in 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?

The NPM Package Manager helps us ensure that our packages execute on the correct Node.js version. This means that we can specify in the package.json file, which we'll explore in *Chapter 3, Node.js Modules*, the compatible Node.js versions for a package.

We can add an entry to package.json like this:

```
engines: {  
  "node": ">=5.x"  
}
```

This means exactly what it implies—that the given package is compatible with Node.js version 5.x or later.

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 you with the command-line tools.

Node.js's command-line tools

The basic install 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 debug 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)  
  --no-deprecation      silence deprecation warnings
```

```
--trace-deprecation      show stack traces on deprecations
--throw-deprecation    throw an exception anytime a deprecated function
is used
--trace-sync-io        show stack trace when use of sync IO
                      is detected after the first tick
--track-heap-objects   track heap object allocations for heap snapshots
--prof-process          process v8 profiler output generated
                      using --prof
--v8-options            print v8 command line options
--tls-cipher-list=val  use an alternative default TLS cipher list
--icu-data-dir=dir     set ICU data load path to dir
                      (overrides NODE_ICU_DATA)
```

Environment variables:

NODE_PATH	' : '-' separated list of directories prefixed to the module search path.
NODE_DISABLE_COLORS	set to 1 to disable colors in the REPL
NODE_ICU_DATA	data path for ICU (Intl object) data
NODE_REPL_HISTORY	path to the persistent REPL history file

Documentation can be found at <https://nodejs.org/>

It prints the USAGE message giving you the command-line options.

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 them.

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 next section, *Running a simple script with Node.js*.

Running Node.js with no arguments plops 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-orientated 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 see how to run scripts with Node.js. It's quite simple and let's start by referring back to the help message shown above. It's just a script filename and some script arguments, which should be familiar for 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++, 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 then inside that creating one directory per chapter (for example, `chap02` and `chap03`).

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

```
var fs = require('fs');
var files = fs.readdirSync('.');
for (fn in files) {
  console.log(files[fn]);
}
```

Next, run it by typing the following command:

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

This is a pale cheap imitation of the Unix `ls` command (as if you couldn't figure that out from the name). The `readdirSync` function is a close analogue to the Unix `readdir` system call (type `man 3 readdir` in a terminal window to learn more) and is used to list the files in a directory.

The script arguments land in a global array named `process.argv`, and you can modify `ls.js`, copying it as `ls2.js`, as follows to see how this array works:

```
var fs = require('fs');
var dir = '.';
if (process.argv[2]) dir = process.argv[2];
var files = fs.readdirSync(dir);
```

```
for (fn in files) {  
    console.log(files[fn]);  
}
```

You can run it as follows:

```
$ node ls2 ~/.nvm/versions/node/v5.5.0/bin  
node  
npm
```

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 in the dual mode of verifying the installation and familiarizing you with using Node.js, we want to run a simple HTTP server. Let's borrow the simple serverscript on the Node.js home page (<http://nodejs.org>).

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

```
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');
```

Run it this way:

```
$ 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 – A Web Application's First Steps*; *Chapter 5, Your First Express Application*; and *Chapter 6, Implementing the Mobile-First Paradigm*. At the moment just visit `http://127.0.0.1:8124` in your browser to see the "**Hello, World!**" message.

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

The reason 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 event listener keeps `app.js` running until you do something like typing Control-C in the terminal window. In `ls.js`, there is nothing that creates a long-running event listener, so when `ls.js` reaches the end of its script, the node process will exit.

NPM – the Node.js package manager

Node.js by itself is a pretty basic system, being a JavaScript interpreter with a few interesting asynchronous I/O libraries. One of the things that make Node.js interesting is the rapidly growing ecosystem of third-party modules for Node.js. At the center of that ecosystem is NPM. The modules can be downloaded as source and assembled manually for use with Node.js programs. NPM gives us a simpler way; 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 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. **Hexy** is a utility program for printing hex dumps of files. It serves our purpose right now in giving us something to quickly install and try out:

```
$ npm install -g hexy
/home/david/.nvm/versions/node/v5.9.1/bin/hexy -> /home/david/.nvm/
versions/node/v5.9.1/lib/node_modules/hexy/bin/hexy_cmd.js
/home/david/.nvm/versions/node/v5.9.1/lib
--- hexy@0.2.7
```

Adding the `-g` flag makes the module available globally, irrespective of which directory it is installed in. It is most useful when the module provides a command-line interface because `npm` ensures that the command is installed correctly for use by all users of the computer.

Depending on how Node.js is installed for you, that 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: 7661 7220 6673 203d 2072 6571  var.fs.=.req
0000000c: 7569 7265 2827 6673 2729 3b0a  uire('fs');//.
00000018: 7661 7220 6669 6c65 7320 3d20  var.files.=.
00000024: 6673 2e72 6561 6464 6972 5379  fs.readdirSy
00000030: 6e63 2827 2e27 293b 0a66 6f72  nc('.');.for
0000003c: 2028 666e 2069 6e20 6669 6c65  .(fn.in.file
00000048: 7329 207b 0a20 2063 6f6e 736f  s).{...conso
```

```
00000054: 6c65 2e6c 6f67 2866 696c 6573  le.log(files
00000060: 5b66 6e5d 293b 0a7d 0a          [fn]);..}.
```

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.

Node.js and ECMAScript 6 (ES-2015, ES-2016, and so on)

In 2015, the ECMAScript committee released a major update of the JavaScript language. 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 into Node.js starting with version 4.x.



To learn about the current status of ES-2015 in Node.js, visit
<https://nodejs.org/en/docs/es6/>.

By default, only the ES-2015 features which V8 considers *stable* are enabled. Further features can be enabled with command-line options. The *almost complete* features are enabled with the `--es_staging` option. The features still in progress are enabled by the `--harmony_destructuring` option. The website documentation gives more information.

The ES-2015 features make a big improvement in the JavaScript language. One feature, the `Promise` class, should mean a fundamental rethinking of common idioms in Node.js programming. In ES-2016, a pair of new keywords, `async` and `await`, will simplify coding asynchronous code in Node.js, and it 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 two of them we'll use extensively.

The first is 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. The value of `this` is the same as it was outside this invocation. 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 asynchronous execution is to be performed
- The first argument to the callback function is an error indicator

While convenient, these conventions 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 ...;
        });
      }
    });
  }
});
```

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 above pyramid can be coded as follows:

```
doThis(arg1, arg2)
  .then((result1, result2) => {
    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.

At the time of writing, the Node.js community has barely begun to use this new paradigm, and instead it is still using the old *pass in a callback function* paradigm. In the interim, we'll have to be flexible enough to use both paradigms. Over the long term, there should be a switch to this paradigm.

Using Babel to use experimental JavaScript features

The Babel transpiler (<http://babeljs.io/>) is a great way to use cutting edge JavaScript features on older implementations. The word *transpile* means Babel rewrites JavaScript code into other JavaScript code, specifically to rewrite ES-2015 or ES-2016 features to older JavaScript code. Babel does this by converting JavaScript source to an abstract syntax tree, then manipulating that tree to rewrite the code using older JavaScript features, and then writing that tree to a JavaScript source code file.

Since Node.js doesn't yet support all the ES-2015 features, we can use Babel to go ahead and use those features.



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

In the root directory of your project, type these commands:

```
$ npm install --save-dev babel-cli  
$ npm install babel-preset-es2015 --save-dev
```

This installs the Babel software, saving references in the package.json file. We'll go over that file in more depth in the next chapter, but NPM uses package.json to document attributes of the module. Some of those attributes are dependencies on other modules, and in this case, we're documenting a development dependency on Babel.

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

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

Next, in the same directory as `package.json`, create a file named `.babelrc` containing the following:

```
{  
  "presets": ["es2015"]  
}
```

This instructs Babel to use `presets` to configure the set of transformations it will perform.

The last step is to create a directory named `src` to store code written with ES2015 or ES2016 features. The transpiled code will land in the directory named `lib` and it's that code which will be executed.

To transpile your code, run the following command:

```
$ ./node_modules/.bin/babel src -d lib
```

This command asks to take all files in the `src` directory and transpile each to a matching file name in the `lib` directory.

To automate the process, edit the `package.json` file and insert the following snippet:

```
"scripts": {  
  "build": "babel src -d lib"  
},
```

With this, you can use the following command:

```
$ npm run build
```

We'll go over `npm` scripts in more depth later, but this is a way to record certain commands for later use.

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 the 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 **Node Package Manager (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 ES-2015. Finally, we saw how to use Babel to implement those features in your code.

Now that we've seen how to set up the basic system, we're ready to start working on implementing applications with Node.js. First, you must learn the basic building block of Node.js applications, modules, which we will cover in the next chapter.

3

Node.js Modules

Before writing Node.js applications, you must learn about Node.js modules and packages. Modules and packages are the building blocks for breaking down your application into smaller pieces.

In this chapter, we will cover the following topics:

- Defining a module
- The CommonJS module specification
- Using ES-2015 modules in Node.js
- Understanding how Node.js finds modules
- The npm package management system

So, let's get on with it.

Defining a module

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

We have already seen modules in action in the previous chapter. Every JavaScript file we use in Node.js is itself a module. It's time to see what they are and how they work.

In the `1s.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:

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

The `require` function searches for modules, loading the module definition into the Node.js runtime, and makes its functions available. In this case, the `fs` object contains the code (and data) exported by the `fs` module.

This is true of every Node.js module, the `exports` object within the module is the interface provided to other code. Anything assigned to a field of the `exports` object is available to other pieces of code, and everything else is hidden.

Let's look at a brief example of this before diving into the details. Ponder over the `simple.js` module:

```
var count = 0;
exports.next = function() { return count++; }
```

We have one variable, `count`, which is not attached to the `exports` object and a function, `next`, which is attached. Now, let's use it.

```
[MacBook-Pro-2:chap03 david$ node --version
v5.9.1
[MacBook-Pro-2:chap03 david$ node
[> var s = require('./simple');
undefined
[> s.next();
0
[> s.next();
1
[> s.next();
2
[> s.next();
3
[> console.log(s.count);
undefined
undefined
[> MacBook-Pro-2:chap03 david$
MacBook-Pro-2:chap03 david$ ]
```

The `exports` object is what's 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.

To reiterate the rule:

- Anything (functions or objects) assigned as a field of `exports` is available to other code outside the module
- Objects not assigned to `exports` are not available

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

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

Node.js module format

Node.js's module implementation is strongly inspired by, but not identical to, the CommonJS module specification. The differences between them might only be important if you need to share code between Node and other CommonJS systems.

One change that came with ES2015 is a standard module format. It has some interesting features, but it is incompatible with the CommonJS/Node.js module system. Further, the Node.js runtime (as of versions 5.x and 6.x) does not support ES2015 modules.

Babel supports transpiling ES2015 modules to Node.js modules. This suggests a possible strategy for code to be shared between Node.js and browser applications. Namely, to write ES2015 modules, transpiling them as needed to either Node.js or straight to JavaScript for browsers that don't support ES2015 modules.

In the long run, the Node.js community will face a choice once compatibility with ES2015 modules comes about. It will be extremely important for the `require` function to support both CommonJS style Node.js modules and ES2015 modules. The specifics for doing so are under discussion.

In the meantime, we need to focus on the task at hand, which is to learn about Node.js modules.

File modules

The `simple.js` module we just saw is what the Node.js documentation describes as a **file module**. Such modules are contained within a single file, whose filename ends with either `.js` or `.node`. The latter are compiled from C or C++ source code, while the former are of course written in JavaScript.

Node.js modules provide a simple encapsulation mechanism to hide implementation details while exposing an API. Modules are treated as if they were written as follows:

```
(function() { ... contents of module file ... })();
```

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 which looks global is actually contained within this private context.

There are two free variables inserted by Node.js into this private context: `module` and `exports`. The `module` object contains several fields you might find useful. Refer to the online Node.js documentation for details.

The `exports` object is an alias of the `module.exports` field. This means that the following two lines of code are equivalent:

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

Your code can break the alias between the two if you do this:

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

If you want the module to return a single function, you must instead do this:

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

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

Demonstrating module-level encapsulation

That was a lot of words, so let's do a quick example. Create a file named `module1.js` containing this:

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

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

```
var util = require('util');  
var A = "a different value A";  
var B = "a different value B";  
var m1 = require('../module1');  
util.log('A=' + A + ' B=' + B + ' values=' + util.inspect(m1.values()));
```

Then, run it as follows (you must have Node.js already installed):

```
$ node module2.js  
19 May 21:36:30 - A=a different value A B=a different value B  
values={ 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`. Values encapsulated within a module can be exported, such as the `.values` function in `module1.js`.

Directories as modules

A module can contain a whole directory structure full of stuff. Such a module might contain several internal modules, data files, template files, documentation, tests, assets, and more. Once stored within a properly constructed directory structure, Node.js will treat these as a module that satisfies a `require('moduleName')` call.



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*.

One way to implement this is with a package `.json` file. As the name implies, it is in the JSON format, and it contains data about the module (package). The npm package management system puts a lot of data into the `package.json` file, but the Node.js runtime recognizes only these two values:

```
{ name: "myAwesomeLibrary",
  main: "./lib/awesome.js" }
```

If this `package.json` file is in a directory named `awesomelib`, then `require('./awesomelib')` will load the file module in `./awesomelib/lib/awesome.js`.

If there is no `package.json`, then Node.js will look for either `index.js` or `index.node`. In this case, `require('./awesomelib')` will load the file module in `./awesomelib/index.js`.

In either case, the directory module can easily contain other file modules. The module that's initially loaded would simply use `require('./anotherModule')` one or more times to load other, private modules.

Node.js's algorithm for require (module)

In Node.js, modules are either stored in a single file, as discussed previously, or as a directory with particular characteristics. There are several ways to specify module names and several ways to organize module deployment in the filesystem. It's quite flexible, especially when used with the npm package management system for Node.js.

Module identifiers and path names

Generally speaking, the module name is a pathname but with the file extension removed. Earlier, when we wrote `require('./simple')`, Node.js knew to add `.js` to the filename and load in `simple.js`.

Modules whose filenames end in `.js` are of course expected to be written in JavaScript. Node.js also supports binary code native libraries as Node.js modules, whose filename extension is `.node`. It's outside the scope of this book to discuss implementation of native code Node.js modules. This gives you enough knowledge to recognize them when you come across one.

Some Node.js modules are not files in the filesystem, but they are baked into the Node.js executable. These are the core modules, the ones documented on nodejs.org. Their original existence is as files in the Node.js source tree, but the build process compiles them into the binary Node.js executable.

There are three types of module identifiers: *relative*, *absolute*, and *top-level*

- **Relative module identifiers:** These begin with `./` or `../` and absolute identifiers begin with `/`. The module name is identical with POSIX filesystem semantics. The resulting 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 `/` and are, of course, looked for in the root of the filesystem, but this is not a recommended practice.
- **Top-level module identifiers:** These begin with none of 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.

The search begins in the directory containing the file calling `require()`. If that directory contains a `node_modules` directory, which then contains either a matching directory module or a matching file module, then the search is satisfied. If the local `node_modules` directory does not contain a suitable module, it tries again in the parent directory, and it will continue upward in the filesystem until it either finds a suitable module or it reaches the root directory.

That is, with a `require` call in `/home/david/projects/notes/foo.js`, the following directories will be consulted:

- `/home/david/projects/notes/node_modules`
- `/home/david/projects/node_modules`
- `/home/david/node_modules`
- `/home/node_modules`
- `/node_modules`

If the module is not found through this search, there are global folders in which folders can be located. The first is specified in the `NODE_PATH` environment variable. This is interpreted as a colon-delimited list of absolute paths similar to the `PATH` environment variable. Node.js will search those directories for a matching module.



The `NODE_PATH` approach is not recommended because of surprising behavior, which can happen if people are unaware this variable must be set.

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 are beginning to recommend against using global modules. 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 which was magically installed elsewhere on the system? Will you remember to deploy such files?

An example application directory structure

Let's take a look at files and directories in a filesystem.

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.

Name	Date Modified
<code>app.js</code>	Jul 3, 2013, 9:14 AM
<code>models-sequelize</code>	Jan 21, 2016, 9:40 PM
<code>notes.js</code>	Jun 28, 2013, 8:34 AM
<code>setup.js</code>	Mar 24, 2013, 2:02 PM
<code>users.js</code>	Jun 28, 2013, 8:35 AM
<code>node_modules</code>	Today, 1:06 PM
<code>async</code>	Today, 1:06 PM
<code>CHANGELOG.md</code>	Jan 7, 2016, 4:03 PM
<code>dist</code>	Today, 1:06 PM
<code>lib</code>	Jan 7, 2016, 2:59 PM
<code>async.js</code>	Jan 7, 2016, 2:59 PM
<code>LICENSE</code>	May 19, 2015, 2:03 PM
<code>package.json</code>	Today, 1:06 PM
<code>README.md</code>	Jan 7, 2016, 3:18 PM
<code>connect-flash</code>	Today, 1:06 PM
<code>connect-mysql-session</code>	Today, 1:06 PM
<code>ejs</code>	Today, 1:06 PM
<code>express</code>	Today, 1:06 PM
<code>mongoose</code>	Today, 1:07 PM
<code>passport</code>	Today, 1:06 PM
<code>passport-local</code>	Today, 1:06 PM
<code>sequelize</code>	Today, 1:06 PM
<code>socket.io</code>	Today, 1:06 PM
<code>package.json</code>	Apr 21, 2013, 2:17 PM
<code>public</code>	Jan 21, 2016, 9:40 PM
<code>routes</code>	Jan 21, 2016, 9:40 PM
<code>views</code>	Jan 21, 2016, 9:40 PM

The `async` module can be used with `require('async')` in any file shown here. Its `package.json` must include the following attribute:

```
"main": "./lib/async.js"
```

Then, when `require('async')` is called, the `node_modules/async/lib/async.js` file is loaded.

For `app.js` to load `models-sequelize/notes.js`, it calls the following:

```
var notesModel = require('./models-sequelize/notes');
```

Use the following code to do the reverse in `models-sequelize/notes.js`:

```
var app = require('../app');
```

Any of the modules in the `node_modules` directory can contain other `node_modules` directories. Indeed, the two Passport modules do include `node_modules` directories. In such a case, the search algorithm works as described earlier. Since the search proceeds upward in the directory hierarchy, a `require` call will not resolve to a directory deeper in the directory hierarchy.

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` (Redhat/Fedora), MacPorts (Mac OS X), CPAN (Perl), or PEAR (PHP). Its' purpose is publishing and distributing Node.js packages over the Internet using a simple command-line interface. With npm, you can quickly find packages to serve specific purposes, download them, install them, and manage packages you've already installed.

The `npm` package defines a package format for Node.js 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 to build in 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 we just referred to 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` are the CommonJS Packages/1.0 specification. The documentation for npm's `package.json` implementation is accessed using the following command:

```
$ npm help json
```

A basic package.json file is as follows:

```
{ "name": "packageName",
  "version": "1.0",
  "main": "mainModuleName",
  "modules": {
    "mod1": "lib/mod1",
    "mod2": "lib/mod2"
  }
}
```

The file is in JSON format, which, as a JavaScript programmer, you should be familiar with.

The most important tags are `name` and `version`. The name will appear 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 at <http://npmjs.com> or with the following command:

```
$ npm search packageName
```

The `main` tag is treated the same as we discussed in the previous section on directory modules. It defines which file module will be returned when invoking `require('packageName')`. Packages can contain many modules within themselves and they can be listed in the `modules` list.

Packages can be bundled as tar-gzip (tarballs), especially to send them over the Internet.

A package can declare dependencies on other packages. That way, `npm` can automatically install other modules required by the module being installed. Dependencies are declared as follows:

```
"dependencies": {
  "foo": "1.0.0 - 2.x.x",
  "bar": ">=1.0.2 <2.1.2"
}
```

The `description` and `keyword` fields help people find the package when searching in an `npm` repository (<http://search.npmjs.org>). Ownership of a package can be documented in the `homepage`, `author`, or `contributors` fields:

```
"description": "My wonderful packages that walks dogs",
"homepage": "http://npm.dogs.org/dogwalker/",
"author": "dogwhisperer@dogs.org"
```

Some `npm` packages provide executable programs meant to be in the user's PATH. These are declared using the `bin` tag. It's a map of command names to the script which implements that command. The command scripts are installed into the directory containing the node executable using the name given:

```
bin: {  
  'nodeload.js': './nodeload.js',  
  'nl.js': './nl.js'  
},
```

The `directories` tag describes the package directory structure. The `lib` directory is automatically scanned for modules to load. There are other directory tags for binaries, manuals, and documentation:

```
directories: { lib: './lib', bin: './bin' },
```

The `script` tags are script commands run at various events in the life cycle of the package. These events include `install`, `activate`, `uninstall`, `update`, and more. For more information about script commands, use the following command:

```
$ npm help scripts
```

We've already used the `scripts` feature when showing how to set up Babel. We'll use these later for automating the build, test, and execution processes.

This was only a taste of the `npm` package format; see the documentation (`npm help json`) for more.

Finding npm packages

By default, `npm` modules 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 that registry.

The npm package also has a command-line search function to consult the same index:

NAME	DESCRIPTION
acoustid	Get music metadata from AcoustID Web Service
allmuz	Search, stream and download MP3s from Allmuz.
audio-converter	A utility tool for batch converting wave files to ogg/mp3...
audio-editor	An audio editor built with React
audio-metadata	Extract metadata from audio files
audio-tag-injector	Automatically adds html5 audio elements inline beside links...
audio-type	Detect the audio type of a Buffer/Uint8Array
audio-vs1053b-textspeech	Text-to-Speech module for tessel audio module (vs1053b),...
audioconcat	Concat multiple audio files into a unique one (uses ffmpeg)
audiohub	Command Line Audio Playback From The Terminal
audiosource	utility for managing audio buffers
avconv_id3	Read and write media metadata using avconv
bfysong	Node module and CLI to beautify song files (.mp3) by...
browser-media-mime-type	mime type lookup for browser video and audio
choon	Found a tune (choon) on YouTube? Download mp3s directly...
cmp3-test	react component
code2mp3	A library to convert your code to music
cordova-plugin-audiometadata	BlackBerry 10 Community Contributed AudioMetaData API
cordova-plugin-nativeaudio	Cordova/PhoneGap Plugin for low latency Native Audio...
cylon-audio	Cylon module for Audio
daisuke	A simple MP3 parser that returns stat infos in a similar...
desktop-mp3-player	deskop music player
djesbe	A terminal based audio player
douban.fm	a tiny and smart cli player of douban.fm in Node.js
downtify	Despotify Mp3 Downloader
dstrpy	Spotify to mp3
exiftool	Metadata extraction from numerous filetypes including JPEG,...
eyed3	A wrapper for ID3 manipulation using eyeD3
festival	convert text into a mp3 file with festival and LAME
ffmetadata	Read and write media metadata using ffmpeg
file-type	Detect the file type of a Buffer/Uint8Array

Of course, upon finding a module, it's installed as follows:

```
$ npm install acoustid
```

After installing a module you may want to see the documentation, which would be on the module's website. The homepage tag in package.json lists that URL. The easiest way to look at the package.json file is with the npm view command, as follows:

```
$ npm view akashacms
...
{
  name: 'akashacms',
  description: 'A content management system to generate static websites
using latest HTML5/CSS3/JS goodness.',
  'dist-tags': { latest: '0.4.16' },
  versions:
  [ '0.0.1',
```

```
...
  author: 'David Herron <david@davidherron.com> (http://davidherron.com)',
  repository: { type: 'git', url: 'git://github.com/akashacms/akashacms.git' },
  homepage: 'http://akashacms.com/',
...

```

You can use `npm view` to extract any tag from `package.json`, like the following which lets you view just the `homepage` tag:

```
$ npm view akashacms homepage
http://akashacms.org/
```

Other fields in the `package.json` can be viewed by simply giving the desired tag name.

Other npm commands

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.
Or, see the website: http://docs.npmjs.com
```

Installing an npm package

The `npm install` command makes it easy to install packages upon finding the 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.

If you instead want to install a module globally, 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
```

Initializing a new npm package

If you want to create a new package, you can create the package.json file by hand or you can get npm's help. The `npm init` command leads you through a little dialog to get starting values for the package.json file.

```
MacBook-Pro-2:~ david$ mkdir new
MacBook-Pro-2:~ david$ cd new
MacBook-Pro-2:new david$ 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> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (new)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Users/david/new/package.json:

{
  "name": "new",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this ok? (yes) yes
MacBook-Pro-2:new david$
```

Once you get through the questions, the package.json file is written to disk.

Maintaining package dependencies with npm

In the preceding screenshot, you'll see that npm instructs you how to add package dependencies to package.json. What happens is npm looks in package.json for the dependencies or devDependencies field, and it will automatically install the mentioned npm packages.

Suppose you've run this:

```
$ npm install akashacms --save
```

In response, npm will add a dependencies tag to package.json:

```
"dependencies": {  
    "akashacms": "^0.4.16"  
}
```

Now, when your application is installed, npm will automatically also install any packages listed in the dependencies.

The devDependencies are modules used during development. That field is initialized the same as above but with the --save-dev flag.

By default, when an npm install is run, modules listed in both dependencies and devDependencies are installed. Of course, the purpose for having two lists is to not install the devDependencies in some cases:

```
$ npm install --production
```

This installs only the modules listed in dependencies and none of the devDependencies modules.

In the **Twelve-Factor** application model, it's suggested that we explicitly identify the dependencies required by the application. On the Node.js platform, npm gives us this dependencies section, including a flexible mechanism to declare compatible package versions by their version number.

Throughout this book, we'll do the following:

```
$ npm install express@4.x --save
```

This combination installs the specified version, and adds the dependency declaration to the package.json file. Using a version number such as 4.x is a convenience because it will automatically install the latest of a given version number range.

For example, we know that Express 5.x is in its alpha phase at the time of writing this, with the full release some time in the future, and that 5.x will introduce changes that might break the code shown in this book. By instructing you to install 4.x, you'll be insulated from potential breaking changes.

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, or 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.

You must first identify whether the problem exists in the package or in your code. Once you've determined that it's a problem in another package, it's time to 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. The package.json dependencies section lets you reference packages in many ways other than npm version numbers. For example, you can list a URL to a tarball or a reference to repositories on services such as Github. Refer to "npm help package.json" for more information on your options. Consider the following example:

```
"dependencies": {  
    "express": "http://example.com/my/patched/express.tar.gz"  
}
```

During the development of this book, an example of breakage due to a Node.js platform update occurred. Node.js 6.0 was released, bringing with it many great new features, including deeper compatibility with ES-2015 language features. It also introduced changes that broke some modules, including the sqlite3 module we will use starting in *Chapter 7, Data Storage and Retrieval*. An updated release was quickly made, but this begs a question of what to do when this happens.

A situation like this generally prevents us from moving to the newer platform version until all our dependent packages work with the new version. The specific action to take depends on if, or how badly, the new platform version breaks your application.

It is for these reasons that the Twelve-Factor application model recommends explicitly declaring specific dependencies to install. Listing an explicit version number in `package.json` means that the package maintainer can release a new version, but you're in control of if, or when, you adopt that version.

Declaring Node.js version compatibility

It's important that your Node.js software run on the correct version of Node.js. Breaking changes are introduced from time to time in the platform. It's either intentional when new features are added or unintentional when a bug is added. What's most common is to require a specific Node.js version, or later, because of which release in which feature was added.

For example, the ES2015 features started to be implemented in the 4.2.x release, and a large number of other ES2015 features were added in 6.0.

This dependency is declared in `package.json` using the `engines` tag:

```
"engines": {  
  "node": ">= \"5.x\""  
}
```

This, of course, uses the same version number matching that we just discussed.

Updating outdated packages you've installed

The coder codes, updating their package, leaving you in their dust unless you keep up.

To find out if 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 the outdated packages is very simple:

```
$ npm update express  
$ npm update
```

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. While you can rent a private npm repository, there's another way. You can install packages from other locations.

Details about this are in `package.json` help.

Your package dependency can be any URL that downloads a tarball, that is, a `.tar.gz` file:

```
$ npm install http://dev.mydomain.com/path/to/v1.2.3/packageName.tar.gz  
--save
```

One way to use this from a Github repository is that each time a *release* is made, a new tarball is generated at a versioned URL. Simply navigate to the **Releases** tab of the Github repository.

There's also direct support for Git URLs' even ones using SSH authentication. Since Git URLs can include a hash code for a specific repository revision, branch, or tag, this is also versioned:

```
$ npm install git+ssh://user@hostname:project.git#tag --save
```

The `#tag` portion is where you can specify the release tag. For Git repositories on Github, there's a shortcut:

```
$ npm install strongloop/express --save
```

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 can be found at <https://docs.npmjs.com/getting-started/publishing-npm-packages>.

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 over 250,000 packages in the repository, with well over 400 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.

Package version numbers

We've been showing version numbers all along without explaining how npm uses them.

Node.js doesn't know anything about version numbers. It knows about modules, and it can treat a directory structure as if it were a module. It has a fairly rich system of looking for modules, but it doesn't know version numbers.

The npm package, however, knows about version numbers. It uses the Semantic Versioning model (<http://semver.org>), and as we saw, you can install modules over the Internet, query for out-of-date modules, and update them with npm. All of this is version controlled, so let's take a closer look at the things npm can do with version numbers and version tags.

Earlier, we used `npm list` to list installed packages, and the listing includes version numbers of installed packages. If instead, you wish to see the version number of a specific module, type the following command:

```
$ npm view express version  
4.13.4
```

Whenever `npm` commands take a package name, you can append a version number or version tag to the package name. This way, you can deal with specific package versions if needed. Sometimes, package maintainers introduce bugs that you want to avoid. The dependencies in your application can list a specific version number. You can list specific version numbers, changing the version dependency only when you're satisfied it is safe to do so. The `npm install` command, and several other `npm` commands, take version numbers as follows:

```
$ npm install express@2.3.1
```

The `npm help package.json` documentation has a full discussion of what you can do with version number strings, including these 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`

One feature of the Twelve-Factor methodology is step 2, explicitly declaring your dependencies. We've already touched on this, but it's worth reiterating and to see how npm makes this easy to accomplish.

Step 1 of the Twelve-Factor methodology 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.

Doing so ensures that each module progresses on its own timeline. For example, a breakage in one module is easy to back out by changing the version dependency in `package.json`.

This gets us to step 2. There are two aspects of this step, one of which is the package versioning we discussed previously. With npm, we can explicitly declare module version numbers in `dependencies` and `devDependencies`. It's extremely easy, then, to ensure that everyone on the team is on the same page, developing against the same versions of the same modules. When it's time to deploy to testing, staging, or production servers, and the deployment script runs `npm install` or `npm update`, the code will use a known version of the module.

The lazy way to declare dependencies is putting `*` in the version field. Doing so uses whatever is the latest version in the npm repository. Maybe this will work for you, until one day the maintainers of that package introduce a bug. You'll head over to the Github site for the package, look in the issue queue, and 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 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?

You don't want installation instructions to be the length of a small book full of arcane steps to ensure correct setup. That's error prone and hard to automate for modern cloud-based computing systems. Instead, you would want to automate as much of the installation and setup as possible.

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.

A quick note about CommonJS

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 `Modules/1.0` spec.

With ES2015, the JavaScript community now has a native module format which is different than the CommonJS spec. At the time of writing this, Node.js does not support that format. If you wish, Babel supports transpiling from the new modules format to the Node.js format.

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, managing installed modules and packages, and saw how Node.js locates modules.

Now that you've learned about modules and packages, we're ready to use them to build applications, which is the topic for the next chapter.

4

HTTP Servers and Clients – A Web Application's First Steps

Now that you've learned about Node.js modules, it's time to put this knowledge to work in building a simple Node.js web application. In this chapter, we'll keep to a simple application, enabling us to explore three different application frameworks for Node.js. In later chapters, we'll build some more complex applications, but before we can walk, we must learn to crawl.

We will cover the following topics in this chapter:

- EventEmitters
- Listening to HTTP events and the HTTP Server object
- HTTP request routing
- ES-2015 template strings
- Building a simple web application with no frameworks
- The Express application framework
- Express middleware functions
- How to deal with computationally intensive code
- The HTTP Client object
- Creating a simple REST service with Express

Sending and receiving events with EventEmitters

EventEmitters are one of the core idioms of Node.js. Many of the core modules are EventEmitters, and also EventEmitters make an excellent skeleton to implement asynchronous programming. EventEmitters have nothing to do with web application development, but they are so much part of the woodwork that you may skip over their existence. It is with the HTTP classes that we will get our first taste of EventEmitters.

In this chapter, we'll work with the `HTTPServer` and `HTTPClient` objects. Both of them are subclasses of `EventEmitter` and rely on it to send events for each step of the HTTP protocol. Understanding `EventEmitter` will help you understand not only these objects but many other objects in Node.js.

The `EventEmitter` object is defined in the `events` module of Node.js. Directly using the `EventEmitter` class means performing `require('events')`. In most cases, you won't require this module, but there are cases where the program needs to implement a subclass of `EventEmitter`.

Create a file named `pulser.js` that will contain the following code. It shows both sending as well as receiving events while directly using the `EventEmitter` class:

```
var events = require('events');
var util   = require('util');

// Define the Pulser object

function Pulser() {
  events.EventEmitter.call(this);
}

util.inherits(Pulser, events.EventEmitter);

Pulser.prototype.start = function() {
  setInterval(() => {
    util.log('>>> pulse');
    this.emit('pulse');
    util.log('<<< pulse');
  }, 1000);
};
```

This defines a `Pulser` class, which inherits from `EventEmitter` (using `util.inherits`). Its purpose is to send timed events, once a second, to any listeners. The `start` method uses `setInterval` to kick off repeated callback execution, scheduled for every second, and call `emit` to send the `pulse` events to any listeners.

This much could be the starting point of a general-purpose module to send events at a given interval. Instead, we're just exploring how `EventEmitter` works.

Now, let's see how to use the `Pulser` object by adding the following code to `pulser.js`:

```
// Instantiate a Pulser object
var pulser = new Pulser();
// Handler function
pulser.on('pulse', () => {
  util.log('pulse received');
});
// Start it pulsing
pulser.start();
```

Here we create a `Pulser` object and consume its pulse events. Calling `pulser.on('pulse')` sets up connections for the `pulse` events to invoke the callback function. It then calls the `start` method to get the process going.

Enter this into a file and name the file `pulser.js`. When you run it, you should see the following output:

```
$ node pulser.js
19 Apr 16:58:04 - >>> pulse
19 Apr 16:58:04 - pulse received
19 Apr 16:58:04 - <<<< pulse
19 Apr 16:58:05 - >>> pulse
19 Apr 16:58:05 - pulse received
19 Apr 16:58:05 - <<<< pulse
19 Apr 16:58:06 - >>> pulse
19 Apr 16:58:06 - pulse received
19 Apr 16:58:06 - <<<< pulse
```

The `EventEmitter` theory

The `EventEmitter` event has many names, and it can be anything that makes sense to you. 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, no registry of event names. Simply making a call to `.emit` is enough to define an event name.



By convention, the event name `error` indicates errors.



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.

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 that event, the data appears as arguments to the callback function. Your program would listen to such an event as follows:

```
emitter.on('eventName', (data1, data2, ...) => {
  // 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 received, any action taken, or any error occurred.

HTTP server applications

The HTTP server 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 that protocol. In most cases, you'll be able to use an application framework such as Express that hides the HTTP protocol details, allowing the programmer to focus on business logic.

We already saw a simple HTTP server application in *Chapter 2, Setting up Node.js*, which is as follows:

```
var http = require('http');
http.createServer((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');
```

The `http.createServer` function creates an `http.Server` object. Because it is an `EventEmitter`, this can be written in another way to make that fact explicit:

```
var http = require('http');
var 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');
```

The request event takes a function, which receives request and response objects. The request object has data from the web browser, while the response object is used to gather the data to be sent in the response. The listen function causes the server to start listening and arranging to dispatch an event for every request arriving from a web browser.

Now, let's look at something more interesting with different actions based on the URL.

Create a new file named `server.js` containing the following code:

```
var http = require('http');
var util = require('util');
var url = require('url');
var os = require('os');

var server = http.createServer();
server.on('request', (req, res) => {
  var requrl = url.parse(req.url, true);
  if (requrl.pathname === '/') {
    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>`);
  } else if (requrl.pathname === "/osinfo") {
    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()} ${os.arch()} ${os.release()}</td></tr>
```

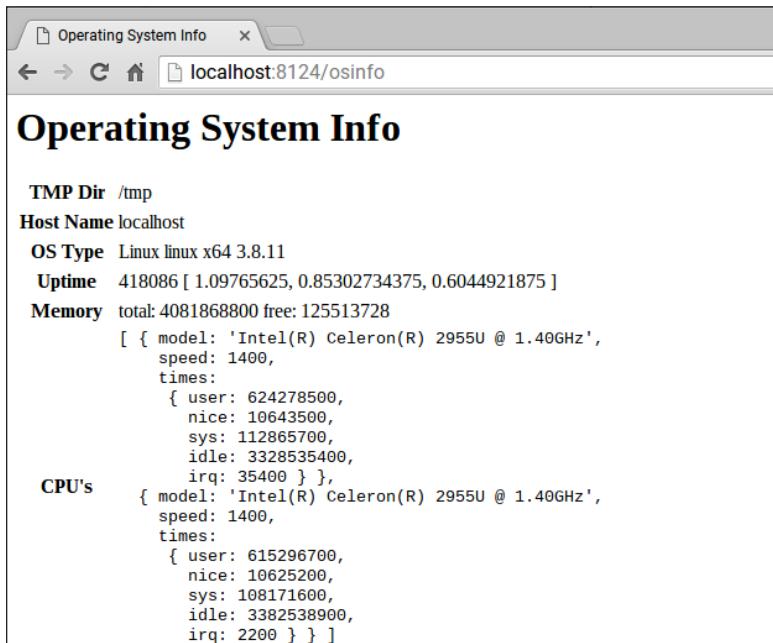
```
<tr><th>Uptime</th><td>${os.uptime()} ${util.inspect(os.loadavg())}</td></tr>
<tr><th>Memory</th><td>total: ${os.totalmem()} free: ${os.freemem()}</td></tr>
<tr><th>CPU's</th><td><pre>${util.inspect(os.cpus())}</pre></td></tr>
<tr><th>Network</th><td><pre>${util.inspect(os.networkInterfaces())}</pre></td></tr>
</table>
</body></html>`);
} else {
    res.writeHead(404, {'Content-Type': 'text/plain'});
    res.end("bad URL " + req.url);
}
});

server.listen(8124);
console.log('listening to http://localhost:8124');
```

To run it, type the following command:

```
$ node server.js
listening to http://localhost:8124
```

This application is meant to be similar to PHP's `sysinfo` function. Node's `os` module is consulted to give information about the server. This example can easily be extended to gather other pieces of data about the server:



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.js`, we consult the `request.url` data to determine which page to show, after parsing (using `url.parse`) to ease the digestion process. In this case, this lets us do a simple comparison of the pathname to determine what to do.

Some web applications care about the HTTP verb (GET, DELETE, POST, and so on) used and must consult the `request.method` field of the `request` object. For example, POST is frequently used for FORM submissions.

The pathname portion of the request URL is used to dispatch the handler code. While this routing method, based on simple string comparison, will work for a small application, it'll quickly become unwieldy. 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 while looking at Express later in the *Getting started with Express* section.

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.

ES-2015 multiline and template strings

The previous example showed two of the new features introduced with ES-2015, multiline and template strings. The feature is meant to simplify our life while creating text strings.

The existing string representations use single quotes and double quotes. Template strings are delimited with the backtick character that's also known as the *grave accent* (or, in French, accent grave):

```
`template string text`
```

Before ES-2015, one way to implement a multiline string was this 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')
```

Yes, this is the code used in the same example in previous versions of this book. This is what we can do with ES-2015:

```
`<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 on the last line, and everything in between is part of our string.

The real purpose of the *template strings* feature is supporting strings where we can easily substitute values directly into the string. Most other programming languages support this ability, and now JavaScript does too.

Pre-ES-2015, a programmer would do 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())
```

Again, this is extracted from the same example 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. It can be a simple mathematical expression, a variable reference, or, as in this case, a function call.

The last thing to mention is a matter of indentation. In normal coding, one indents a long argument list to the same level as the containing function call. But, for these multiline string examples, the text content is flush with column 0. What's up?

This may impede readability of your code, so it's worth weighing code readability against another issue, excess characters in the HTML output. The blanks we would use to indent the code for readability will become part of the string and will be output in the HTML. By making the code flush with column 0, we don't add excess blanks to the output at the cost of some code readability.

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 the HTTP Server events. It could be a useful debugging tool, which also demonstrates how HTTP server objects operate.

Node.js's `HTTP Server` object is an `EventEmitter` and the `HTTP Sniffer` simply listens to every server event, printing out information pertinent to each event.

Create a file named `httpsniffer.js` containing the following code:

```
var util = require('util');
var url = require('url');

exports.sniffOn = function(server) {
  server.on('request', (req, res) => {
    util.log('e_request');
    util.log(reqToString(req));
  });
  server.on('close', errno => { util.log('e_close errno=' + errno); });
  server.on('checkContinue', (req, res) => {
    util.log('e_checkContinue');
    util.log(reqToString(req));
    res.writeContinue();
  });
  server.on('upgrade', (req, socket, head) => {
    util.log('e_upgrade');
    util.log(reqToString(req));
  });
  server.on('clientError', () => { util.log('e_clientError'); });
};

var reqToString = exports.reqToString = function(req) {
  var ret = `req ${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 += req.trailers +'\n';
    return ret;
};
```

That was a lot of code! But the key to it is the `sniffOn` function. When given an HTTP Server function, it uses the `.on` function to attach listener functions that print data about each event emitted by the HTTP Server object. It gives a fairly detailed trace of HTTP traffic on an application.

In order to use it, simply insert this code just before the `listen` function in `server.js`:

```
require('./httpsniffer').sniffOn(server);
server.listen(8124);
console.log('listening to http://localhost:8124');
```

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.js
listening to http://localhost:8124
30 Jan 16:32:39 - request
30 Jan 16:32:39 - request GET 1.1 /
{"protocol":null,"slashes":null,"auth":null,"host":null,"port":null,
"hostname":null,"hash":null,"search":"",
"query":{},
"pathname":"/",
"path":"/",
"href":"/" }

0 host: localhost:8124
1 connection: keep-alive
2 accept: text/html,application/xhtml+xml,application/xml;q=0.9,
image/webp,*/*;q=0.8
3 upgrade-insecure-requests: 1
4 user-agent: Mozilla/5.0 (X11; CrOS x86_64 7520.67.0) AppleWebKit/
537.36 (KHTML, like Gecko) Chrome/47.0.2526.110 Safari/537.36
5 accept-encoding: gzip, deflate, sdch
6 accept-language: en-US,en;q=0.8
[object Object]

30 Jan 16:32:49 - request
30 Jan 16:32:49 - request GET 1.1 /osinfo
{"protocol":null,"slashes":null,"auth":null,"host":null,"port":null,
"hostname":null,"hash":null,"search":"",
"query":{},
"pathname":"/osinfo",
"path":"/osinfo",
"href":"/osinfo"}
```

```
0 host: localhost:8124
1 connection: keep-alive
2 accept: text/html,application/xhtml+xml,application/xml;q=0.9,image
/webp,*/*;q=0.8
3 upgrade-insecure-requests: 1
4 user-agent: Mozilla/5.0 (X11; CrOS x86_64 7520.67.0) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/47.0.2526.110 Safari/537.36
5 referer: http://localhost:8124/
6 accept-encoding: gzip, deflate, sdch
7 accept-language: en-US,en;q=0.8
[object Object]
```

You now have a tool for snooping on `HTTPServer` events. This simple technique prints a detailed log of event data. The pattern can be used for any `EventEmitter` object. You can use this technique as a way to inspect the actual behavior of `EventEmitter` objects in your program.

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 a car, typical web application programming is better done at a higher level. It's better to abstract away the HTTP details and concentrate on your application.

The Node.js developer community has developed quite a few modules to help with different aspects of abstracting away HTTP protocol details. You can take a look at a couple of curated lists at <http://nodeframework.com/> and <https://github.com/vndmtr/x/awesome-nodejs>.

One reason to use a web framework is that they often provide all the usual best practices that have come from web application development of over 20 years. They have been listed as follows:

- 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 page size or execution time

Web frameworks help you invest your time in the task without getting lost in the details of implementing 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 providing prepackaged functions that take care of the details.

Getting started with Express

Express is perhaps the most popular Node.js web app framework. It's so popular that it's part of the MEAN Stack acronym. MEAN refers to MongoDB, ExpressJS, AngularJS, and Node.js. Express is described as being *Sinatra-like*, referring to a popular Ruby application framework, and that it isn't very opinionated. 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/>.

Shortly, we'll implement a simple application to calculate Fibonacci numbers using Express, and in later chapters, we'll do quite a bit more with Express. We'll also explore how to mitigate the performance problems from computationally intensive code we discussed earlier.

Let's start by installing the **Express Generator**. While we can just start writing some code, the Express Generator gives a blank starting application. We'll take that and modify it.

Install it using 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 was to use the `-g` tag for a global install. We're also using an explicit version number to ensure compatibility.

Earlier, we discussed how many now recommend against installing modules globally. In the Twelve-Factor model, it's strongly recommended to not install global dependencies, and that's what we're doing.

The result is that an `express` command is installed in the `./node_modules/.bin` directory:

```
$ ls node_modules/.bin/  
express
```

To run the command:

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

```
Usage: express [options] [dir]
```

Options:

```
-h, --help          output usage information
-v, --version       output the version number
-e, --ejs           add ejs engine support (defaults to jade)
--hbs              add handlebars engine support
-H, --hogan         add hogan.js engine support
-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
```

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. Fortunately, modifying the `PATH` environment variable is an easy solution:

```
$ export PATH=node_modules/.bin:${PATH}
```

This works for a bash shell. But, for csh users, try this:

```
$ setenv PATH node_modules/.bin:${PATH}
```

With the `PATH` environment variable set like this, the `express` command can be directly executed. Let's now use it to generate a blank starter application.

Now that you've installed express-generator in the fibonacci directory, use it to set up the blank framework application:

```
[laptop:fibonacci david$ express --ejs --git
[destination is not empty, continue? [y/N] y

  create : .
  create : ./package.json
  create : ./app.js
  create : ./gitignore
  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/index.ejs
  create : ./views/error.ejs
  create : ./bin
  create : ./bin/www

  install dependencies:
    $ cd . && npm install

  run the app:
    $ DEBUG=fibonacci:* npm start

[laptop:fibonacci david$ npm uninstall express-generator
unbuild express-generator@4.13.1
laptop:fibonacci david$ ]
```

This created for us a bunch of files which we'll walk through in a minute. The node_modules directory still has the express-generator module, which is now not useful. We can just leave it there and ignore it, or we can add it to the devDependencies of the package.json it generated. Alternatively, we can uninstall it as shown in the preceding screenshot.

The next thing to do is run the blank application in the way we're told. The command shown, npm start, relies on a section of the supplied package.json file:

```
"scripts": {
  "start": "node ./bin/www"
},
```

The npm tool supports scripts that are ways to automate various tasks. We'll use this capability throughout the book to do various things. Most npm scripts are run with the `npm run scriptName` command, but the `start` command is explicitly recognized by npm and can be run as shown previously.

The first step is to install the dependencies (`npm install`), then to start the application:

```
[laptop:fibonacci david$ npm start

> fibonacci@0.0.0 start /Users/david/fibonacci
> DEBUG=fibonacci:* node ./bin/www

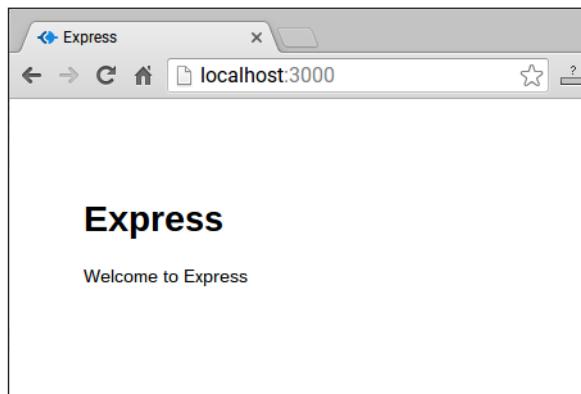
fibonacci:server Listening on port 3000 +0ms
```

We can modify the supplied npm `start` script to print debugging information on the screen. Change the `scripts` section to the following:

```
"scripts": {
  "start": "DEBUG=fibonacci:* node ./bin/www"
},
```

Adding `DEBUG=fibonacci:*` this way sets an environment variable, `DEBUG`, as shown in the screenshot. In Express applications, the `DEBUG` variable turns on debugging code, which prints an execution trace that's useful for debugging.

Since the output says it is listening on port 3000, we direct our browser to `http://localhost:3000/` and see the following output:



Walking through the default Express application

We have a working blank Express application; let's look at what was generated for us. We're doing this to familiarize ourselves with Express before diving in to start coding our Fibonacci application.

Because we used the `--ejs` option, this application is set up to use the EJS template engine. The template engine chosen assists with generating HTML. EJS is documented at <http://ejs.co/>.

The `views` directory contains two files, `error.ejs` and `index.ejs`. These are both EJS templates that we'll modify later.

The `routes` directory contains the initial routing setup, that is, the code to handle specific URLs. We'll modify these later.

The `public` directory will contain 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. That's a Node.js script, which initializes the `HTTPServer` objects, starts it listening on a TCP port, and calls the last file we'll discuss, `app.js`. These scripts initialize Express, hook up the routing modules, and do 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`:

```
var express = require('express');
...
var app = express();
...
module.exports = app;
```

This means that `app.js` is a module that exports the object returned by the `express` module.

Now, let's turn to the `www` script. The first thing to see is it starts with this 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.

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);
```

We see where port 3000 comes from; it's a parameter to the `normalizePort` function. We also see that setting the `PORT` environment variable will override the default port 3000. Try running the following command:

```
$ PORT=4242 DEBUG=fibonacci:* npm start
```

The application now tells you that it's listening on port 4242, where you can ponder the meaning of life.

The `app` object is next passed to `http.createServer()`. A look in the Node.js documentation tells us this function takes a `requestListener`, which is simply a function that takes the `request` and `response` objects we've seen previously. This tells us the `app` object is such a function.

Finally, the `www` script starts the server listening on the port we specified.

Let's now walk through `app.js` in more detail:

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
```

This tells Express to look for templates in the `views` directory and to use the EJS templating engine.

The `app.set` function is used for setting 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('/', routes);  
app.use('/users', users);
```

The `app.use` function mounts middleware functions. This is an important piece of Express jargon we will discuss shortly. At the moment, let's say that middleware functions are executed during the processing of routes. This means all the features named here are enabled in `app.js`:

- Logging is enabled using the Morgan request logger. Visit <https://www.npmjs.com/package/morgan> for its documentation.
- The `body-parser` module handles parsing HTTP request bodies. Visit <https://www.npmjs.com/package/body-parser> for its documentation.
- The `cookie-parser` module is used to parse HTTP cookies. Visit <https://www.npmjs.com/package/cookie-parser> for its documentation.
- A static file web server configured to serve the asset files in the `public` directory.
- Two router modules, `routes` and `users`, to set up which functions handle which URLs.

The Express middleware

Let's round out the walkthrough of `app.js` by discussing what middleware functions do for our application. We have an example at the end of the script:

```
app.use(function(req, res, next) {  
  var err = new Error('Not found');  
  err.status = 404;  
  next(err);  
});
```

The comment says catch 404 and forward to error handler. As you probably know, an HTTP 404 status means the requested resource was not found. We need to tell the user their request wasn't satisfied, and this is the first step in doing so. Before getting to the last step of reporting this error, you must learn how middleware works.

We do have a middleware function right in front of us. Refer to its documentation 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` of the Node.js HTTP request object. However, Express expands the objects with additional data and capabilities. The last, `next`, is a callback function controlling when the request-response cycle ends, and it can be used to send errors down the middleware pipeline.

The incoming request gets handled by the first middleware function, then the next, then the next, and so on. Each time the request is to be passed down the chain of middleware functions, the next function is called. If next is called with an error object, as shown here, an error is being signaled. Otherwise, the control simply passes to the next middleware function in the chain.

What happens if next is not called? The HTTP request will *hang* because no response has been given. A middleware function gives a response when it calls functions on the `response` object, such as `res.send` or `res.render`.

For example, consider the inclusion of `app.js`:

```
app.get('/', function(req, res) {  
    res.send('Hello World!');  
});
```

This does not call `next`, but instead calls `res.send`. This is the correct method of ending the request-response cycle, by sending a response (`res.send`) to the request. If neither `next` nor `res.send` is called, the request never gets a response.

Hence, a middleware function does one of the following four things:

- Executes its own business logic. The request logger middleware shown earlier is an example.
- Modifies the request or response objects. Both the `body-parser` and `cookie-parser` do so, looking for data to add to the `request` object.
- Calls `next` to proceed to the next middleware function or else signals an error.
- Sends a response, ending the cycle.

The ordering of middleware execution depends on the order they're added to the `app` object. The first added is executed first, and so on.

Middleware and request paths

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 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 this example, we're suggesting a user profiles service, and that for this URL we want to display information about the named user.

Another way to use a middleware function is on a specific HTTP request method. With `app.use`, any request will be matched, but in truth, GET requests are supposed to behave differently than POST requests. You call `app.METHOD` where `METHOD` matches one of the HTTP request verbs. That is, `app.get` matches the GET method, `app.post` matches POST, and so on.

Finally, we get to the `router` object. This is a kind of middleware used explicitly for routing requests based on their URL. Take a look at `routes/users.js`:

```
var express = require('express');
var router = express.Router();
router.get('/', function(req, res, next) {
  res.send('respond with a resource');
});
module.exports = router;
```

We have a module whose `exports` object is a router. This router has only one route, but it can have any number of routes you think is appropriate.

Back in `app.js`, this is added as follows:

```
app.use('/users', users);
```

All the functions we discussed for the `app` object apply to the `router` object. If the request matches, the router is given the request for its own chain of processing functions. An important detail is that the request URL prefix is stripped when the request is passed to the router instance.

You'll notice that the `router.get` in `users.js` matches `'/'` and that this router is mounted on `'/users'`. In effect that `router.get` matches `/users` as well, but because the prefix was stripped, it specifies `'/'` instead. This means a router can be mounted on different path prefixes, without having to change the router implementation.

Error handling

Now, we can finally get back to the generated `app.js`, the `404` error (*page not found*), and any other errors the application might want to show to the user.

A middleware function indicates an error by passing a value to the next function call. Once Express sees an error, it will skip any remaining non-error routing, and it will only pass it to error handlers instead. An error handler function has a different signature than what we saw earlier.

In `app.js` that we're examining, this is our error handler:

```
app.use(function(err, req, res, next) {  
  res.status(err.status || 500);  
  res.render('error', {  
    message: err.message,  
    error: {}  
  });  
});
```

Error handler functions take four parameters, with `err` added to the familiar `req`, `res`, and `next`. For this handler, we use `res.status` to set the HTTP response status code, and we use `res.render` to format an HTML response using the `views/error.ejs` template. The `res.render` function takes data, rendering it with a template to produce HTML.

This means any error in our application will land here, bypassing any remaining middleware functions.

Calculating the Fibonacci sequence with an Express application

The Fibonacci numbers are the integer sequence: `0 1 1 2 3 5 8 13 21 34 ...`

Each entry in the list is the sum of the previous two entries in the list, and the sequence was invented in 1202 by Leonardo of Pisa, who was also known as Fibonacci. One method to calculate entries in the Fibonacci sequence is the recursive algorithm we showed earlier. We will create an Express application that uses the Fibonacci implementation and then 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 had you name that application `fibonacci` for a reason. We were thinking ahead.

In `app.js`, make the following changes:

- Delete the line `require('./routes/users')` and replace it with `var fibonacci = require('./routes/fibonacci');`
- Delete the line: `app.use('/users', users)` and replace it with `app.use('/fibonacci', fibonacci);`

For the Fibonacci application, we don't need to support users. We need a page to query for a number for which we'll calculate the Fibonacci number. The `fibonacci` module we'll show next will serve that purpose.

In the top-level directory, create a file, `math.js`, containing this extremely simplistic Fibonacci implementation:

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

In the `views` directory, create a file named `top.ejs`:

```
<html>  
<head>  
    <title><%= title %></title>  
    <link rel='stylesheet' href='/stylesheets/style.css' />  
</head>  
<body>  
    <h1><%= title %></h1>  
    <div class='navbar'>  
        <p><a href='/'>home</a>  
        | <a href='/fibonacci'>Fibonacci's</a></p>  
    </div>
```

This file contains the top part of HTML pages we'll send to the users.

Another file named `bottom.ejs` contains the following:

```
</body>  
</html>
```

This file contains the bottom part of the HTML pages.

Change `views/index.ejs` to just contain the following:

```
<% include top %>  
<p>Welcome to the Math calculator</p>  
<% include bottom %>
```

This serves as the front page of our application. While it doesn't contain any functionality, note that `top.ejs` has a link to `/fibonacci`, which we'll look at next.

Together, `top.ejs` and `bottom.ejs` let us use a consistent look and overall page layout without having to replicate it in every page.

Create a file, `views/fibonacci.ejs`, containing the following code:

```
<% include top %>
<% if (typeof fiboval !== "undefined") { %>
<p>Fibonacci for <%= fibonum %> is <%= fiboval %></p>
<hr/>
<% } %>
<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>
<% include bottom %>
```



Remember that the files in `views` are templates into which data is rendered. They serve the **View** aspect of the **Model-View-Controller** paradigm, hence the directory name.

Each of the two views, `views/index.ejs` and `views/fibonacci.ejs`, are truncated pieces of a full HTML page. Where does the rest come from? It comes from `top.ejs` and `bottom.ejs`, which are meant to be included by the every template to give us a consistent page layout.

In the `routes` directory, delete the `user.js` module. It is generated by the Express framework, but we will not use it in this application.

In `routes/index.js`, change the router function to the following:

```
/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: "Math Calculator" });
});
```

Then, finally, in the `routes` directory, create a file named `fibonacci.js` containing the following code:

```
var express = require('express');
var router = express.Router();

var 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;
```

In `package.json`, change the `scripts` section to the following:

```
"scripts": {
  "start": "DEBUG=fibonacci:* node ./bin/www"
},
```

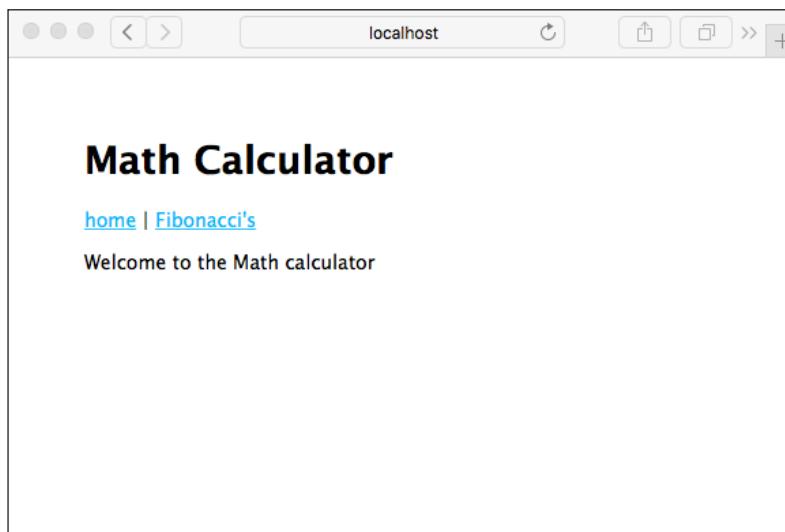
This lets us use `npm start` to run the script and always have debugging messages enabled. And, now we're ready to do so:

```
$ npm start

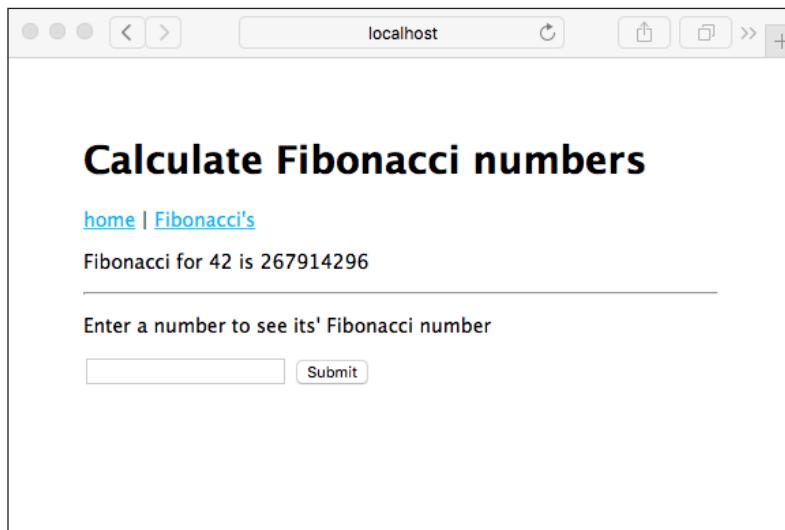
> fibonacci@0.0.0 start /Users/david/chap04/fibonacci
> DEBUG=fibonacci:* node ./bin/www

fibonacci:server Listening on port 3000 +0ms
```

As it suggests, you can visit `http://localhost:3000/` and see what we have:



This page is rendered from the `views/index.ejs` template. Simply click on the Fibonacci link to go to the next page, which is of course rendered from the `views/fibonacci.ejs` 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):



Let's walk through the application to discuss how it works.

There are two routes in `app.js`, the route for `/`, which is handled by `routes/index.js`, and the route for `/fibonacci`, which is handled by `routes/fibonacci.js`.

The `res.render` function renders the named template, using the provided data values and emits the result as an HTTP response. For the home page of this application, the rendering code (`routes/index.js`) and template (`views/index.ejs`) aren't much, and it is on the Fibonacci page where all the action is happening.

The `views/fibonacci.ejs` template contains a form in which the user enters a number. Because it is a GET form, when the user clicks on the **Submit** button, the browser will issue an HTTP GET on the `/fibonacci` URL. What distinguishes one GET 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` and hence nothing to calculate. After the user has entered a number and clicked on **Submit**, there is a `fibonum` and something to calculate.

Express automatically parses the query parameters, making them available as `req.query`. That means `routes/fibonacci.js` can quickly check whether there is a `fibonum`. If there is, it calls the `fibonacci` function to calculate the value.

Earlier, 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.

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 running long computations? To see the effect, open two browser windows, each visiting the Fibonacci page. In one, enter the number 55 or greater, and in the other, enter 10. Note that the second window freezes, and if you leave it running long enough, the answer will eventually pop up in both windows. What's happening is 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 asynchronous I/O is non-blocking and control is quickly returned to the event loop. The naïve 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 blistering fast web server.

In this case, the long-response-time problem is obvious. Response time quickly escalates to the point where you can take a vacation to Tibet during the time it takes to respond with the Fibonacci number!

To see this more clearly, create a file named `fibotimes.js` containing the following code:

```
var math = require('./math');
var util = require('util');

for (var num = 1; num < 80; num++) {
    util.log('Fibonacci for ' + num + ' = ' + math.fibonacci(num));
}
```

Now run it. You will get the following output:

```
$ node fibotimes.js
31 Jan 14:41:28 - Fibonacci for 1 = 1
31 Jan 14:41:28 - Fibonacci for 2 = 1
31 Jan 14:41:28 - Fibonacci for 3 = 2
31 Jan 14:41:28 - Fibonacci for 4 = 3
31 Jan 14:41:28 - Fibonacci for 5 = 5
31 Jan 14:41:28 - Fibonacci for 6 = 8
31 Jan 14:41:28 - Fibonacci for 7 = 13
31 Jan 14:41:28 - Fibonacci for 8 = 21
31 Jan 14:41:28 - Fibonacci for 9 = 34
...
31 Jan 14:42:27 - Fibonacci for 38 = 39088169
31 Jan 14:42:28 - Fibonacci for 39 = 63245986
31 Jan 14:42:31 - Fibonacci for 40 = 102334155
31 Jan 14:42:34 - Fibonacci for 41 = 165580141
```

```
31 Jan 14:42:40 - Fibonacci for 42 = 267914296
31 Jan 14:42:50 - Fibonacci for 43 = 433494437
31 Jan 14:43:06 - Fibonacci for 44 = 701408733
```

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.

There are two general ways in Node.js to solve this problem:

- **Algorithmic refactoring:** Perhaps, like the Fibonacci function we chose, one of your algorithms is suboptimal and can be rewritten to be faster. Or, if not faster, it can be split into callbacks dispatched through the event loop. We'll look at one such method in a moment.
- **Creating a backend service:** Can you imagine a backend server 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.

Algorithmic refactoring

To prove that we have an artificial problem on our hands, here is a much more efficient Fibonacci function:

```
var fibonacciLoop = exports.fibonacciLoop = function(n) {
  var fibos = [];
  fibos[0] = 0;
  fibos[1] = 1;
  fibos[2] = 1;
  for (var 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`, these programs run 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 your head will spin:

```
for (var num = 1; num < 8000; num++) {
  // util.log('Fibonacci for ' + num + ' = ' +
```

```
//           math.fibonacci(num));
util.log('Fibonacci for ' + num + ' = ' +
         math.fibonacciLoop(num));
}
```

Some algorithms aren't so simple to optimize and still take a long time to calculate the result. In this section, we're exploring how to handle inefficient algorithms, and therefore will stick with the inefficient Fibonacci implementation.

It is possible to divide the calculation into chunks and then dispatch computation of those chunks through the event loop. Add the following code to `math.js`:

```
var fibonacciAsync = exports.fibonacciAsync = function(n, done) {
    if (n === 0)
        done(undefined, 0);
    else if (n === 1 || n === 2)
        done(undefined, 1);
    else {
        setImmediate(function() {
            fibonacciAsync(n-1, function(err, val1) {
                if (err) done(err);
                else setImmediate(function() {
                    fibonacciAsync(n-2, function(err, val2) {
                        if (err) done(err);
                        else done(undefined, val1+val2);
                    });
                });
            });
        });
    }
}
```

This converts the `fibonacci` function from a synchronous function to an asynchronous function one with a callback. Using `setImmediate`, each stage of the calculation is managed through Node.js's event loop and 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 silly, inefficient Fibonacci algorithm. All we've done is spread the computation through the event loop.

Because it's an asynchronous function, we will need to change our application code in `router/fibonacci.js` to the following:

```
var express = require('express');
var router = express.Router();

var math = require('../math');
```

```
router.get('/', function(req, res, next) {
  if (req.query.fibonum) {
    math.fibonacciAsync(req.query.fibonum, (err, fiboval) => {
      res.render('fibonacci', {
        title: "Calculate Fibonacci numbers",
        fibonum: req.query.fibonum,
        fiboval: fiboval
      });
    });
  } else {
    res.render('fibonacci', {
      title: "Calculate Fibonacci numbers",
      fiboval: undefined
    });
  }
});
```

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 on the application, entering 55 in one window and in the other start requesting smaller Fibonacci numbers. Unlike the previous behavior, the second browser window will give you Fibonacci numbers while the other is still computing away.

It's up to you, and your specific algorithms, to choose how to best optimize your code and to handle any long-running computations you may have.

Making HTTP Client requests

The next 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 HTTP Client object to do so. However, before we look at that, let's first talk in general about using the HTTP Client object.

Node.js includes an HTTP Client object useful for making HTTP requests. It has enough capability to issue any kind of HTTP request, but, for example, it does not emulate a full browser, so don't get delusions of this being a full-scale test automation tool. Its scope focuses solely on the HTTP protocol. It's possible to build a browser emulator on top of this HTTP client, for example, to build a test automation tool. The HTTP Client object can be used for any kind of HTTP request, such as calling a **Representational State Transfer (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 this code:

```
var http = require('http');
var url  = require('url');
var util = require('util');

var argUrl = process.argv[2];
var parsedUrl = url.parse(argUrl, true);

// The options object is passed to http.request
// telling it the URL to retrieve
var options = {
  host: parsedUrl.hostname,
  port: parsedUrl.port,
  path: parsedUrl.pathname,
  method: 'GET'
};

if (parsedUrl.search) options.path += "?" + parsedUrl.search;

var req = http.request(options);
// Invoked when the request is finished
req.on('response', res => {
  util.log('STATUS: ' + res.statusCode);
  util.log('HEADERS: ' + util.inspect(res.headers));
  res.setEncoding('utf8');
  res.on('data', chunk => { util.log('BODY: ' + chunk); });
  res.on('error', err => { util.log('RESPONSE ERROR: ' + err); });
});
// Invoked on errors
req.on('error', err => { util.log('REQUEST ERROR: ' + err); });
req.end();
```

You can run the script as follows:

```
$ node wget.js http://example.com
31 Jan 15:04:29 - STATUS: 200
31 Jan 15:04:29 - HEADERS: { 'accept-ranges': 'bytes',
  'cache-control': 'max-age=604800',
  'content-type': 'text/html',
  date: 'Sun, 31 Jan 2016 23:04:29 GMT',
  etag: '"359670651+gzip"'}
```

```
expires: 'Sun, 07 Feb 2016 23:04:29 GMT',
'last-modified': 'Fri, 09 Aug 2013 23:54:35 GMT',
server: 'ECS (rhv/818F)',
vary: 'Accept-Encoding',
'x-cache': 'HIT',
'x-ec-custom-error': '1',
'content-length': '1270',
connection: 'close' }
```

There's more in the printout, namely the HTML of the page at <http://example.com/>.

The purpose of `wget.js` is to make an HTTP request and show you voluminous detail of the response.

An HTTP request is initiated with the `http.request` method, as follows:

```
var http = require('http');
var options = {
  host: 'example.com',
  port: 80,
  path: null,
  method: 'GET'
};
var request = http.request(options,
  function(response) { .. });
```

The `options` object describes the request to make, and the `callback` function is called when the response arrives. The `options` object is fairly straightforward with the `host`, `port`, and `path` fields specifying the URL being requested. The `method` field must be one of the HTTP verbs (GET, PUT, POST, and so on). You can also give 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`, which 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`, which is useful for HTTP requests containing data, such as `PUT` or `POST`. This means the `request` object has a `write` function that writes data to the requester. The data format in an HTTP request is specified by the MIME standard originally created to give us better e-mail. Around 1992, the WWW community worked with the MIME standard committee to apply portions of MIME for use in HTTP. HTML forms will post with a `Content-Type` of `multipart/form-data`, for example.

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 inside an Express web application. What that effectively means is to make an HTTP `GET` request to a backend server, which responds with 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 look at the basics of implementing a multi-tier application stack.

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 `HTTP` client object to do so.

Implementing a simple REST server with Express

While Express has a powerful templating system, making it suitable for delivering HTML web pages to browsers, it can also be used to implement a simple REST service. The parameterized URL's we showed earlier (`/user/profile/:id`) can act like parameters to a REST call. And Express makes it easy to return data encoded in JSON.

Now, create a file named `fiboserver.js` containing this code:

```
var math = require('./math');
var express = require('express');
var logger = require('morgan');
var 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);
```

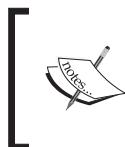
This is a stripped-down Express application that gets right to the point of providing a Fibonacci calculation service. The one route it supports, handles the Fibonacci computation using the same functions we've already worked with.

This is the first time we've seen `res.send` used. It's a flexible way to send responses which 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`.

Then, in `package.json`, add this to the `scripts` section:

```
"server": "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. This is another aspect of the Twelve-Factor application model; to put configuration data in the environment.

Now, let's run it:

```
$ npm run server
```

```
> fibonacci@0.0.0 server /Users/david/chap04/fibonacci
> 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 took to process.

Now, let's create a simple client program, `fiboclient.js`, to programmatically call the Fibonacci service:

```
var http = require('http');
var util = require('util');
[
  "/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 => {
  util.log('requesting ' + path);
  var req = http.request({
    host: "localhost",
    port: 3002,
    path: path,
    method: 'GET'
  }, res => {
    res.on('data', chunk => {
      util.log('BODY: ' + chunk);
    });
  });
  req.end();
});
```

Then, in `package.json`, add this to the `scripts` section:

```
"client": "node ./fiboclient"
```

Then run the client app:

```
$ npm run client
```

```
> fibonacci@0.0.0 client /Users/david/chap04/fibonacci
> node ./fiboclient
```

```
31 Jan 16:37:48 - requesting /fibonacci/30
```

```
31 Jan 16:37:48 - requesting /fibonacci/20
31 Jan 16:37:48 - requesting /fibonacci/10
31 Jan 16:37:48 - requesting /fibonacci/9
31 Jan 16:37:48 - requesting /fibonacci/8
31 Jan 16:37:48 - requesting /fibonacci/7
31 Jan 16:37:48 - requesting /fibonacci/6
31 Jan 16:37:48 - requesting /fibonacci/5
31 Jan 16:37:48 - requesting /fibonacci/4
31 Jan 16:37:48 - requesting /fibonacci/3
31 Jan 16:37:48 - requesting /fibonacci/2
31 Jan 16:37:48 - requesting /fibonacci/1
31 Jan 16:37:48 - BODY: {"n": "2", "result": 1}
31 Jan 16:37:48 - BODY: {"n": "1", "result": 1}
31 Jan 16:37:48 - BODY: {"n": "3", "result": 2}
31 Jan 16:37:48 - BODY: {"n": "4", "result": 3}
31 Jan 16:37:48 - BODY: {"n": "5", "result": 5}
31 Jan 16:37:48 - BODY: {"n": "6", "result": 8}
31 Jan 16:37:48 - BODY: {"n": "7", "result": 13}
31 Jan 16:37:48 - BODY: {"n": "8", "result": 21}
31 Jan 16:37:48 - BODY: {"n": "9", "result": 34}
31 Jan 16:37:48 - BODY: {"n": "10", "result": 55}
31 Jan 16:37:48 - BODY: {"n": "20", "result": 6765}
31 Jan 16:37:59 - 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 the REST service and use the value as data in our program.

We also, inadvertently, demonstrated the issue with long-running calculations. You'll notice the requests were made from the largest to the smallest, but the results appeared in almost exactly the opposite order. Why? It's because of the processing time for each request, and the inefficient algorithm we're using. The computation time increases enough to ensure that the larger request values require enough time to reverse the order.

What happened is that `fiboclient.js` sends all its requests right away, and then each one goes into a wait for the response to arrive. Because the server is using `fibonacciAsync`, it will work on calculating all responses simultaneously. The values which are quickest to calculate are the ones which will be ready first. As the responses arrive in the client, the matching response handler fires, and in this case, the result is printed to the console. The results will arrive when they're ready and not a millisecond sooner.

Refactoring the Fibonacci application for REST

Now that we've implemented a REST-based server, we can return to the Fibonacci application, applying what you've learned to improve it. We will lift some of the code from `fiboclient.js` and transplant it into the application to do this. Change `routes/fibonacci.js` to the following code:

```
router.get('/', function(req, res, next) {
  if (req.query.fibonum) {
    var httpreq = require('http').request({
      host: "localhost",
      port: process.env.SERVERPORT,
      path: "/fibonacci/" + Math.floor(req.query.fibonum),
      method: 'GET'
    },
    httpresp => {
      httpresp.on('data', chunk => {
        var data = JSON.parse(chunk);
        res.render('fibonacci', {
          title: "Calculate Fibonacci numbers",
          fibonum: req.query.fibonum,
          fiboval: data.result
        });
      });
      httpresp.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;
```

Then, in `package.json`, change the `scripts` entry to the following:

```
"start": "SERVERPORT=3002 DEBUG=fibonacci:* node ./bin/www",
```

This way, we duplicate the same environmental configuration for running both the Fibonacci service and the Fibonacci application.

In one command window, start the server:

```
$ npm run server  
  
> fibonacci@0.0.0 server /Users/david/chap04/fibonacci  
> SERVERPORT=3002 node ./fiboserver
```

In the other window, start the application:

```
$ npm start  
  
> fibonacci@0.0.0 start /Users/david/chap04/fibonacci  
> SERVERPORT=3002 DEBUG=fibonacci:* node ./bin/www  
  
fibonacci:server Listening on port 3000 +0ms
```

Because we haven't changed the templates, the screen will look exactly as it did earlier.

We'll run into another problem with this solution. The asynchronous implementation of our inefficient Fibonacci algorithm will easily cause the Fibonacci service process to run out of memory. In the Node.js FAQ, <https://github.com/nodejs/node/wiki/FAQ>, it's suggested to use the `--max_old_space_size` flag. You'd add this in package.json as follows:

```
"server": "SERVERPORT=3002 node ./fiboserver --max_old_space_size  
5000",
```

However, the FAQ also says that if you're running into maximum memory space problems, your application should probably be refactored. This gets back to our point several pages ago, that there are several approaches to addressing performance problems. One of which is algorithmic refactoring of your application.

Why did we go to this trouble when we could just directly call `fibonacciAsync`?

We can now push the CPU load for this heavyweight calculation to a separate server. Doing so would preserve CPU capacity on the frontend server so it can attend to web browsers. 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 like this are made all the time to create mult-tier systems.

What we've demonstrated is that it's possible to implement simple mult-tier REST services in a few lines of Node.js and Express. The whole exercise gave us a chance to think about computationally intensive code in Node.js.

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 (<https://strongloop.com/node-js/loopback-framework/>): This is an offering from StrongLoop, the current sponsor of the Express project. It offers a lot of features and is, of course, built on top of Express.

Summary

You learned a lot in this chapter about Node's HTTP support and implementing web applications.

Now we can move on to implementing a more complete application, one for taking notes. We will use the Notes application for several upcoming chapters as a vehicle to explore the Express application framework, database access, deployment to cloud services or on your own server, and user authentication.

In the next chapter, we will build the basic infrastructure.

5

Your First Express Application

Now that we've got our feet wet building an Express application for Node.js, let's work on an application that performs a useful function. The application we'll build will keep a list of notes, and it will let us explore some aspects of a real application.

In this chapter, we'll only build the basic infrastructure of the application, and in the later chapters, we'll add features such as using different database engines to store the notes, user authentication, deployment on public servers, and other things.

ES-2015 Promises and Express router functions

Before we get into developing our application, we have another new ES-2015 feature to discuss – the `Promise` class. `Promise` objects are used for deferred and asynchronous computation. A `Promise` class represents an operation that hasn't completed yet, but is expected to be completed in the future.

In *Chapter 1, About Node.js*, we briefly discussed the `Promise` class while discussing Node.js's asynchronous execution model. Since Node.js was designed before the `Promise` class, the Node.js community follows a different convention for writing asynchronous code; this is the callback function. Those callback functions are called to perform any kind of deferred or asynchronous computation.

The `Promise` class approaches the same problem from a different angle. It has one large advantage in avoiding the so-called **Pyramid of Doom** that's inevitable with deeply nested callback functions.

Promises are just one way to work around this so-called *callback hell*. Over the years, several other approaches have been developed, some of which are very popular:

- <https://www.npmjs.com/package/bluebird>: This is a Promise implementation that predates ES-2015 and adds additional functionality
- <https://www.npmjs.com/package/q>: This is another Promise implementation
- <https://www.npmjs.com/package/async>: This is a collection of functions implementing many common patterns for using asynchronous functions

While the callback function approach to asynchronous programming has proved to be extremely flexible, we recognize there's a problem: the *Pyramid of Doom*. It's named after the shape the code takes after a few layers of nesting. Any multistage process can quickly escalate to code nested 15 levels deep.

Consider the following example:

```
router.get('/path/to/something', (req, res, next) => {
  doSomething(arg1, arg2, (err, data1) => {
    if (err) return next(err);
    doAnotherThing(arg3, arg2, data1, (err2, data2) => {
      if (err2) return next(err2);
      somethingCompletelyDifferent(arg1, arg42, (err3, data3) => {
        if (err3) return next(err3);
        doSomethingElse((err4, data4) => {
          if (err4) return next(err4);
          res.render('page', { data });
        });
      });
    });
  });
});
```

We're about to write some code that, in the old style of Node.js, would have looked something like the following code. But, with the `Promise` class, we can take this problem.

We generate a Promise this way:

```
exports.asyncFunction = function(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(any, results);
    });
};

```



Note that `asyncFunction` is an asynchronous function, but it does not take a callback. Instead, it returns a `Promise` object, and the asynchronous code is executed within a callback passed to the `Promise` class.

Your code must indicate the asynchronous status via the `resolve` and `reject` functions. Your caller then uses the function as follows:

```

asyncFunction(arg1, arg2)
  .then((any, results) => {
    // the operation succeeded
    // do something with the results
  })
  .catch(err => {
    // an error occurred
  });

```

The system is fluid enough that the function passed in a `.then` can return something, such as another `Promise`, and you can chain the `.then` objects together. The `.then` and `.catch` add handler functions to the `Promise` object that you can think of as a handler queue.

The handler functions are executed one at a time. The result given by one `.then` function is used as the argument list of the next `.then` function.

Promises and error handling

`Promise` objects can be in one of three states:

- **Pending:** This is the initial state, neither fulfilled nor rejected
- **Fulfilled:** This is the final state where it executed successfully and produced a result
- **Rejected:** This is the final state where execution failed

Consider this code segment similar to the one we'll use later in this chapter:

```

notes.read(req.query.key)
  .then(note => { return filterNote(note); })
  .then(note => { return swedishChefSpeak(note); })
  .then(note => {

```

```
res.render('noteview', {
  title: note ? note.title : "",
  notekey: req.query.key,
  note: note
});
})
.catch(err => { next(err); });

```

There are several places where errors can occur in this little bit of code. The `notes.read` function has several possible failure modes: the `filterNote` function might want to raise an alarm if it detects a cross-site scripting attack. The Swedish chef could be on strike. There could be a failure in `res.render` or the template being used. But we have only one way to catch and report errors. Are we missing something?

The `Promise` class automatically captures errors, sending them down the chain of operations attached to `Promise`. If the `Promise` class has an error on its hands, it skips over the `.then` functions and will instead invoke the first `.catch` function it finds. In other words, using instances of `Promise` provides a higher assurance of capturing and reporting errors. With the older convention, error reporting was trickier, and it was easy to forget to add correct error handling.

Flattening our asynchronous code

The problem being addressed is that asynchronous coding in JavaScript results in the **Pyramid of Doom**. This coding pattern looks like simple ordinary function calls, ones that would execute in linear order from top to bottom. Instead, the pyramid is a nest of code that could execute in any order depending on the whims of event processing.

To explain, let's reiterate the example Ryan Dahl gave 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 purpose—to avoid blocking the event loop with a long operation—has an excellent solution here: to defer processing the results or errors until "later". But this solution created this pyramid-shaped problem.

If the `resultSet` requires asynchronous processing, there's another layer of callbacks and so on, until this becomes a little pyramid. The problem is that results and errors land in the callback. Rather than delivering them to the next line of code, the errors and results are buried.

Promise objects help flatten the code so that it no longer takes a pyramidal shape. They also capture errors, ensuring delivery to a useful location. But those errors and results are still buried inside an anonymous function and do not get delivered to the next line of code.

The ECMAScript Committee is working on a proposal for ES-2016 to add two new keywords, `await` and `async`, that should give us the ability to land asynchronous errors and results in the place they belong: the next line of code. Refer to the official documentation at <https://tc39.github.io/ecmascript-asyncawait/> for details.

As proposed, you might be able to write the following code:

```
async UserFind(params) {  
    return await dbLookupFunction(params);  
}  
var userData = async UserFind({ email });
```

Additional tools

While the `Promise` class is a big step forward, it doesn't solve everyone's needs. Additional libraries are becoming available with extra functionality or to wrap existing libraries so they return `Promise` objects.

Here are a few libraries to help you sail through commonly encountered scenarios:

- <https://www.npmjs.com/package/promise-tools>: This extends the `Promise` class with additional processing patterns beyond `Promise.all`
- <https://github.com/jwalton/node-promise-breaker>: This assists writing libraries that optionally accept an asynchronous callback function or return a `Promise` object if no callback is provided
- <https://www.npmjs.com/package/co>: This uses ES-2015 generators and generator functions to get very close to `async/await` functionality
- <https://www.npmjs.com/package/fs-promise>: This wraps Node.js filesystem functions to return `Promise` objects

Express and the MVC paradigm

Express doesn't enforce an opinion on how you should structure the Model, View and Controller modules of your application. 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 coordinating the response to each URL. This corresponds to the Controller.

This leaves one wondering where to put code corresponding to the Model. The role of the Model is to hold application data, changing it as instructed by the Controller, and supplying data requested by View code. At a minimum, the Model code should be in separate modules from the Controller code. This is to ensure a clean separation of concerns, for example, to ease 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 for storing the notes and related data. The API of these modules will provide functions to create, read, update, or delete data items (the CRUD model) and other functions necessary for the View code to do its thing.

The CRUD model stands for Create, Read, Update and Delete (or Destroy). These are the four basic operations of persistent data storage. The Notes application is structured as a CRUD application, to demonstrate the implementation of 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 verb *destroy* rather than *delete*, because `delete` is a reserved word in JavaScript.



Creating the Notes application

Let's start creating the Notes application as before, by using the Express Generator to give us a starting point:

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

```
$ npm install express-generator@4.x
.. output
$ ./node_modules/.bin/express --ejs .
destination is not empty, continue? [y/N] y

create : .
create : ./package.json
create : ./app.js
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/index.ejs
create : ./views/error.ejs
create : ./bin
create : ./bin/www

install dependencies:
$ cd . && npm install

run the app:
$ DEBUG=notes:* npm start

$ npm install
.. output
$ ls
app.js bin node_modules package.json public routes views
$ npm uninstall express-generator
unbuild express-generator@4.13.1
```

If you wish, you can run `npm start` and view the blank application in your browser. Instead, let's move on to setting up the code.

Your first Notes model

Create a directory named `models`, as a sibling of the `views` and `routes` directories.

Then, create a file named `Note.js` in that directory, and put this code in it:

```
'use strict';

module.exports = class Note {
  constructor(key, title, body) {
    this.key = key;
    this.title = title;
    this.body = body;
  }
};
```

This defines a new class, `Note`, for use within our Notes application. This sort of object class definition is new to JavaScript with ES-2015. Previously, we would have just created an anonymous object to hold the data. However, we gain a few advantages by defining a class for our notes.

We can reliably determine whether an object is a note with the `instanceof` operator:

```
if (!(note instanceof Note)) throw new Error('Must be a Note');
```

Classes can also have functions, and you can create subclasses using an `extend` clause (`class LoveNote extends Note { ... }`). The notation is similar to other languages (such as Java) that use classical inheritance, whereas JavaScript has traditionally used prototypal inheritance. If you scratch beneath the surface of ES-2015 classes, you'll find it's still using prototypal inheritance.

Then, create a file named `notes-memory.js` in that directory, and put this code in it:

```
'use strict';

var notes = [];
const Note = require('./Note');

exports.update = exports.create = function(key, title, body) {
  return new Promise((resolve, reject) => {
    notes[key] = new Note(key, title, body);
    resolve(notes[key]);
  });
};

exports.read = function(key) {
```

```
        return new Promise((resolve, reject) => {
            if (notes[key]) resolve(notes[key]);
            else reject(`Note ${key} does not exist`);
        });
    };

exports.destroy = function(key) {
    return new Promise((resolve, reject) => {
        if (notes[key]) {
            delete notes[key];
            resolve();
        } else reject(`Note ${key} does not exist`);
    });
};

exports.keylist = function() {
    return new Promise((resolve, reject) => {
        resolve(Object.keys(notes));
    });
};

exports.count = function() {
    return new Promise((resolve, reject) => {
        resolve(notes.length);
    });
};
```

This is a simple in-memory data store that's fairly self-explanatory. It does not support any long-term data persistence. Any data stored in this model will disappear when the server is killed.

The `create` and `update` functions are being handled by the same function. At this stage of the Notes application, the code for both these functions can be exactly the same because they perform the exact same operation.

Later, when we add database support to Notes, the `create` and `update` functions will need to be different. For example, in a SQL data model, `create` would be implemented with an `INSERT INTO` command, while `update` would be implemented with an `UPDATE` command.

Creating an instance of an ES-2015 class is done like so:

```
notes[key] = new Note(key, title, body);
```

If you have experience with other languages, this will be a familiar way of constructing objects. The `constructor` method is automatically called. In our case, the arguments are simply assigned to matching fields of the object.

Each note is identified by a key, which is allowed to be any value JavaScript supports as an array index. As we'll see, the key is passed around as an identifier in forms for creating or editing notes and for retrieving notes from the data store.

Each of these functions returns a `Promise` object. In this case, this is technically unnecessary. Because the data is in memory, there's no time delay in accessing the data and little chance of error, so there's no need for deferred asynchronous execution. However, in later chapters we'll store data in databases, which definitely will involve asynchronous access and in which we will want to use Promises.

In this case, we are using Promises to detect errors in the arguments and reliably indicate such errors.

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 fixing up the home page. It should show a list of notes, and the top navigation bar should link to an *Add Note* page so that we can always add a new note.

The generated `app.js` file needs no modification for the home page. There will be modifications to this file, just not yet. To support the home page are these lines of code:

```
var routes = require('./routes/index');

...
app.use('/', routes);
```

Change `routes/index.js` to this:

```
var express = require('express');
var router = express.Router();
var notes = require('../models/notes-memory');

router.get('/', function(req, res, next) {
  notes.keylist()
    .then(keylist => {
      var keyPromises = [];
      for (var key of keylist) {
        keyPromises.push(
          notes.read(key))
```

```
        .then(note => {
          return { key: note.key, title: note.title };
        })
      );
    }
    return Promise.all(keyPromises);
})
.then(notelist => {
  res.render('index', { title: 'Notes', notelist: notelist });
})
.catch(err => { next(err); });
});

module.exports = router;
```

This gathers data about the notes that we'll be displaying on the home page.

We're using a new kind of loop supported in ES-2015:

```
for (var key of iterable) { code }
```

An **iterable** is a generalized list of items, and the `for..of` loop makes it easy to iterate over those items. Several built-in objects, such as arrays, are iterable, and it is possible to define your own iterable if needed.

The `Promise.all` function executes an array of Promises. If any Promise in the array fails, execution jumps to the `.catch` function; otherwise, it's handled by the `.then` function. The result of the Promise returned by `Promise.all` is an array containing the result of each Promise.

Because we wrote `notes.read` to be an asynchronous operation, we must use a mechanism to wait until all the `notes.read` calls are finished. We do that with `Promise.all`. We first construct an array of Promises resulting from calling `notes.read`, using a `.then` clause to construct an anonymous object containing the `key` and `title` of the note. `Promise.all` captures all those results, bundling them into the array passed as `notelist`.

What we've done is enqueue a list of asynchronous operations and neatly wait for them all to finish.

The `notelist` array is then passed into a view template we're about to write.

Change `views/index.ejs` to this:

```
<!DOCTYPE html>
<html>
<head>
```

```
<% include headerStuff %>
</head>
<body>
<% include pageHeader %>
<% if (notelist.length > 0) { %>
<ul><%
for (var note of notelist) {
  %><li><%= note.key %>:
    <a href="/notes/view?key=<%= note.key %>"><%= note.title %></a>
  </li><%
}
%></ul>
<% } %>
<% include footer %>
</body>
</html>
```

This simply steps through the array of note data and formats a simple listing. Each item links to the /noteview URL with a key parameter. We have yet to look at that code, but this URL will obviously display the note. Another thing of note is that no HTML for the list is generated if the notelist is empty.

This page follows a different strategy than the one we used in the previous chapter. The general principle is **don't repeat yourself (DRY)**, and there are multiple ways to avoid repeating code in templates. Using `top.ejs` and `bottom.ejs` was an approach that worked for a simple application, but it could be limiting. It's debatable which approach is better. At least with this approach, you have the entire HTML structure in your hands and can easily customize each page as desired.

Create `views/headerStuff.ejs` with this text:

```
<title><%= title %></title>
<link rel='stylesheet' href='/stylesheets/style.css' />
```

There's 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.

Now, create `views/pageHeader.ejs`, 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>
```

Finally, add `views/footer.ejs`, with this code inside:

```
<footer></footer>
```

For our purposes, right now, we don't have anything to use at the bottom. Of course, public websites often have administrative links or a brief site map.

In your applications, you'll need to plan how to support flexible layout choices while avoiding repeating the same code in multiple template files.

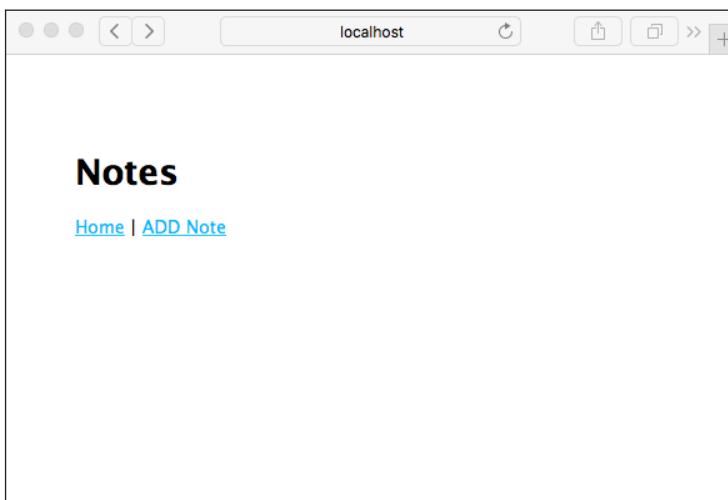
We now have enough written to run the application; let's view the home page:

```
$ npm start
```

```
> notes@0.0.0 start /Users/david/chap05/notes
> node ./bin/www
```

```
GET / 200 36.965 ms - 281
GET /stylesheets/style.css 304 6.147 ms - -
```

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.js` is performing as expected.

Adding a new note – create

Now, let's look at creating notes. Because the application doesn't have a route configured for the /notes/add URL, we must add one. To do that, we need a Controller for the notes.

In app.js, make the following changes.

Comment out these lines:

```
// var users = require('./routes/users');  
..  
// app.use('/users', users);
```

These aren't required currently, but will be in the future.

Next, we'll add similar lines for the notes Controller:

```
// var users = require('./routes/users');  
var notes = require('./routes/notes');  
..  
// app.use('/users', users);  
app.use('/notes', notes);
```

Now, we'll add a Controller module containing the notes router. Create a file named routes/notes.js, with this content:

```
'use strict';  
  
var util = require('util');  
var express = require('express');  
var router = express.Router();  
var notes = require('../models/notes-memory');  
  
// Add Note.  
router.get('/add', (req, res, next) => {  
    res.render('noteedit', {  
        title: "Add a Note",  
        docreate: true,  
        notekey: "",  
        note: undefined  
    });  
});  
  
module.exports = router;
```

The resulting /notes/add URL corresponds to the link in pageHeader.ejs.

In the `views` directory, add a template named `noteedit.ejs`, containing the following:

```
<!DOCTYPE html>
<html>
<head>
<% include headerStuff %>
</head>
<body>
<% include pageHeader %>
<form method='POST' action='/notes/save'>
<input type='hidden' name='docreate' value='<%
                                docreate ? "create" : "update"%>'>
<p>Key:
<% if (docrete) { %>
    <input type='text' name='notekey' value=' ' />
<% } else { %>
    <%= note ? notekey : "" %>
    <input type='hidden' name='notekey' value='<%
                                note ? notekey : "" %>' />
<% } %>
</p>
<p>Title: <input type='text' name='title' value='<%
                                note ? note.title : "" %>' /></p>
<br/><textarea rows=5 cols=40 name='body' ><%
                                note ? note.body : "" %></textarea>
<br/><input type='submit' value='Submit' />
</form>
<% include footer %>
</body>
</html>
```

We'll be reusing this template to support both editing notes and creating new ones.

Notice that the `note` and `notekey` objects passed to the template are empty in this case. The template detects this condition and ensures 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 of course, no `note` object exists. The template code is being written defensively, so it won't throw errors if it doesn't have a note.

This template is a FORM that will POST its data to the `/notes/save` URL. If you were to run the application at this time, it would give you an error message because no route is configured for that URL.

To support the /notes/save URL, add this to routes/notes.js:

```
// Save Note
router.post('/save', (req, res, next) => {
  var p;
  if (req.body.docrete === "create") {
    p = notes.create(req.body.notekey,
                     req.body.title, req.body.body);
  } else {
    p = notes.update(req.body.notekey,
                     req.body.title, req.body.body);
  }
  p.then(note => {
    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, it needs to detect the `docrete` flag and call the appropriate model operation.

The model returns a Promise for both `notes.create` and `notes.update`. Normally, we'd chain the `.then` and `.catch` functions off those calls, but in this case, we need to take the same action for either operation. This is a simple matter of saving the Promise into a variable and then attaching our `.then` and `.catch` functions.

Because this code is called by a POST operation, the form data is added to the `req.body` object that's created by the `bodyParser` middleware. The fields attached to `req.body` correspond directly to elements in the HTML form.

Now, we can run the application again and use the **Add a Note** form:

Add a Note

Home | ADD Note

Key:

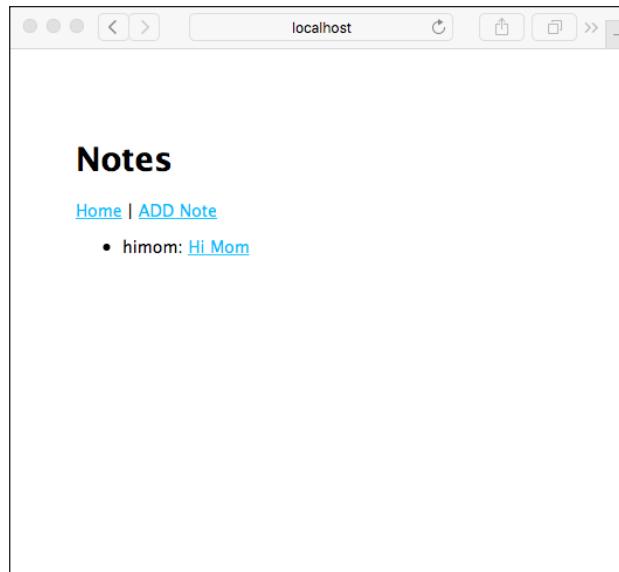
Title:

This is where we say thank you for everything

Submit

But upon clicking on the **Submit** button, we get an error message. 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 like the following screenshot on the home page:



The note is actually there; we just need to implement /notes/view.

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.

To routes/notes.js, add this router function:

```
router.get('/view', (req, res, next) => {
  notes.read(req.query.key)
    .then(note => {
      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.

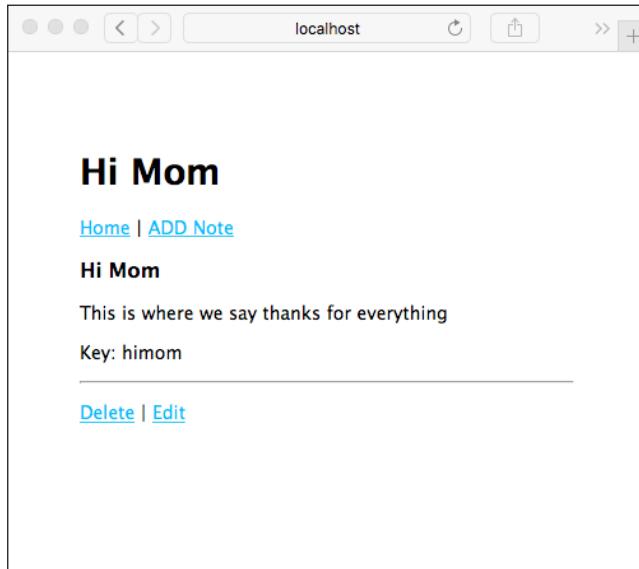
If notes.read successfully reads the note, it is rendered with the noteview template. If something goes wrong, we'll instead display an error to the user through Express.

To the views directory, add the noteview.ejs template, referenced by this code:

```
<!DOCTYPE html>
<html>
<head>
<% include headerStuff %>
</head>
<body>
<% include pageHeader %>
<h3><%= note ? note.title : "" %></h3>
<p><%= note ? note.body : "" %></p>
<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>
<% } %>
<% include footer %>
</body>
</html>
```

This is straightforward: taking data out of the note object and displaying 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.

The code for neither of these exists at the moment. But that won't stop us from going ahead and executing the application, will it?



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 you an error, because the code hasn't been implemented.

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. This will be the third of the four CRUD operations.

To routes/notes.js, add this router function:

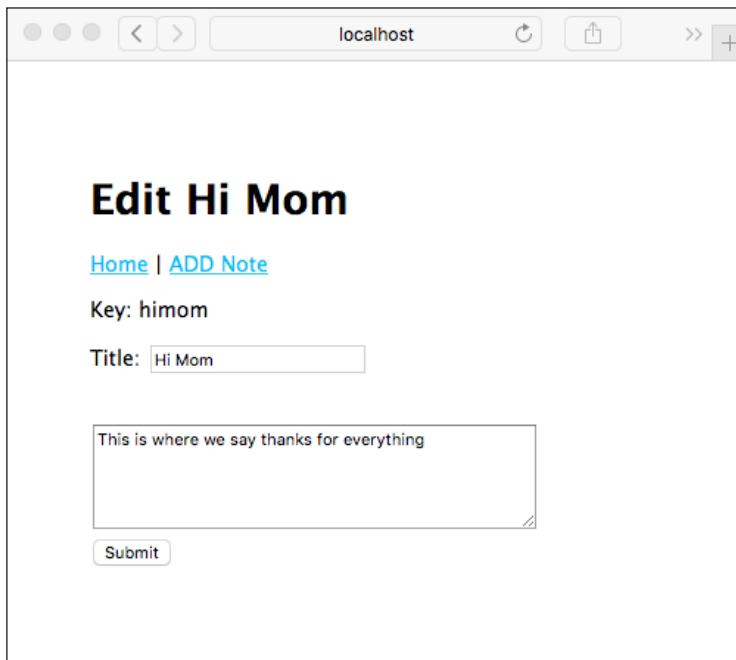
```
router.get('/edit', (req, res, next) => {
  notes.read(req.query.key)
    .then(note => {
      res.render('noteedit', {
        title: note ? ("Edit " + note.title) : "Add a Note",
        docreate: false,
        notekey: req.query.key,
```

```
        note: note
    });
})
.catch(err => { next(err); });
});
```

Of course, we're reusing the `noteedit.ejs` template, because we made sure it can be used for both create and update/edit operations.

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'll end up in the same `/notes/save` route handler shown in the preceding screenshot. It already does the right thing of calling the `notes.update` method in the model, rather than `notes.create`.

Because that's all we need do, we can go ahead and rerun the application:



Click on the **Submit** button here, and you'll come to the `/notes/view` screen and will be able to read our newly created note.

Back to the `/notes/view` screen: we've just taken care of the **Edit** link, but the **Delete** link still produces an error.

Deleting notes – destroy

Now, let's look at how to implement the /notes/destroy URL, to delete notes.

To routes/notes.js, add the following router function:

```
router.get('/destroy', (req, res, next) => {
  notes.read(req.query.key)
    .then(note => {
      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 we make a mistake. Therefore, we want to ask the user whether they're sure they want to delete that note. In this case, we retrieve the note and then render the following page, displaying a question to ensure they do want to delete the note.

To the views directory, add a notedestroy.ejs template:

```
<!DOCTYPE html>
<html>
<head>
<% include headerStuff %>
</head>
<body>
<% include pageHeader %>
<form method='POST' action='/notes/destroy/confirm'>
<input type='hidden' name='notekey' value='<%= note ? notekey : "" %>'>
<p>Delete <%= note.title %> ?</p>
<br/>
<input type='submit' value='DELETE' /> <a href="/notes/view?key=<%= notekey %>">Cancel</a>
</form>
<% include footer %>
</body>
</html>
```

This is a simple form, asking the user to confirm by clicking on the button. The **Cancel** link just sends them back to the /notes/view page.

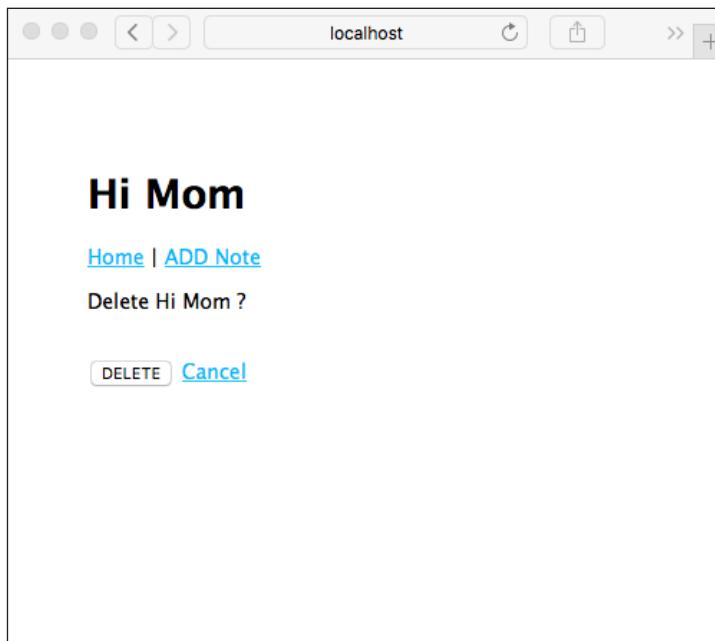
Clicking on the **Submit** button generates a POST request on the /notes/destroy/confirm URL.

That URL needs a request handler. Add this code to routes/notes.js:

```
router.post('/destroy/confirm', (req, res, next) => {
  notes.destroy(req.body.notekey)
    .then(() => { 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.

Theming your Express application

The Express team has done a decent job of making sure Express applications look okay out 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, focusing on that all-important goal of addressing the mobile market.

If you're running the Notes app using the recommended method, `npm start`, a nice log of activity is being printed in your console window. One of those is the following line of code:

```
GET /stylesheets/style.css 304 0.702 ms - -
```

This is due to this line of code that we put in `headerStuff.ejs`:

```
<link rel='stylesheet' href='/stylesheets/style.css' />
```

This file was autogenerated for us by the Express Generator at the outset and dropped inside the `public` directory. The `public` directory is managed by the Express static file server, using this line in `app.js`:

```
app.use(express.static(path.join(__dirname, 'public')));
```

Let's open `public/stylesheets/style.css` and take a look:

```
body {  
  padding: 50px;  
  font: 14px "Lucida Grande", Helvetica, Arial, sans-serif;  
}  
  
a {  
  color: #00B7FF;  
}
```

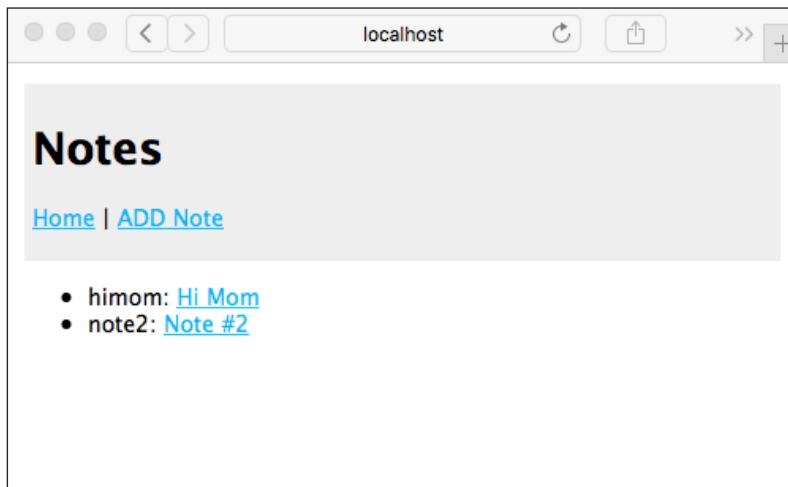
Something that leaps out is that the application content has a lot of white space at the top and left-hand sides of the screen. The reason is that `BODY` tags have the `padding: 50px` style. Changing it is a quick business.

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. It's possible to turn on cache-control headers and ETAGS generation, as you would do for a production website. Look in the online Express documentation for details.

It involves a little bit of work:

```
body {  
  padding: 5px;  
}  
  
header {  
  background: #eeeeee;  
  padding: 5px;  
}
```

As a result, we'll have this:



We're not going to win any design awards with this either, but there's the beginning of some branding and theming possibilities.

Generally speaking, the way we've structured the page templates, applying a site-wide theme is just a matter of adding appropriate code to `headerStuff.ejs`. Many of the modern theming frameworks, such as Twitter's Bootstrap, serve up CSS and JavaScript files out of a CDN server, making it incredibly easy to incorporate into a site design.

For jQuery, refer to <http://jquery.com/download/>.

Google's Hosted Libraries service provides a long list of libraries, hosted on Google's CDN infrastructure. Refer to <https://developers.google.com/speed/libraries/>.

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 this isn't a toy application, but one that is interesting enough to draw a million users a day. Serving a high load typically means adding servers, load balancers, and many other things. A core part 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 `bin/www`, you'll find these lines, which control the port being used:

```
var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);
```

Meaning it's just a matter of setting the `PORT` environment variable to the desired value. 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 these lines to the `scripts` section:

```
"scripts": {
  "start": "node ./bin/www",
  "server1": "PORT=3001 node ./bin/www",
  "server2": "PORT=3002 node ./bin/www"
},
```

The `server1` script runs on `PORT 3001`, while the `server2` script runs on `PORT 3002`. Isn't it nice to have all this documented in one place?

Then, in one command window, run this:

```
$ npm run server1

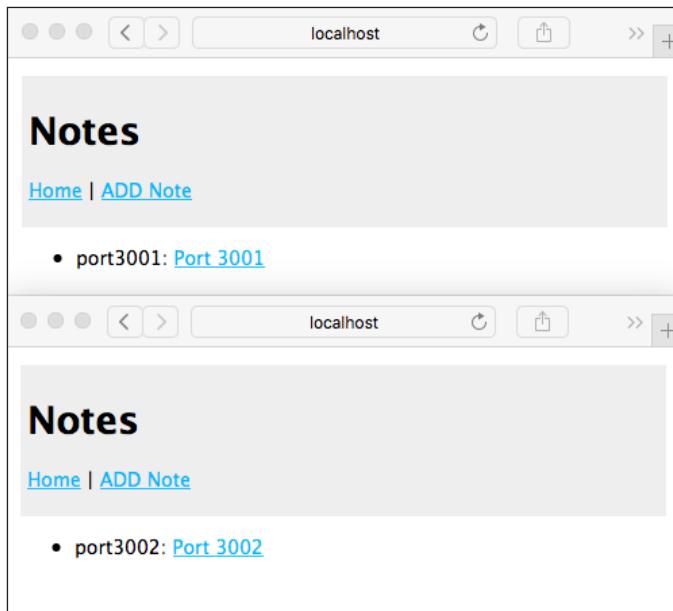
> notes@0.0.0 server1 /Users/david/chap05/notes
> PORT=3001 node ./bin/www
```

In another command window, run this:

```
$ npm run server2
```

```
> notes@0.0.0 server2 /Users/david/chap05/notes
> PORT=3002 node ./bin/www
```

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 some editing and adding notes, your two browser windows could look like the preceding screenshot. What's going on is that the two instances do not share the same data pool. Each is instead running in its own memory space. You add a note in one, and it does not show in the other screen.

Additionally, because the model code does not persist data anywhere, the notes are not saved. You might have written the greatest novel 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.

Summary

We've come a long way in this chapter.

We started with the Pyramid of Doom and how the ES-2015 Promise object can help us tame asynchronous code. We'll be using Promises all through this book.

We quickly moved to writing the foundation of a real application with Express. At the moment, it keeps its data in memory, but it has the basic functionality of what will become a note-taking application supporting real-time collaborative commenting on the notes.

In the next chapter, we'll dip our toes in 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 using their smartphone.

Afterward, we'll keep growing the capabilities of the Notes application, starting with database storage models.

6

Implementing the Mobile-First Paradigm

Now that we have a useful first Express application, we must do what's the mantra of this age of software development: make it mobile-friendly from the outset. Mobile devices, be it smart phones, tablet computers, automobile dashboards, refrigerator doors, or bathroom mirrors, are taking over the world. In 30 or 40 years, we may be fighting a war with robots who want equal rights, if *The Matrix* movies have any predictive value. In the meantime, we have gizmos to build and software to write.

Overwhelmingly, that software needs to display on a mobile device. Thankfully, modern mobile devices have real web browsers that use modern technology such as HTML5 and CSS3, and they have fast JavaScript engines.

The primary considerations in designing for mobiles are the small screen sizes, the touch-oriented interaction, that there's no mouse, and the somewhat different user interface expectations. With the **Notes** application, our user interface needs are modest, and the lack of a mouse doesn't make any difference to us. However, these are certainly big considerations for other kinds of applications.

In this chapter, we won't do much Node.js development. Instead, we'll modify the templates, editing CSS (LESS actually) files, and learning about a couple useful tools in the frontend web development arena. And, we'll dip our toes in the water of what it means to be a full stack web engineer.

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 app. 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.

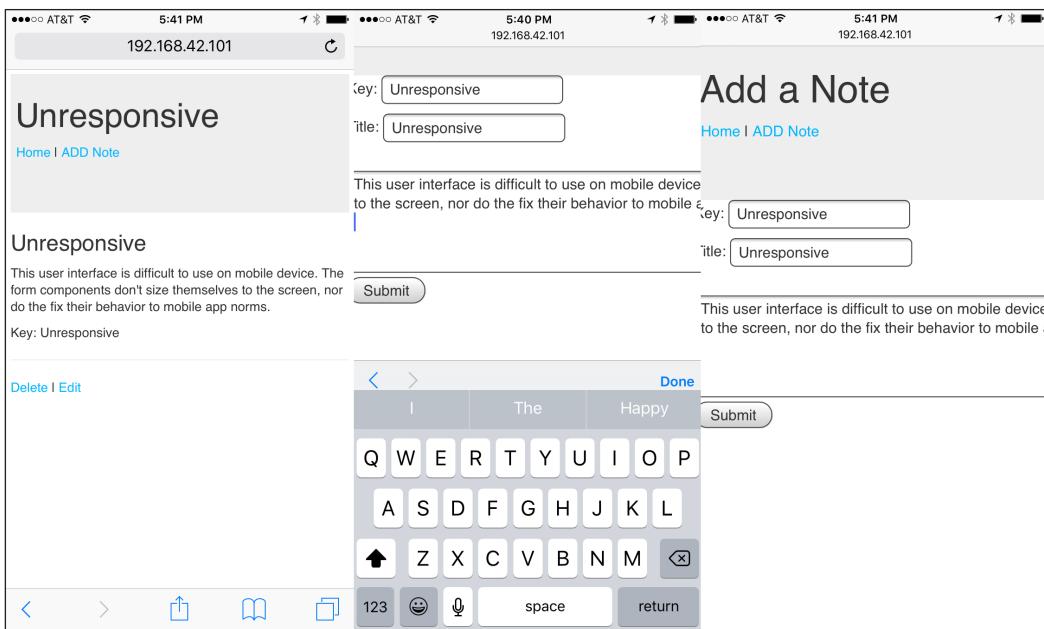
Another way to approach this is to use your desktop browser, resizing it to be very narrow. That way, you can mimic a smart phone screen's small size, showing you many of the user interface difficulties, even if it doesn't otherwise behave as a mobile app.

You may also find development-oriented browser extensions to assist. For example, in the **Chrome Developer Tools**, you can view the current webpage as if it were any of several popular mobile devices.

To see a real user interface problem on a mobile screen, edit `views/noteedit.ejs` to change this line:

```
<br/><textarea rows=5 cols=80 name='body' ><%= note ? note.body : "" %></textarea>
```

What's changed is the `cols=80` parameter. 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 the following 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. The interaction with the FORM elements works well. In general, browsing around the Notes application gives an acceptable user experience.

We can improve the Notes application all around.

Mobile-first paradigm

Mobile devices have a smaller screen, are generally touch oriented, and have different user experience expectations than a desktop computer.

To accommodate the smaller screen, we use **responsive web design** techniques. This means designing the application in such a way that it accommodates the screen size and other device attributes. The idea is crafting websites to provide an optimal viewing and interaction experience across a wide range of devices. The techniques include changing font sizes, rearranging elements on the screen, using collapsible elements that open when touched, and resizing images or videos to fit the 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 first to work well on a mobile device and then move on to devices with larger screens. It's about prioritizing mobile devices first.



The primary technique is to use media queries in the stylesheet to detect device characteristics. Within sections controlled by media queries targeting a range of devices, the CSS declarations must match the targeted devices.

It may help to consult a concrete example. The **Twenty Twelve** theme for Wordpress has a fairly 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. Refer to its source code 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 make sense for devices with larger screens.

It does this with the following two media queries:

```
@media screen and (min-width: 600px) {  
    /* For screens above 600px width */  
}  
@media screen and (min-width: 960px) {  
    /* For screens above 960px width */  
}
```

In other words, the first segment of the stylesheet configures the page layout for all devices. Next, for any browser viewport at least 600px wide, reconfigure the page to display on the larger screen. Then, for any browser viewport at least 960px wide, reconfigure it again. The stylesheet has a final media query to cover print devices.

The pixel widths given earlier are what's called a **breakpoint**. Meaning, those threshold viewport widths are where the design changes itself around. You can see these breakpoints in action easily by going to any website with a responsive design, 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 more general characteristics? The Twenty Twelve theme did fairly well on mobile devices using only two media queries. At the other end of the scale, the *CSS-Tricks* blog 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 device scenarios:

- **Small:** This includes iPhone 4.
- **Medium:** This can refer to tablet computers, or the larger smart phones.
- **Large:** This includes larger tablet computers, or the 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.

Enough with the theory, let's get back to our code.

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 that use 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.

Setting it up

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 prevents us from generating a custom theme

Instead, we'll install a local copy of Bootstrap. There are several ways to install Bootstrap. We'll use a tool, **Bower**, that's widely used in frontend engineering projects. Bower gives access to a huge repository of JavaScript packages for frontend coding. It serves a similar role for frontend engineering as npm does for Node.js. Visit <http://bower.io> for details.

We install Bower using the following command:

```
$ npm install bower@1.x --save-dev
```

The Bower website recommends a global install (with the `-g` flag) but, as mentioned earlier, we will avoid installing global dependencies.

Next, we initialize the Notes application to use Bower, and use it to install Bootstrap and jQuery:

```
$ bower init  
.. answer some questions posed by Bower  
$ bower install 'bootstrap#3.3.6' --save
```

The first step sets up a `bower.json` file to contain package information used by Bower. The purpose is similar to the `package.json` file we use for Node.js. For our purposes, we need this so that Bower can properly pull down the required dependencies.

The second step installs a specific version of Bootstrap. One principle from the Twelve Factor application model is explicitly declaring dependencies, and that applies to Bootstrap just as much as anything else. At the time of writing this, we see the Bootstrap team is preparing version 4.0, and are promising lots of changes—changes that theoretically could break the code in this book, giving us an example of that principle in action. By naming a specific version dependency, we avoid a future potential application failure if an incompatibility sneaks into third-party code.

Now, when setting up the Notes application, we will run these two commands:

```
$ npm install  
$ bower install
```

The two together set up packages from both the npm and Bower repositories. However, this isn't optimal because we need to remember to run both the commands.

In step 12 of the Twelve Factor guideline, it is recommended to automate all administrative functions. How do we ensure that `bower install` is run whenever the Notes application is installed?

Add this to the `scripts` section of `package.json`:

```
"postinstall": "bower install"
```

As it suggests, this script is run after `npm install` is executed. Therefore, we've ensured that `bower install` will execute at the appropriate time. We can verify this works, as follows:

```
$ npm install  
  
> notes@0.0.0 postinstall /Users/david/chap06/notes  
> bower install
```

Finally, let's see what this installed for us:

```
$ ls bower_components/  
bootstrap  jquery
```

We only installed Bootstrap, but because Bootstrap depends on jQuery, it was automatically installed for us.

Within each package is a `dist` directory containing the prebuilt code for the respective package:

```
$ ls bower_components/bootstrap/dist/  
css  fonts  js  
$ ls bower_components/jquery/dist/  
jquery.js  jquery.min.js  jquery.min.map
```

These files are what need to be referenced from our templates in order to use Bootstrap and jQuery.

Adding Bootstrap to application templates

On the Bootstrap website, they give a recommended HTML structure. Refer to the **Getting Started** page at <http://getbootstrap.com/getting-started/>.

There are `meta` elements to set up browser characteristics, `link` elements pointing to CSS files, and `script` elements pointing to JavaScript files. Each of these plays a role in properly initializing Bootstrap on the page, and ensuring a consistent experience across all web browsers.

To implement this, we'll need to modify `headerStuff.ejs` and `footer.ejs`.

Change `views/headerStuff.ejs` to have this content:

```
<meta charset="utf-8">  
<meta http-equiv="X-UA-Compatible" content="IE=edge">  
<meta name="viewport"  
      content="width=device-width, initial-scale=1">  
<!-- The above 3 meta tags *must* come first in the head; any other  
head content must come *after* these tags -->  
  
<title><%= title %></title>  
  
<!-- Bootstrap -->  
<link href="/vendor/bootstrap/css/bootstrap.min.css"  
      rel="stylesheet">  
  
<!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements  
     and media queries -->  
<!-- WARNING: Respond.js doesn't work if you view the page  
          via file:// -->  
<!--[if lt IE 9]>  
<script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js">  
</script>
```

```
<script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js">
</script>
<!-- [endif] -->

<link rel='stylesheet' href='/stylesheets/style.css' />
```

While most of this is from the Bootstrap site, it incorporates the previous content of `headerStuff.ejs`. Our own stylesheet is loaded following the Bootstrap stylesheet, giving us the opportunity to override anything in Bootstrap we want to change. The path to load `bootstrap.min.css` is different than recommended, and we'll go over that in a minute.

Similarly, we need to edit `views/footer.ejs` to add the code to load JavaScript files:

```
<footer></footer>

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
<script src="/vendor/jquery/jquery.min.js"></script>
<!-- Include all compiled plugins (below),
     or include individual files as needed -->
<script src="/vendor/bootstrap/js/bootstrap.min.js"></script>
```



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.

This `/vendor` URL is not currently recognized by the Notes application. To add support, edit `app.js` to add these lines:

```
app.use('/vendor/bootstrap', express.static(
  path.join(__dirname, 'bower_components', 'bootstrap', 'dist')));
app.use('/vendor/jquery', express.static(
  path.join(__dirname, 'bower_components', 'jquery', 'dist')));
```

This configures the static file server middleware to handle `/vendor/bootstrap` and `/vendor/jquery`. The content of each will come from the `dist` directory within the content of each package.

Now we have enough in place to run the application and see that Bootstrap gets loaded as pages are visited:

```
$ npm start
```

```
> notes@0.0.0 start /Users/david/chap06/notes
```

```
> node ./bin/www
```

```
GET / 200 26.419 ms - 1255
GET /stylesheets/style.css 304 5.063 ms - -
GET /vendor/jquery/jquery.min.js 304 3.700 ms - -
GET /vendor/bootstrap/js/bootstrap.min.js 304 0.566 ms - -
GET /vendor/bootstrap/css/bootstrap.min.css 304 160.282 ms - -
```

The onscreen differences are minor, but this is the proof necessary that the CSS and JavaScript files for Bootstrap are being loaded.

Mobile-first design for the Notes application

We've come a fair way already, learning about the basics of responsive design and Bootstrap, and we hooked the Bootstrap framework into our application. Now ,we're ready to launch into a redesign of the application so that it works well on mobiles and uses some Bootstrap components to make improvements.

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. It automatically scales components as the viewport changes size or shape. The method relies on `<div>` elements with classes to describe the role each `<div>` plays in the layout.

The basic layout pattern is as follows:

```
<div class="container-fluid">
  <div class="row">
    <div class="col-sm-3">Column 1 content</div>
    <div class="col-sm-9">Column 2 content</div>
  </div>
  <div class="row">
    <div class="col-sm-3">Column 1 content</div>
    <div class="col-sm-6">Column 2 content</div>
    <div class="col-sm-3">Column 3 content</div>
  </div>
</div>
```

The outermost layer is the `.container` or `.container-fluid` element. The latter is fluid, meaning it fills the width available to it. Within that are the `.row` elements for each row of the presentation, and within those are elements to contain individual columns of the presentation. The preceding example has two rows, the first having two columns, the second having three.

The column specifiers follow a naming convention that tells you which media breakpoint it applies to and the number of columns it claims:

- Using `col-xs-#` targets extra-small devices (smart phones)
- Using `col-sm-#` targets small devices
- Using `col-md-#` targets medium devices
- Using `col-lg-#` targets large devices

The number of columns claimed within a row should add up to 12 columns or less. If the columns add to more than 12, the columns beyond the 12th column wrap around to become a new row.

The grid system can do a lot more, but this gives us enough to start modifying our application. We structured each of the page templates as follows:

```
<!DOCTYPE html>
<html>
<head> ... headerStuff </head>
<body>
  ... pageHeader
  ... main content
  ... footer
</body>
</html>
```

The page content therefore has three rows: the header, the main content, and the footer.

Let's start with editing `views/index.ejs`:

```
<!DOCTYPE html>
<html>
<head><% include headerStuff %></head>
<body>
<div class="container-fluid">
<% include pageHeader %>
<% if (notelist.length > 0) { %>
<div class="row"><div class="col-xs-12">
```

```
.. show the notelist
</div></div>

<% include footer %>
</div>
</body>
</html>
```

The `<div class="container-fluid"> ... </div>` tags set up the grid container for the whole page. We add the first just after the `<body>` tag, and the second just before the closing `</body>` tag. This way, everything visible to the user is controlled by the fluid Bootstrap layout grid.

The next addition is to add the `<div>` tag for the main content row. It's one row, and the content claims the full 12 columns.

Similar changes are required in `views/noteedit.ejs` and `views/notedestroy.ejs`. We want every page to use Bootstrap for page layout and its components. Therefore, we must add Bootstrap layout `<div>` elements to every page.

In `views/noteview.ejs`, there is an additional row required for buttons to act on the note:

```
<!DOCTYPE html>
<html>
<head><% include headerStuff %></head>
<body>
<div class="container-fluid">
<% include pageHeader %>

<div class="row"><div class="col-xs-12">
<h3><%= note ? note.title : "" %></h3>
<p><%= note ? note.body : "" %></p>
<p>Key: <%= notekey %></p>
</div></div>

<% if (notekey) { %>
    <div class="row">
        <div class="btn-group col-sm-12">
            <a class="btn btn-default" href="/notes/destroy?key=<%= notekey %>" role="button">Delete</a>
            <a class="btn btn-default" href="/notes/edit?key=<%= notekey %>" role="button">Edit</a>
        </div>
    </div>
<% } %>
```

```
<% } %>

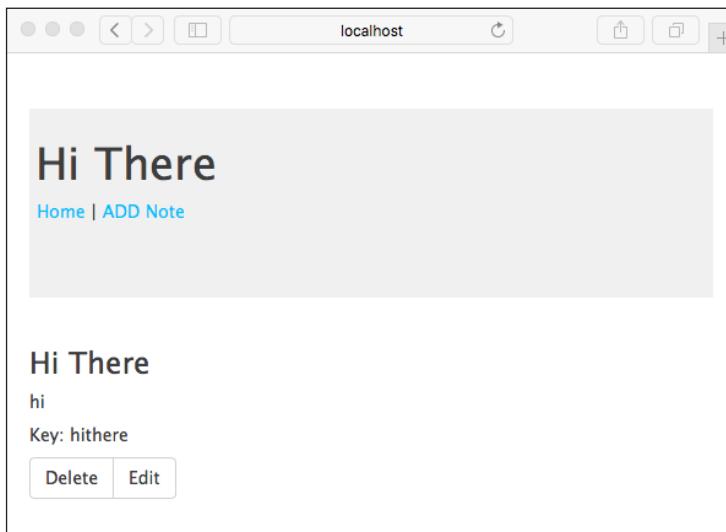
<% include footer %>
</div>
</body>
</html>
```

We have added a row for a pair of buttons to *Edit* or *Delete* the note. This row is what Bootstrap calls a **button group**, which Bootstrap will render nicely for us. Even though these are links, the `class="btn btn-default"` attribute means Bootstrap will configure these links to act like buttons; there will be a box around the button, mouseovers cause the background to change color, and a click anywhere within the box (not just on the link text) will cause the link to trigger.

There is one final change to complete coding the initial grid structure. In `views/pageHeader.ejs`, make this change:

```
<header class="page-header">
<h1><%= title %></h1>
<div class='navbar'>
<p><a href='/>Home <|> <a href='/notes/add'>ADD Note</a></p>
</div>
</header>
```

Adding `class="page-header"` informs Bootstrap this is, well, the page header.



Improving the notes list on the front page

The current home page has a 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.

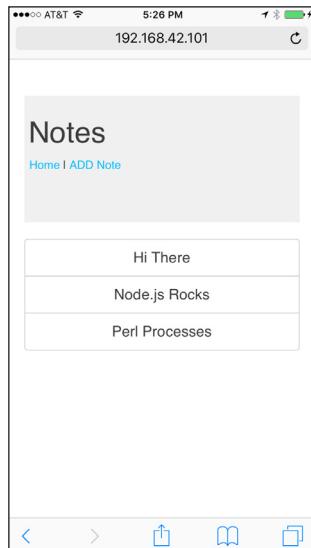
Again, edit `views/index.ejs` and make this change:

```
<div class="row"><div class="col-xs-12">
<div class="btn-group-vertical" role="group" style="width: 100%;"><%
for (var note of notelist) {
    %>
    <a class="btn btn-lg btn-block btn-default" href="/notes/
view?key=<%= note.key %>">
        <%= note.title %>
    </a><%
}
%></div>
<% } %>
</div></div>
```

The first change is to switch away from using a list and to use a vertical button group. By again making the text links look and behave like buttons, we're improving the user interface, especially its touch friendliness.

The next step is to add the classes converting the links into buttons. This time, we're making it a large button (`btn-lg`) that fills the width of its container (`btn-block`).

Finally, we eliminate showing the note key to the user. This information doesn't add anything to the user experience, does it?



This is beginning to take shape, with a decent looking home page that handles resizing very nicely, and is touch friendly. That page header though, it takes up too much space.

Breadcrumbs for the page header

The *Breadcrumb* component gives us an easy way to construct a breadcrumb trail of links. We can use that for navigation within the Notes application. Another change to make is converting the **Add New** link into a touch-friendly proper button. And, finally, while we're here, the header takes up too much vertical space, so let's tighten it up a bit.

Change `views/pageHeader.ejs` to this:

```
<header class="page-header">
<h1><%= title %></h1>

<% if (typeof hideAddNote === 'undefined') { %>
<a class="btn btn-primary" href='/notes/add'>ADD Note</a>
<% } %>

<% if (typeof breadcrumbs !== 'undefined' && breadcrumbs) { %>
<ol class="breadcrumb">
<%
for (var crumb of breadcrumbs) {
    if (crumb.active) {
        %><li class="active"><%= crumb.text %></li><%
    } else {
        %><li><a href="<%= crumb.href %>">
            <%= crumb.text %></a></li><%
    }
}
%>
</ol>
<% } %>

</header>
```

We converted the *add note* link to a button. The `btn-primary` class gives it a bold background so it'll stand out. We're also allowing this button to be hidden because it isn't necessary on every page.

Next is the breadcrumb trail. Bootstrap uses a normal list, with `class="breadcrumb"`, and arranges the list items horizontally. It even inserts a separator between each item. An item with `class="active"` gets a different presentation, and is supposed to be the final breadcrumb in the trail.

Both of these require adding parameters to the object passed to the `res.render` calls. The template looks for this data, and if it doesn't exist, there'll be nothing to output.

In `routes/index.js`, change the `res.render` call to the following:

```
res.render('index', {
  title: 'Notes',
  notelist: notelist,
  breadcrumbs: [
    { href: '/', text: 'Home' }
  ]
});
```

The `breadcrumbs` parameter will simply be an array of items describing each segment of the breadcrumb trail.

In `routes/notes.js`, we have several `res.render` calls to change.

For the `/add` route use the following code:

```
res.render('noteedit', {
  title: "Add a Note",
  docreate: true,
  notekey: "",
  note: undefined,
  breadcrumbs: [
    { href: '/', text: 'Home' },
    { active: true, text: "Add Note" }
  ],
  hideAddNote: true
});
```

Here, we want to hide the **Add Note** button. Simply add the parameter, and give it any value. The template simply looks to check whether `hideAddNote` exists, and if it does, it suppresses the code for the button. The *add note* portion of the breadcrumb trail is the active element, and therefore does not need an `href`.

For the `/view` route:

```
res.render('noteview', {
  title: note ? note.title : "",
  notekey: req.query.key,
  note: note,
  breadcrumbs: [
    { href: '/', text: 'Home' },
    { active: true, text: note.title }
  ]
});
```

For the `/edit` route, you'll need the same breadcrumbs array and also the `hideAddNote` flag:

```
res.render('noteedit', {
  title: note ? ("Edit " + note.title) : "Add a Note",
  docreate: false,
  notekey: req.query.key,
  note: note,
  hideAddNote: true,
  breadcrumbs: [
    { href: '/', text: 'Home' },
    { active: true, text: note.title }
  ]
});
```

Finally, the `/destroy` route:

```
res.render('notedestroy', {
  title: note ? note.title : "",
  notekey: req.query.key,
  note: note,
  breadcrumbs: [
    { href: '/', text: 'Home' },
    { active: true, text: 'Delete Note' }
  ]
});
```

To tighten up the page header, we need to override some default Bootstrap behavior. Add these CSS declarations to `public/stylesheets/style.css`:

```
header.page-header h1 {
  margin-top: 5px;
}

header.page-header .breadcrumb {
  margin-bottom: 5px;
}

header.page-header a.btn-primary {
  float: right;
}
```

Bootstrap puts too much margin around those items, making the page header bigger than it should be. This CSS overrides those settings, nicely tightening things up.

Browse around the application, and you'll see the breadcrumbs add some refinement.

Cleaning up the add/edit note form

The next major glaring problem is the form for adding and editing notes. Bootstrap has extensive support for making nice looking forms that work well on mobile devices.

Change the FORM in `views/noteedit.ejs` to this:

```
<div class="row"><div class="col-xs-12">
<form method='POST' action='/notes/save'>
<input type='hidden' name='docreate' value='<%= docreate ? "create" : "update"%>'>

<div class="form-group">
    <label for="notekey">Key</label>
    <% if (docreate) { %>
        <input class="form-control" type='text' id='notekey'
            name='notekey' value='' placeholder='key'/>
    <% } else { %>
        <%= note ? notekey : "" %>
        <input class="form-control" type='hidden' id='notekey'
            name='notekey' value='<%= note ? notekey : "" %>' />
    <% } %>
</div>

<div class="form-group">
    <label for="title">Title</label>
    <input type="text" class="form-control" id='title' name='title'
        placeholder="note title" value='<%= note ? note.title : "" %>'>
</div>

<div class="form-group">
    <textarea class="form-control" name='body' rows="5">
        <%= note ? note.body : "" %></textarea>
</div>

<button type="submit" class="btn btn-default">Submit</button>

</form>
</div></div>
```

There's a lot going on here, and 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:

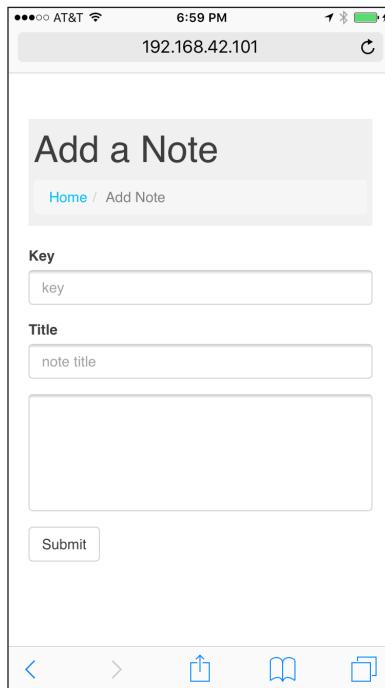
```
<div class="form-group"> . . </div>
```

Bootstrap uses this to group together elements related to one item in the FORM. It's good practice to use a `<label>` with every `<input>` so the browser can do better things than if you simply left some dangling text around. Bootstrap wants us to put both the `<label>` and the `<input>` inside the `form-group`.

The next item of note is that every form element has `class="form-control"`. Bootstrap uses this to identify the controls, for its own purposes.

This attribute, `placeholder='key'`, puts some sample text in an otherwise empty text input element. This sample text disappears as soon as the user types something. It's 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.

Building a customized Bootstrap

One of the reasons to use Bootstrap is that you can rebuild a customized version fairly easily. The stylesheets are built using LESS, which is one of the CSS preprocessors to simplify CSS development. In Bootstrap's code, one file (`less/variables.less`) contains variables used throughout the rest of Bootstrap's `.less` files. Change one variable, and it can automatically affect the rest of Bootstrap.

Earlier, we overrode a couple of Bootstrap behaviors with our custom CSS file, `public/stylesheets/style.css`. This is acceptable for changing a couple of specific things, but that technique doesn't work for large-scale changes to Bootstrap. For serious Bootstrap customizations you'll want to learn your way around this file.

Let's set ourselves up to allow rebuilding a customized Bootstrap. First, run this:

```
$ cd bower_components/bootstrap  
$ npm install  
$ npm install grunt-cli
```

These commands install dependencies to support building Bootstrap. Fair warning, this can take a while. You can test it right now by running Grunt:

```
$ grunt  
.. Grunt output from building Bootstrap
```

Grunt is a build tool written in Node.js that's widely used for frontend development projects. It's worth learning to use Grunt, but we won't do anything more with it than this. What we need to do next is modify `variables.less`.

Browse through this file, and you'll see a number of lines like these:

```
@brand-primary:      darken (#428bca, 6.5%); // #337ab7  
@brand-success:      #5cb85c;  
@brand-info:          #5bc0de;  
@brand-warning:      #f0ad4e;  
@brand-danger:        #d9534f;
```

These are variable definitions, and you'll see them used throughout the other `.less` files in this directory. Change one of these variables, rebuild Bootstrap, and it'll affect everything that uses that variable.

A quick and easy change that quickly demonstrates rebuilding Bootstrap is to change the background color:

```
@body-bg:    #fff;
```

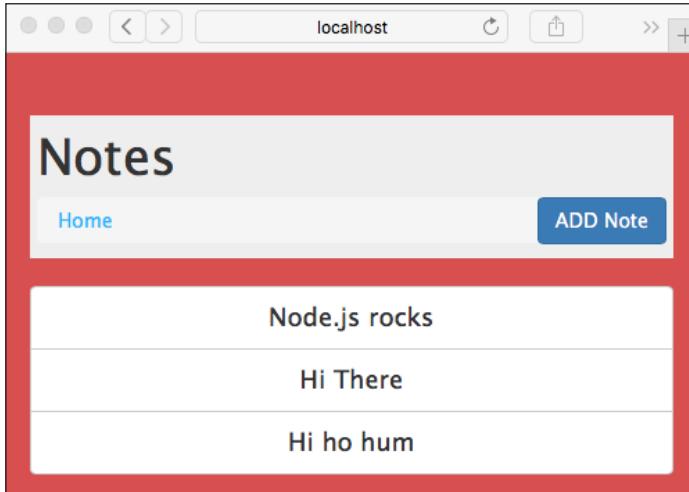
Change it to this:

```
@body-bg: @brand-danger;
```

Next, rebuild Bootstrap:

```
$ grunt
```

Then reload the application, and you'll see the following:



That's cool, we can now rework the Bootstrap color scheme any way we want. Now we need to automate this. 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 `variables.less` and change the Bootstrap theme.

Because of where `variables.less` is located, the next time we run `bower install`, any modifications will be overwritten. What we need is a way to keep our copy of `variables.less` outside the `bower_components` directory and still use it when building Bootstrap.

If you want to make deeper Bootstrap customizations, you probably instead need to maintain a fork of the Bootstrap Git repository. Instead, let's plan how we'll keep a custom `variables.less` file.

To start, simply copy the modified `variables.less` object into the root directory of the Notes application. It's this copy of the file you'll modify to customize Bootstrap, not the copy in `bower_components/bootstrap/less`.

As a good student of the Twelve Factor application recommendations, we'll automate the build process with these scripts in package.json:

```
"bootstrapsetup": "cd bower_components/bootstrap && npm install && npm install grunt-cli ",  
"buildbootstrap": "cp variables.less bower_components/bootstrap/less && cd bower_components/bootstrap && grunt"
```

This technique, command && command && command, is a way to execute a multistep process as an npm script. It will automatically fail if one of the commands fails and otherwise execute one command at a time.

The bootstrapsetup script runs the commands shown earlier to install the dependencies required to rebuild Bootstrap.

With buildbootstrap, we copy our customized variables.less object into the Bootstrap package and then use grunt for the rebuild.

Bootstrap customizers

In case all this is too complicated for you, several websites make it easy to build a customized Bootstrap:

- <http://getbootstrap.com/customize/>: The Bootstrap website offers this customizer. You enter values for all the variables.less entries, and then a button at the bottom of the page rebuilds Bootstrap and downloads it to you.
- <http://paintstrap.com/>: This site builds Boostrap themes using Adobe Kuler or COLOURlovers color schemes.
- <https://jetstrap.com/>: This site bills itself as the premium interface-building tool for Bootstrap.
- <https://wrapbootstrap.com/>: This is a Bootstrap theme marketplace.
- <http://bootswatch.com/>: This has a number of free Bootstrap themes.

In some cases, these prebuilt themes provide just bootstrap.min.css and variables.less. Using this with the Notes application means a couple of small changes to app.js.

Let's use the **Cyborg** theme from Bootswatch to explore the needed changes. You can download the theme from the website or use a command-line tool such as curl or wget as shown here:

```
$ mkdir cyborg
$ cd cyborg
$ curl http://bootswatch.com/cyborg/bootstrap.min.css -o bootstrap.min.css
$ curl http://bootswatch.com/cyborg/bootstrap.css -o bootstrap.css
$ curl http://bootswatch.com/cyborg/variables.less -o variables.less
$ curl http://bootswatch.com/cyborg/bootswatch.less -o bootswatch.less
```

Now in app.js, change the mounts for /vendor/bootstrap to the following:

```
app.use('/vendor/bootstrap/css', express.static(path.join(__dirname, 'cyborg')));
app.use('/vendor/bootstrap/fonts', express.static(path.join(__dirname, 'bower_components', 'bootstrap', 'dist', 'fonts')));
app.use('/vendor/bootstrap/js', express.static(path.join(__dirname, 'bower_components', 'bootstrap', 'dist', 'js')));
app.use('/vendor/jquery', express.static(path.join(__dirname, 'bower_components', 'jquery', 'dist')));
```

Instead of one mount for /vendor/bootstrap, we now have three mounts for each of the subdirectories. Simply make the /vendor/bootstrap/css mount point to a directory containing the CSS files you download from the theme provider.

Summary

The possibilities for using Bootstrap are endless. While we covered a lot of material, we only touched the surface, and we could have done much more to the Notes application.

You learned what the Twitter Bootstrap framework can do for us. Bootstrap's goal is to make mobile-responsive development easy. We used Bootstrap to make great improvements to the way the Notes app looks and feels. We customized Bootstrap, dipping our toes into generating a custom theme.

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 so that the Notes application can be stopped and restarted without losing our notes. In *Chapter 7, Data Storage and Retrieval*, we'll dive into using databases 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 the application works reasonably well, it doesn't store those notes anywhere for long-term storage, which means the notes are lost when you stop the server. Further, if you run multiple instances of Notes, each instance has its own set of notes; this makes it difficult to scale the application to serve lots of users.

The typical next step in such an application is to introduce a database tier. Databases provide long-term reliable storage, while enabling easy sharing of data between multiple application instances.

In this chapter, we will look at database support in Node.js in order to provide these capabilities:

- The user must see the same set of notes for any Notes instance accessed
- Reliably store notes for long-term retrieval

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'll add additional model implementations using several database engines.

Let's get started!

The first step is to duplicate the code from the previous chapter. For instance, if you were working in `chap06/notes`, duplicate that to be `chap07/notes`.

Data storage and asynchronous code

It's worth discussing the concept of asynchronous code again, since external data storage systems, by definition, require asynchronous code. The access time to retrieve data from disk, from another process, or from a database always takes enough time to require deferred execution.

Accessing in-memory data takes a few clock cycles, making it possible to directly operate on in-memory data without delays that would prevent a Node.js server from handling events. Data stored anywhere else, even in the memory space of another process, requires a delay to access that data. This is long enough to prevent the Node.js server from handling events. The rule of thumb is that asynchronous coding is not required for data retrievable within microseconds. But when the time to retrieve the data takes longer, asynchronous coding is required. Since Node.js is a single-thread system, application code cannot block the event loop.

The existing Notes application used an in-memory data store. Going by that rule of thumb, we can write the Notes model with non-asynchronous code, but instead we wrote it to use Promises for asynchronous access. That's because we thought ahead and knew we will extend the Notes application to use a database.

Logging

Before we get into databases, we have to address one of the attributes of a high-quality software system—managing logged information, including normal system activity, system errors, and debugging information. Logs are supposed to give us visibility into the behavior of the system. How much traffic is it 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. Log rotation means moving the log file out of the way every day, or so, to start with a fresh one. You should process logged data to produce reports. A high priority on screening for security vulnerabilities is a must.

The Twelve-Factor application model suggests simply sending logging information to the console, and then some other software system captures that output and directs it to a logging service. Following their advice can reduce system complexity by having fewer things that can break. In a later chapter, we'll use PM2 for that purpose.

Let's first complete a tour of information logging as it stands right now in Notes.

When we used the Express Generator to initially create the Notes app, it configured an activity logging system using Morgan:

```
var logger = require('morgan');  
..  
app.use(logger('dev'));
```

This is what prints the requests on the terminal window. Visit <https://github.com/expressjs/morgan> for more information.

Internally, Express uses the **Debug** package for debugging traces. You can turn these on using the `DEBUG` environment variable. We should try to use this package in our application code. For more information, visit <https://www.npmjs.com/package/debug>.

There may be random instances where we print stuff to `stdout` or `stderr`. This should be done using the Debug package, but third-party modules might do this and we should capture their output somehow.

Finally, the application might generate uncaught exceptions. The `uncaughtException` error needs to be captured, logged, and dealt with appropriately.

Request logging with Morgan

The Morgan package has two general areas for configuration:

- Log format
- Log location

As it stands, the code uses the `dev` format, which is described as a concise status output meant 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. It also has a nice method to generate a custom log format, if you prefer.

To change the format, simply change this line in `app.js`:

```
app.use(logger(process.env.REQUEST_LOG_FORMAT || 'dev'));
```

Then run Notes as follows:

```
$ REQUEST_LOG_FORMAT=common npm start  
  
> notes@0.0.0 start /Users/david/chap07/notes  
> node ./bin/www
```

```
::1 - - [12/Feb/2016:05:51:21 +0000] "GET / HTTP/1.1" 304 -
::1 - - [12/Feb/2016:05:51:21 +0000] "GET /vendor/bootstrap/css/
bootstrap.min.css HTTP/1.1" 304 -
::1 - - [12/Feb/2016:05:51:21 +0000] "GET /stylesheets/style.css
HTTP/1.1" 304 -
::1 - - [12/Feb/2016:05:51:21 +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.

According to the Morgan documentation, we can even pass a format string this way. If we need a custom log format, it's easy to define one.

We can declare victory on request logging and move on to debugging messages. However, let's look at logging this to a file directly. While it's possible to capture `stdout` through a separate process, Morgan is already installed in Notes and it does provide the capability to direct its output to a file.

The Morgan documentation suggests this:

```
// create a write stream (in append mode)
var accessLogStream = fs.createWriteStream(
    __dirname + '/access.log', {flags: 'a'})

// setup the logger
app.use(morgan('combined', {stream: accessLogStream}));
```

But this has a problem; it's impossible to perform log rotation without killing and restarting the server. Instead, we'll use their `file-stream-rotator` package.

First, install the package:

```
$ npm install file-stream-rotator --save
```

Then we add this code to `app.js`:

```
var FileStreamRotator = require('file-stream-rotator');
...
var accessLogStream;
if (process.env.REQUEST_LOG_FILE) {
    var logDirectory = path.dirname(process.env.REQUEST_LOG_FILE);
    fs.existsSync(logDirectory) || fs.mkdirSync(logDirectory);
    accessLogStream = FileStreamRotator.getStream({
        filename: process.env.REQUEST_LOG_FILE,
        frequency: 'daily',
```

```
    verbose: false
  });
}

..
app.use(logger(process.env.REQUEST_LOG_FORMAT || 'dev', {
  stream: accessLogStream ? accessLogStream : process.stdout
}));
```

Here, we're using an environment variable, `REQUEST_LOG_FILE`, to control whether to send the log to `stdout` or to a file. The log can go into a directory, and the code will automatically create that directory if it doesn't exist. By using the `file-stream-rotator` (<https://www.npmjs.com/package/file-stream-rotator>), we're guaranteed to have log file rotation with no extra systems required.

Debugging messages

You can generate quite a detailed trace of what Express does by running Notes this way:

```
$ DEBUG=express:* npm start
```

This is a pretty useful thing if you want to debug Express. But, we can use this in our own code as well. This is like debugging with the `console.log` statements, but without having to remember to comment out the debugging code.

It is very simple to enable debugging in a module:

```
var debug = require('debug')('module-name');
..
debug('some message');
..
debug('got file %s', fileName);
```

Capturing `stdout` and `stderr`

Important messages can be printed to `process.stdout` or `process.stderr`, which can be lost if you don't capture that output. The Twelve Factor model suggests using a system facility to capture these output streams. With Notes, we'll use PM2 for that purpose, which we'll cover in *Chapter 10, Deploying Node.js Applications*.

The `logbook` module (<https://github.com/jpillora/node-logbook>) offers some useful capabilities to not only capture `process.stdout` and `process.stderr`, but to send that output to useful places.

Uncaught exceptions

Uncaught exceptions is another area where important information can be lost. This is easy to fix in the Notes application:

```
var error = require('debug')('notes:error');

process.on('uncaughtException', function(err) {
  error("I've crashed!!! - "+(err.stack || err));
});

..

if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    // util.log(err.message);
    res.status(err.status || 500);
    error((err.status || 500) +' '+ error.message);
    res.render('error', {
      message: err.message,
      error: err
    });
  });
}

app.use(function(err, req, res, next) {
  // util.log(err.message);
  res.status(err.status || 500);
  error((err.status || 500) +' '+ error.message);
  res.render('error', {
    message: err.message,
    error: {}
  });
});
```

The Debug package has a convention we're following. For an application with several modules, all debugger objects should use the naming pattern `app-name:module-name`. In this case, we used `notes:error` that will be used for all error messages. We could also use `notes:memory-model` or `notes:mysql-model` for debugging different models.

While we were setting up the handler for uncaught exceptions, it is also a good idea to add error logging in the error handlers.

Storing notes in the filesystem

The filesystem is an often overlooked database engine. While filesystems don't have the sort of query features supported by database engines, they are a reliable place to store files. The notes schema is simple enough that the filesystem can easily serve as its data storage layer.

Let's start by adding a function to Note.js:

```
get JSON() {
    return JSON.stringify({
        key: this.key, title: this.title, body: this.body
    });
}
```

This is a **getter**, which means `note.JSON` (no parenthesis) will simply give us the JSON representation. We'll use this later for writing to JSON files.

Take a peek at the static function mentioned in the following code:

```
static fromJSON(json) {
    var data = JSON.parse(json);
    var note = new Note(data.key, data.title, data.body);
    return note;
}
```

This is a static function to aid in constructing Note objects if we have a JSON string. In other words, `fromJSON` is a factory method. The difference is that JSON is associated with an instance of the Note class, while `fromJSON` is associated with the class. That means if you have a Note instance:

```
var note = new Note("key", "title", "body");
var json = note.JSON();
```

But if you have JSON text, you can generate a Note instance as follows:

```
var note = Note.fromJSON(jsonText);
```

Now, let's create a new module, `models/notes-fs.js`, to hold the filesystem model:

```
'use strict';

const fs      = require('fs-extra');
const path   = require('path');
const util   = require('util');

const log    = require('debug')('notes:fs-model');
```

```
const error = require('debug')('notes:error');

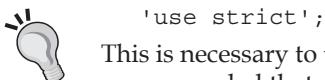
const Note = require('./Note');

function notesDir() {
  const dir = process.env.NOTES_FS_DIR || "notes-fs-data";
  return new Promise((resolve, reject) => {
    fs.ensureDir(dir, err => {
      if (err) reject(err);
      else resolve(dir);
    });
  });
}

function filePath(notesdir, key) {
  return path.join(notesdir, key + ".json");
}

function readJSON(notesdir, key) {
  const readFrom = filePath(notesdir, key);
  return new Promise((resolve, reject) => {
    fs.readFile(readFrom, 'utf8', (err, data) => {
      if (err) return reject(err);
      log('readJSON ' + data);
      resolve(Note.fromJSON(data));
    });
  });
}
```

You'll note this script starts with:



```
'use strict';
```

This is necessary to use some of the new ES-2015 features. It's highly recommended that you get into the habit of putting this line at the beginning of each file.

We're using another environment variable, NOTES_FS_DIR, to configure a directory within which to store notes. We'll have one file per note and store the note as JSON. If no environment variable is specified, we'll fall back on using notes-fs-data as the directory name.

The notesDir function will be used throughout notes-fs to ensure that the directory exists. To make this simple, we're using the fs-extra module because of the useful functions it adds to the regular fs module (<https://www.npmjs.com/package/fs-extra>). In this case, the fs.ensureDir function checks if the named directory structure exists. If not, it creates the named directory path for us.

Because we're adding another dependency:

```
$ npm install fs-extra --save
```

The filename for each data file is the key with .json appended. That gives one limitation that filenames cannot contain the / character, so we test for that using the following code:

```
exports.update = exports.create = function(key, title, body) {
    return notesDir().then(notesdir => {
        if (key.indexOf('/') >= 0)
            throw new Error(`key ${key} cannot contain '/'`);
        var note = new Note(key, title, body);
        const writeTo = filePath(notesdir, key);
        const writeJSON = note.JSON;
        log('WRITE ' + writeTo + ' ' + writeJSON);
        return new Promise((resolve, reject) => {
            fs.writeFile(writeTo, writeJSON, 'utf8', err => {
                if (err) reject(err);
                else resolve(note);
            });
        });
    });
};
```

To flatten the code a little bit, we receive a `Promise` object from the `notesDir` function, and then chain our code in a `.then` clause.

Another advantage is we can simply throw an error in the natural way. The thrown error will naturally bubble out to a `.catch` handler.

Since the final step, `fs.writeFile`, is an asynchronous function, we had to wrap it with a `Promise`.

Otherwise, this is straightforward. The `filePath` function gives us a pathname to the data file, and then we use `fs.writeFile` to write the JSON to that file.

Where's the error handling code? We do check for a couple of errors, but where are they handled? Aren't there potential unchecked errors here? We have `Promise` objects that are generated, but no `.catch` function to capture any errors.

Notice that the `notesDir` function returns a `Promise` object, and this function is returning that `Promise`. We already know that `Promise` objects capture thrown exceptions, channeling them to a `.catch` function. For example, if one of these variables was `undefined` for some reason, triggering an unexpected exception, the `Promise` object will capture that exception.

As for *where's the error handler*, consider that these functions will be called from router functions in routes/index.js or routes/notes.js. The missing .catch function is located in those router functions. That .catch function causes the error to get passed through the Express middleware chain until it lands in an error handler. That error handler is in app.js, and it causes an error page to be shown:

```
exports.read = function(key) {
    return notesDir().then(notesdir => {
        return readJSON(notesdir, key).then(thenote => {
            log('READ ' + notesdir + '/' + key
                + ' ' + util.inspect(thenote));
            return thenote;
        });
    });
};
```

Using readJSON, read the file from the disk. It already generates the Note object, so all we have to do is hand the note to the promise chain:

```
exports.destroy = function(key) {
    return notesDir().then(notesdir => {
        return new Promise((resolve, reject) => {
            fs.unlink(filePath(notesdir, key), err => {
                if (err) reject(err);
                else resolve();
            });
        });
    });
};
```

The fs.unlink function deletes our file. Because this module uses the filesystem, deleting the file is all that's necessary to delete the note object.

Because it's an asynchronous function, we have to wrap fs.unlink within a Promise. You'll be doing a lot of this, so get used to this pattern:

```
exports.keylist = function() {
    return notesDir().then(notesdir => {
        return new Promise((resolve, reject) => {
            fs.readdir(notesdir, (err, filez) => {
                if (err) return reject(err);
                if (!filez) filez = [];
                resolve({ notesdir, filez });
            });
    });
};
```

```

        })
      .then(data => {
        log('keylist dir=' + data.notesdir
          +' files=' + util.inspect(data.filez));
        var thenotes = data.filez.map(fname => {
          var key = path.basename(fname, '.json');
          log('About to READ ' + key);
          return readJSON(data.notesdir, key).then(thenote => {
            return thenote.key;
          });
        });
        return Promise.all(thenotes);
      });
    };
}

```

The contract for `keylist` is to return a Promise that will resolve to an array of keys for existing note objects. Since they're stored as individual files in the `notesdir`, we have to read every file in that directory to retrieve its key.

`Array.map` constructs a new array from an existing array, using the return values of the provided function. We're using `Array.map` to construct an array containing `Promise` objects that read each file and resolve the promise with the `key`. We then use `Promise.all` to process that array of Promises:

```

exports.count = function() {
  return notesDir().then(notesdir => {
    return new Promise((resolve, reject) => {
      fs.readdir(notesdir, (err, filez) => {
        if (err) return reject(err);
        resolve(filez.length);
      });
    });
  });
}

```

Counting the number of notes is simply a matter of counting the number of files in `notesdir`.

We're almost ready to do this, but there's a few changes required in the router modules.

Let's start with `routes/index.js`, the router module for the home page:

```

'use strict';
..
var path = require('path');
var notes = require(process.env.NOTES_MODEL

```

```
? path.join('..', process.env.NOTES_MODEL)
: '../models/notes-memory';

const log = require('debug')('notes:router-home');
const error = require('debug')('notes:error');
```

We're changing how Notes model is being loaded.

We want to be able to use different Notes models at different times, making it easy to switch between them. This is the most flexible method to set an environment variable naming the path to the desired module.

NOTES_MODEL must be set to the path from the application's root directory. In this case, that would be `models/notes-fs`. This code then generates a relative pathname, because the router code and model code are in sibling directories, as shown.

In `routes/notes.js`, we make the same change:

```
var path = require('path');
var notes = require(process.env.NOTES_MODEL
    ? path.join('..', process.env.NOTES_MODEL)
    : '../models/notes-memory');

const log = require('debug')('notes:router-notes');
const error = require('debug')('notes:error');
```

In `package.json`, add this to the `scripts` section:

```
"start-fs": "NOTES_MODEL=models/notes-fs node ./bin/www",
```

 When you put these entries in `package.json`, make sure that you use 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 up an error message.

In `app.js`, you might want to readjust the theme to the default Bootstrap look:

```
// app.use('/vendor/bootstrap/css',
//   express.static(path.join(__dirname, 'cyborg')));
app.use('/vendor/bootstrap/css', express.static(path.join(
  __dirname, 'bower_components', 'bootstrap', 'dist', 'css')));
app.use('/vendor/bootstrap/fonts', express.static(path.join(
  __dirname, 'bower_components', 'bootstrap', 'dist', 'fonts')));
app.use('/vendor/bootstrap/js', express.static(path.join(
  __dirname, 'bower_components', 'bootstrap', 'dist', 'js')));
```

With this code in place, we can now run the notes application as follows:

```
$ DEBUG=notes:* npm run start-fs

> notes@0.0.0 start-fs /Users/david/chap07/notes
> NOTES_MODEL=models/notes-fs node ./bin/www

notes:server Listening on port 3000 +0ms
notes:fs-model keylist dir=notes-fs-data files=[ ] +4s
```

Then 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 simply by 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 at the command line using regular text editors like `vi`. You can now start multiple servers on different ports and see exactly the same notes:

```
"server1": "NOTES_MODEL=models/notes-fs PORT=3001 node ./bin/www",
"server2": "NOTES_MODEL=models/notes-fs PORT=3002 node ./bin/www",
```

Then you start `server1` and `server2` in separate command windows as we did in *Chapter 5, Your First Express Application*. Then, visit the two servers in separate browser windows, and you will see that both browser windows show the same 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 therefore 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 this isn't allowed.

Storing notes with the LevelUP data store

To get started with actual databases, let's look at an extremely lightweight, small-footprint database engine: **LevelUP**. This is a Node.js-friendly wrapper around the LevelDB engine developed by Google, which is normally used in web browsers for local data persistence. It is a non-indexed, NoSQL data store designed originally for use in browsers. The Node.js module, LevelUP, uses the LevelDB API, and supports multiple backends, including LevelDOWN that integrates the C++ LevelDB database into Node.js.

Visit <https://www.npmjs.com/package/levelup> for information on the module.

To install the database engine, run this command:

```
$ npm install levelup@1.x leveldown@1.x --save
```

Then, start creating the module `models/notes-levelup.js`:

```
'use strict';

const util      = require('util');
const levelup   = require('levelup');

const log       = require('debug')('notes:levelup-model');
const error     = require('debug')('notes:error');

const Note     = require('./Note');

var db; // store the database connection here

function connectDB() {
  return new Promise((resolve, reject) => {
    if (db) return resolve(db);
    levelup(process.env.LEVELUP_DB_LOCATION
      || 'notes.levelup', {
        createIfMissing: true,
        valueEncoding: "json"
      },
      (err, _db) => {
        if (err) return reject(err);
        db = _db;
        resolve();
      });
  });
}
```

When connected to a LevelUP database, the `levelup` module gives us a `db` object through which to interact with the database. We're storing that object as a global within the module for easy usage. If the `db` object is set, we can just return it immediately. Otherwise, we open the database, telling `createIfMissing` to go ahead and create the database if needed:

```
exports.update = exports.create = function(key, title, body) {
  return connectDB().then(() => {
    var note = new Note(key, title, body);
```

```
        return new Promise((resolve, reject) => {
            db.put(key, note, err => {
                if (err) reject(err);
                else resolve(note);
            });
        });
    );
};


```

Calling `db.put` either creates a new database entry, or replaces an existing one. Therefore, both `exports.update` and `exports.create` are set to be the same function.

Under the covers, LevelUP automatically encodes our object as JSON, and then reconstitutes an object when reading from the database. Therefore, we don't need to convert to and from JSON ourselves:

```
exports.read = function(key) {
    return connectDB().then(() => {
        return new Promise((resolve, reject) => {
            db.get(key, (err, note) => {
                if (err) reject(err);
                else resolve(new Note(note.key,
                    note.title, note.body));
            });
        });
    });
};


```

LevelUP makes it easy: just call `db.get` and it retrieves the data. While the data is reconstituted as an object, we need to ensure it's converted into a `Note` object:

```
exports.destroy = function(key) {
    return connectDB().then(() => {
        return new Promise((resolve, reject) => {
            db.del(key, err => {
                if (err) reject(err);
                else resolve();
            });
        });
    });
};


```

The `db.destroy` function deletes a record from the database:

```
exports.keylist = function() {
    return connectDB().then(() => {
        var keyz = [];
        return new Promise((resolve, reject) => {
            db.createReadStream()
                .on('data', data => keyz.push(data.key))
                .on('error', err => reject(err))
                .on('end', () => resolve(keyz));
        });
    });
};

exports.count = function() {
    return connectDB().then(() => {
        var total = 0;
        return new Promise((resolve, reject) => {
            db.createReadStream()
                .on('data', data => total++)
                .on('error', err => reject(err))
                .on('end', () => resolve(total));
        });
    });
};
```

Where other operations were easy with `LevelUP`, these two are not straightforward. The `LevelUP` database doesn't give us an easy way to retrieve the keys or count the items in the database. `LevelUP` does not provide any indexing or a query function.

Instead we have to read the items one at a time. It provides an `EventEmitter` style interface to make the code more readable, at least.

Now, because of our preparation, we can go straight to using the `LevelUP` model. First, add this to `package.json` in the `scripts` section:

```
"start-levelup": "NOTES_MODEL=models/notes-levelup node ./bin/www",
```

Then, you can run the Notes application:

```
$ DEBUG=notes:* npm run start-levelup
> notes@0.0.0 start /Users/david/chap07/notes
> node ./bin/www
```

```
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 and see that everything works correctly.

LevelUP does not support simultaneous access to a database from multiple instances. Therefore, you won't be able to use the multiple Notes application scenario. You will, however, be able to stop and restart the application at will without losing any notes.

Storing notes in SQL with SQLite3

To get started with actual databases, let's see how to use SQL from Node.js. First, we'll use SQLite3, a lightweight, simple-to-set-up database engine eminently suitable for many applications.

To learn about that database engine, visit <http://www.sqlite.org/>.

To learn 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 first step is to install the module:

```
$ npm install sqlite3@3.x --save
```

SQLite3 database scheme

Next, we need to make sure our database is configured. We're using this SQL table definition for the schema (save this as `models/schema-sqlite3.sql`):

```
CREATE TABLE IF NOT EXISTS notes (
    notekey VARCHAR(255) ,
    title    VARCHAR(255) ,
    author   VARCHAR(255) ,
    body     TEXT
);
```

How do we initialize this schema before writing some code? One way is to ensure the `sqlite3` package is installed through your operating system package management system, such as using `apt-get` on Ubuntu/Debian, Macports on Mac OS X, and so on. Once it's installed, you can run this command:

```
$ sqlite3 chap07.sqlite3
SQLite version 3.10.2 2016-01-20 15:27:19
Enter ".help" for usage hints.
```

```
sqlite> CREATE TABLE IF NOT EXISTS notes (
...>     notekey VARCHAR(255),
...>     title   VARCHAR(255),
...>     author  VARCHAR(255),
...>     body    TEXT
...> );
sqlite> .schema notes
CREATE TABLE notes (
    notekey VARCHAR(255),
    title   VARCHAR(255),
    author  VARCHAR(255),
    body    TEXT
);
sqlite> ^D
```

While we can do that, the Twelve Factor application model says we must automate any administrative processes like this. To that end, we should instead write a little script to run a SQL operation on SQLite3 and use that 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 it:

```
$ npm run sqlite3-setup

> notes@0.0.0 sqlite3-setup /Users/david/chap07/notes
> sqlite3 chap07.sqlite3 --init models/schema-sqlite3.sql

-- Loading resources from models/schema-sqlite3.sql

SQLite version 3.10.2 2016-01-20 15:27:19
Enter ".help" for usage hints.
sqlite> .schema notes
CREATE TABLE notes (
    notekey VARCHAR(255),
    title   VARCHAR(255),
```

```
author  VARCHAR(255),  
body    TEXT  
);  
sqlite> ^D
```

We could have written a small Node.js script to do this, and it's easy to do so. However, by using the tools provided by the package, we have less code to maintain in our own project.

SQLite3 model code

Now, we can write code to use this database in the Notes application.

Create the file `models/notes-sqlite3.js`:

```
'use strict';  
  
const util      = require('util');  
const sqlite3 = require('sqlite3');  
  
const log       = require('debug')('notes:sqlite3-model');  
const error     = require('debug')('notes:error');  
  
const Note      = require('./Note');  
  
sqlite3.verbose();  
var db; // store the database connection here  
  
exports.connectDB = function() {  
    return new Promise((resolve, reject) => {  
        if (db) return resolve(db);  
        var dbfile = process.env.SQLITE_FILE || "notes.sqlite3";  
        db = new sqlite3.Database(dbfile,  
            sqlite3.OPEN_READWRITE | sqlite3.OPEN_CREATE,  
            err => {  
                if (err) reject(err);  
                else {  
                    log('Opened SQLite3 database ' + dbfile);  
                    resolve(db);  
                }  
            }  
        );  
    });  
};
```

This serves the same purpose as the `connectDB` function in `notes-levelup.js`: to manage the database connection. If the database is not open, it'll go ahead and do so, and even make sure the database file is created (if it doesn't exist). Whether or not the database is already open, the database connection will be passed to the next stage of the Promise.

You could, if needed, insert a `.then` function to automatically create the table if it does not exist. The SQL for that would be:

```
CREATE TABLE IF NOT EXISTS notes ( .. schema definition );
```

This way there would be no need for a separate step to initialize the table, but it would carry the overhead of executing this SQL every time the database is opened:

```
exports.create = function(key, title, body) {
    return exports.connectDB().then(() => {
        var note = new Note(key, title, body);
        return new Promise((resolve, reject) => {
            db.run("INSERT INTO notes ( notekey, title, body ) "+
                "VALUES ( ?, ?, ? );",
                [ key, title, body ], err => {
                    if (err) reject(err);
                    else {
                        log('CREATE '+ util.inspect(note));
                        resolve(note);
                    }
                })
        });
    });
};

exports.update = function(key, title, body) {
    return exports.connectDB().then(() => {
        var note = new Note(key, title, body);
        return new Promise((resolve, reject) => {
            db.run("UPDATE notes "+
                "SET title = ?, body = ? "+
                "WHERE notekey = ?",
                [ title, body, key ], err => {
                    if (err) reject(err);
                    else {
                        log('UPDATE '+ util.inspect(note));
                        resolve(note);
                    }
                })
        });
    });
};
```

These are our `create` and `update` functions. As promised, we are now justified in defining the Notes model to have separate functions for `create` and `update` operations. The SQL statement for each is different.

Calling `db.run` executes a SQL query, giving us the opportunity to insert parameters into the query string.

The `sqlite3` module uses a parameter substitution paradigm that's common in SQL programming interfaces. The programmer puts the SQL query into a string, and then places a question mark in each place it's desired to insert a value into the query string. Each question mark in the query string has to match with 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 preventing SQL injection attacks.

The `db.run` function simply runs the SQL query it is given, and does not retrieve any data:

```
exports.read = function(key) {
    return exports.connectDB().then(() => {
        return new Promise((resolve, reject) => {
            db.get("SELECT * FROM notes WHERE notekey = ?",
                [ key ], (err, row) => {
                    if (err) reject(err);
                    else {
                        var note = new Note(row.notekey,
                            row.title, row.body);
                        log('READ ' + util.inspect(note));
                        resolve(note);
                    }
                });
        });
    });
};
```

To retrieve data using the `sqlite3` module, you use the `db.get`, `db.all`, or `db.each` functions. The `db.get` function used here returns the first row of the result set. The `db.all` function returns all rows of the result set at once, which can be a problem for available memory if the result set is large. The `db.each` function retrieves one row at a time, while still allowing processing of the entire result set.

For the Notes application, using `db.get` to retrieve a note is sufficient because there is only one note per `notekey`. Therefore, our `SELECT` query will return at most one row anyway. But what if your application will see multiple rows in the result set? We'll see what to do about that in a minute.

By the way, this `read` function has a bug in it. See if you can spot the error. We'll read more about this in *Chapter 11, Unit Testing*, when our testing efforts uncover the bug.

```
exports.destroy = function(key) {
    return exports.connectDB().then(() => {
        return new Promise((resolve, reject) => {
            db.run("DELETE FROM notes WHERE notekey = ?",
                [ key ], err => {
                    if (err) reject(err);
                    else {
                        log('DESTROY ' + key);
                        resolve();
                    }
                })
        });
    });
};
```

To destroy a note, we simply execute the `DELETE FROM` statement:

```
exports.keylist = function() {
    return exports.connectDB().then(() => {
        return new Promise((resolve, reject) => {
            var keyz = [];
            db.each("SELECT notekey FROM notes",
                (err, row) => {
                    if (err) reject(err);
                    else keyz.push(row.notekey);
                },
                (err, num) => {
                    if (err) reject(err);
                    else resolve(keyz);
                })
        });
    });
};
```

The `db.each` function conveniently iterates over each row of the result set, calling the first function on each. This avoids the memory footprint of loading the whole result set into memory at once. Instead, you process it one item at a time.

There are two callback functions for the `db.each` function. The first is executed for each row of the result set, and here we use that to construct the array of keys. The second is called when the result set is finished, and here we use it to resolve the Promise to return the key array. The code is as follows:

```

exports.count = function() {
  return exports.connectDB().then(() => {
    return new Promise((resolve, reject) => {
      db.get("select count(notekey) as count from notes",
        (err, row) => {
          if (err) return reject(err);
          resolve(row.count);
        });
    });
  });
};

```

We can simply use SQL to count the number of notes for us. In this case, `db.get` returns a row with a single column, `count`, which is the value we want to return.

Running Notes with SQLite3

Finally, we're ready to run the Notes application with SQLite3. Add the following code to the `scripts` section of `package.json`:

```
"start-sqlite3": "SQLITE_FILE=chap07.sqlite3 NOTES_MODEL=models/notes-
sqlite3 node ./bin/www",
```

Run the Notes application:

```
$ DEBUG=notes:* npm run start-sqlite3
```

```

> notes@0.0.0 start-sqlite3 /Users/david/chap07/notes
> SQLITE_FILE=chap07.sqlite3 NOTES_MODEL=models/notes-sqlite3 node ./bin/
www

notes:server Listening on port 3000 +0ms
notes:sqlite3-model Opened SQLite3 database chap07.sqlite3 +5s

```

You can now browse the application at `http://localhost:3000`, and run it through its paces as before.

Because SQLite3 supports simultaneous access from multiple instances, you can run the multiserver example by adding this to the `scripts` section of `package.json`:

```

"server1-sqlite3": "SQLITE_FILE=chap07.sqlite3 NOTES_MODEL=models/
notes-sqlite3 PORT=3001 node ./bin/www",
"server2-sqlite3": "SQLITE_FILE=chap07.sqlite3 NOTES_MODEL=models/
notes-sqlite3 PORT=3002 node ./bin/www",

```

Then, run each of these in separate command windows, 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 to inspect the database:

```
$ sqlite3 chap07.sqlite3
SQLite version 3.10.2 2016-01-20 15:27:19
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
```

Storing notes the ORM way with Sequelize

There are several popular SQL database engines, such as PostgreSQL, MySQL (<https://www.npmjs.com/package/mysql>), and MariaDB (<https://www.npmjs.com/package/mariasql>). These modules are similar in nature to the `sqlite3` module we just used. The programmer is close to the 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 using the database to store objects rather than rows of a database table? **Object Relation Mapping (ORM)** systems provide such a higher-level interface and even offer the ability to use the data model with several databases.

The **Sequelize** module (<http://www.sequelizejs.com/>) is Promise-based, offers strong, well-developed ORM features, and can connect with SQLite3, MySQL, and PostgreSQL, MariaDB, and MSSQL. Because Sequelize is Promise-based, it will fit naturally with the application code we're writing.

A prerequisite is to have access to a suitable database server. Most web hosting providers offer MySQL or PostgreSQL as part of the service.

Before we start on the code, let's install two modules:

```
$ npm install sequelize@3.x --save
$ npm install js-yaml@3.x --save
```

The first obviously installs the Sequelize package. The second, `js-yaml`, is installed so that we can store the Sequelize connection configuration as a YAML file.

Perhaps the best place to learn about YAML is its Wikipedia page at <https://en.wikipedia.org/wiki/YAML>. It is a human-readable *data serialization language*, which simply means YAML is an easy-to-use text file format to describe data objects.

Sequelize model for the Notes application

Let's create a new file `models/notes-sequelize.js`:

```
'use strict';

const util      = require('util');
const fs        = require('fs-extra');
const jsyaml   = require('js-yaml');
const Sequelize = require("sequelize");
const log       = require('debug')('notes:sequelize-model');
const error     = require('debug')('notes:error');
const Note      = require('./Note');

exports.connectDB = function() {

    var SQNote;
    var sequlz;

    if (SQNote) return SQNote.sync();

    return new Promise((resolve, reject) => {
        fs.readFile(process.env.SEQUELIZE_CONNECT, 'utf8',
        (err, data) => {
            if (err) reject(err);
            else resolve(data);
        });
    })
    .then(yamltext => {
        return jsyaml.safeLoad(yamltext, 'utf8');
    })
    .then(params => {
        sequlz = new Sequelize(params.dbname,
                               params.username, params.password,
                               params.params);
        SQNote = sequlz.define('Note', {
            notekey: { type: Sequelize.STRING,
                       primaryKey: true, unique: true },
            title: Sequelize.STRING,
            body: Sequelize.TEXT
        });
        return SQNote.sync();
    });
};
```

This is a little bigger than our other `connectDB` functions. We're using a three-stage process, which first reads in a configuration file for Sequelize connection parameters, parses that with the YAML parser, uses Sequelize to open the database, and finally sets up the schema.

The Sequelize connection parameters are stored in a YAML file we specify in the `SEQUELIZE_CONNECT` environment variable. We'll go over the format of this file later.

The line `new Sequelize(..)` opens the database connection. The parameters obviously contain any needed database name, username, password, and other options required to connect with the database.

The line `sequelize.define` is where we define the database schema. Instead of defining the schema as the SQL command to create the database table, we're giving a high-level description of the fields and their characteristics. Sequelize maps the object attributes into columns in tables. Here, we defined a simple object with the same attributes in the other Notes model implementations. Finally, the `SQNote.sync()` function ensures the table is set up.

We're telling Sequelize to call this schema Note, but we're using a variable `SQNote` to refer to that schema. That's because we already defined Note as a class to represent notes. To avoid a clash of names, we'll keep using the Note class, and use `SQNote` to interact with Sequelize about the notes stored in the database.

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/>

```
exports.create = function(key, title, body) {
    return exports.connectDB()
        .then(SQNote => {
            return SQNote.create({
                notekey: key,
                title: title,
                body: body
            });
        });
};

exports.update = function(key, title, body) {
    return exports.connectDB()
        .then(SQNote => {
            return SQNote.find({ where: { notekey: key } })
        });
};
```

```

        .then(note => {
            if (!note) {
                throw new Error("No note found for key " + key);
            } else {
                return note.updateAttributes({
                    title: title,
                    body: body
                });
            }
        });
    );
}
;

```

There are several ways to create a new object instance in Sequelize, the simplest of which is to call an object's `create` function (in this case, `SQNote.create`). This function collapses together two other functions, `build` (to create the object) and `save` (to write it to the database).

Updating an object instance is a little different. First, we must retrieve its entry from the database using the `find` operation. It uses an object specifying the query to perform. The `find` operation retrieves one instance, whereas the `findAll` operation retrieves all matching instances.

For documentation on Sequelize queries, visit <http://docs.sequelizejs.com/en/latest/docs/querying/>.

You'll notice that `SQNote.find` returns a Promise, and we've attached a `.then` function to that Promise. There are instances where the `.then` function is called, but the `note` object is empty. We account for that case by throwing an error saying no note was found.

Once the instance is found, we can update its values simply with the `updateAttributes` function.

```

exports.read = function(key) {
    return exports.connectDB()
        .then(SQNote => {
            return SQNote.find({ where: { notekey: key } })
                .then(note => {
                    if (!note) {
                        throw new Error("No note found for " + key);
                    } else {
                        return new Note(note.notekey,
                            note.title, note.body);
                    }
                });
        });
}
;
```

To read a note, we use the `find` operation again. There is the possibility of an empty result, and we have to throw an error to match.

The contract for this function is to return a `Note` object. That means taking the fields retrieved using Sequelize and using that to create a `Note` object.

```
exports.destroy = function(key) {
  return exports.connectDB()
    .then(SQNote => {
      return SQNote.find({ where: { notekey: key } })
        .then(note => {
          return note.destroy();
        });
    });
};
```

To destroy a note, we must use the `find` operation to retrieve its instance, and then call its `destroy()` method:

```
exports.keylist = function() {
  return exports.connectDB().then(SQNote => {
    return SQNote.findAll({ attributes: [ 'notekey' ] })
      .then(notes => {
        return notes.map(note => note.notekey);
      });
  });
};
```

Because the `keylist` function needs to act on all `Note` objects, we use the `findAll` operation. We query for the `notekey` attribute on all notes. Because we're given an array of objects with a field named `notekey`, to match our contract we use the `.map` function to convert this into an array of the note keys.

This uses the ES-2015 arrow function to its fullest to collapse this code to pure succinctness. When written as `note => note.notekey`, an arrow function behaves as if it had been written this way:

```
(note) => {
  return note.notekey;
}
```

Remember that the `map` function constructs a new array from the return value of the function passed to `map`. Thus, we're taking the array of `Note` objects and converting it into an array of keys, on one line of code.

```
exports.count = function() {
    return exports.connectDB().then(SQNote => {
        return SQNote.count().then(count => {
            log('COUNT ' + count);
            return count;
        });
    });
};
```

For the `count` function, we can just use the `count()` method to calculate the needed result.

Configuring a Sequelize database connection

Sequelize supports several SQL database engines. Earlier, we said we needed a YAML file to store the database access credentials, but didn't give specifics of that file. Let's do so.

Let's first use SQLite3, because no further setup is required. After that, we'll get adventurous and reconfigure our Sequelize module to use MySQL. For connection configuration, create a file named `models/sequelize-sqlite.yaml` containing the following code:

```
dbname: notes
username:
password:
params:
  dialect: sqlite
  storage: notes-sequelize.sqlite3
```

If you refer back to the Sequelize constructor above, it requires `dbname`, `username`, and `password` as connection credentials, and the `params` object gives additional parameters. That's exactly what we have in the YAML file.

The `dialect` field tells Sequelize what kind of database to use. For a SQLite database, the database filename is given in the `storage` field.

How do you switch to another database server? Create a new file, such as `models/sequelize-mysql.yaml`, containing something like the following code:

```
dbname: notes
username: .. user name
password: .. password
params:
  host: localhost
  port: 3306
  dialect: mysql
```

This is straightforward. Set the database `dialect` and other connection information, and you're good to go.

You will need to install the base MySQL driver so that Sequelize can use MySQL:

```
$ npm install mysql@2.x --save
```

Add an entry into the `scripts` section of `package.json`:

```
"start-sequelize-mysql": "SEQUELIZE_CONNECT=models/sequelize-mysql.yaml NOTES_MODEL=models/notes-sequelize node ./bin/www",
```

No other change is needed; Sequelize takes care of any difference between the SQL database engines.

Running with Sequelize against 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.

Running the Notes application with Sequelize

Now we can get ready to run the Notes application using Sequelize. We can run this against both SQLite3 and MySQL, but let's start with SQLite. Add this entry to the `scripts` entry in `package.json`:

```
"start-sequelize": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_MODEL=models/notes-sequelize node ./bin/www"
```

Then run it as follows:

```
$ DEBUG=notes:* npm run start-sequelize

> notes@0.0.0 start-sequelize /Users/david/chap07/notes
> SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_MODEL=models/
notes-sequelize node ./bin/www

notes:server Listening on port 3000 +0ms
```

As before, the application looks exactly the same because we've not changed the View templates or CSS files. Put it through its paces and everything should work.

With Sequelize, multiple Notes application instances is as simple as adding these lines to the `scripts` section of `package.json`, then starting both instances as before:

```
"server1-sequelize": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml  
NOTES_MODEL=models/notes-sequelize PORT=3001 node ./bin/www",  
"server2-sequelize": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml  
NOTES_MODEL=models/notes-sequelize PORT=3001 node ./bin/www",
```

You will be able to start both instances, use separate browser windows to visit both instances, and see that they show the same set of notes.

Now that we have run Notes using Sequelize with SQLite, we need to also run it with MySQL. We already created the `models/sequelize-mysql.yaml` configuration file and the `scripts` entry in `package.json`. Before we can proceed, you need to provision a MySQL server.

The specific method you follow depends on your preferences. For example, MySQL is available through all package management systems, making it easy to install an instance on your laptop. You can head to <http://dev.mysql.com/downloads/mysql/> and follow the official installation instructions. Since MariaDB is API compatible with MySQL, you can use it instead of MySQL if you prefer.

If you prefer to use PostgreSQL, it's possible to do so with a small adjustment to the instructions, since Sequelize supports that database as well. Simply create a configuration file, `models/sequelize-postgresql.yaml`, by duplicating the MySQL config. You then use `postgres` as the dialect, rather than `mysql`. The Node.js database driver is installed using the following command:

```
$ npm install --save pg pg-hstore
```

Then you add a line to the `scripts` section of `package.json`:

```
"start-sequelize-postgres": "SEQUELIZE_CONNECT=models/sequelize-  
postgres.yaml NOTES_MODEL=models/notes-sequelize node ./bin/www",
```

These steps to use PostgreSQL have not been tested, but the Sequelize website says it should work. It also says a similar set of steps will get you running on MySQL.

Once you have a MySQL or PostgreSQL instance installed, you can run Notes against the database as follows:

```
$ DEBUG=notes:* npm run start-sequelize-mysql
```

```
> notes@0.0.0 start-sequelize-mysql /Users/david/chap07/notes
```

```
> SEQUELIZE_CONNECT=models/sequelize-mysql.yaml NOTES_MODEL=models/notes-sequelize node ./bin/www
```

```
notes:server Listening on port 3000 +0ms
```

With Notes running on your preferred database server, such as MySQL, put it through its paces and verify that Notes works.

Storing notes in MongoDB

MongoDB is widely used with Node.js applications, if only because of the popular MEAN acronym: MongoDB (or MySQL), Express, Angular, and Node.js. Because MongoDB uses JavaScript as its native scripting language, it'll be instantly familiar to JavaScript programmers.

MongoDB is one of the leading NoSQL databases. 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: <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/>).

You will need a running MongoDB instance. **Mongolab** (<https://mongolab.com/>), **compose.io** (<https://www.compose.io/>), and **ScaleGrid.io** (<https://scalegrid.io/>) offer hosted MongoDB services.

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, and the MongoDB website has instructions (<https://docs.mongodb.org/manual/installation/>).

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 one command window, run:

```
$ mkdir data  
$ mongod --dbpath data
```

In another command window, you can test it like this:

```
$ mongo
MongoDB shell version: 3.0.8
connecting to: test
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
  http://docs.mongodb.org/
Questions? Try the support group
  http://groups.google.com/group/mongodb-user
> db.foo.save({ a: 1});
writeResult({ "nInserted" : 1 })
> db.foo.find();
{ "_id" : ObjectId("56c0c98673f65b7988a96a77"), "a" : 1 }
>
bye
```

This saves a *document* in the collection named `foo`. The second command finds all documents in `foo`, printing them out for you. The `_id` field is added by MongoDB and serves as a document identifier. This is useful for testing and debugging. For a real deployment, your MongoDB server must be properly on a server. See the MongoDB documentation for these instructions.

MongoDB model for the Notes application

Now that you've proved you have a working MongoDB server, let's get to work.

Installing the Node.js driver is as simple as running the following command:

```
$ npm install mongodb@2.x --save
```

While Mongoose is a good ORM for MongoDB, the API provided by the `mongodb` driver is very good and easy to use. Because the MongoDB shell uses JavaScript for its command set, the `mongodb` driver implements a nearly identical API.

Now create a new file, `models/notes-mongodb.js`:

```
'use strict';

const util      = require('util');
const mongodb = require("mongodb");
```

```
const MongoClient = require('mongodb').MongoClient;
const log        = require('debug')('notes:mongodb-model');
const error      = require('debug')('notes:error');
const Note       = require('./Note');

var db;

exports.connectDB = function() {
    return new Promise((resolve, reject) => {
        if (db) return resolve(db);
        // Connection URL
        var url = process.env.MONGO_URL;
        // Use connect method to connect to the Server
        MongoClient.connect(url, (err, _db) => {
            if (err) return reject(err);
            db = _db;
            resolve(_db);
        });
    });
};
```

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/chap07`.

This opens the database connection, unless the database connection is already open. In either case, the connection object is passed down the Promise chain:

```
exports.create = function(key, title, body) {
    return exports.connectDB()
        .then(db => {
            var note = new Note(key, title, body);
            var collection = db.collection('notes');
            return collection.insertOne({
                notekey: key, title: title, body: body
            }).then(result => { return note; });
        });
};

exports.update = function(key, title, body) {
    return exports.connectDB()
        .then(db => {
            var note = new Note(key, title, body);
            var collection = db.collection('notes');
            return collection.updateOne({ notekey: key },
```

```

        { $set: { title: title, body: body } })
    .then(result => { return note; } );
}
};

```

MongoDB stores all documents in *collections*. Each document is a BSON object (a binary form of JSON) under the covers. A collection is a group of related documents, and a collection 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 that 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. See its documentation at <http://mongodb.github.io/node-mongodb-native/2.1/api/Collection.html>.

As the method name implies, `insertOne` inserts one document into the collection. Likewise, 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.

You'll see that we're using these methods as if they return a Promise. Indeed, that is what they return. The `mongodb` driver supports both callbacks and Promises. If you call a method and provide a callback function, it will provide results and errors through the callback. But if you leave out the callback function, it returns a Promise that you can use to receive the results and errors.

Further documentation can be found at the following links:

- **Insert:** <https://docs.mongodb.org/getting-started/node/insert/>
- **Update:** <https://docs.mongodb.org/getting-started/node/update/>

Next, let's look at reading a note from MongoDB:

```

exports.read = function(key) {
    return exports.connectDB()
    .then(db => {
        var collection = db.collection('notes');
        // Find some documents
        return collection.findOne({ notekey: key })
        .then(doc => {
            var note = new Note(doc.notekey, doc.title, doc.body);
            return note;
        });
    });
};

```

The `mongodb` driver supports several variants of `find` operations. In this case, the Notes application ensures there is exactly one document matching a given key. Therefore, we can use the `findOne` method. As the name implies, `findOne` will return the first matching document.

The argument to `findOne` is a query descriptor. This simple query looks for documents whose `notekey` field matches the requested `key`. An empty query will, of course, match all documents in the collection. You can match against other fields in a similar way, and the query descriptor can do much more. For documentation on queries, visit <https://docs.mongodb.org/getting-started/node/query/>.

The `insertOne` method we used earlier also took the same kind of query descriptor.

In order to satisfy the contract for this function, we create a `Note` object and then return it to the Promise chain. Hence, we create a `Note` using the data retrieved from the database.

```
exports.destroy = function(key) {
    return exports.connectDB()
        .then(db => {
            var collection = db.collection('notes');
            return collection.findOneAndDelete({ notekey: key });
        });
};
```

One of the `find` variants is `findOneAndDelete`. As the name implies, it finds one document matching the query descriptor, and then deletes that document.

The status, including possible errors, is returned to the Promise chain.

```
exports.keylist = function() {
    return exports.connectDB()
        .then(db => {
            var collection = db.collection('notes');
            return new Promise((resolve, reject) => {
                var keyz = [];
                collection.find({}).forEach(
                    note => { keyz.push(note.notekey); },
                    err => {
                        if (err) reject(err);
                        else resolve(keyz);
                    }
                );
            });
        });
};
```

Here, we're using the base `find` operation and giving it an empty query so that it matches every document. What we're to return is an array containing the `notekey` for every document.

All of the `find` operations return a `Cursor` object. The documentation can be found at <http://mongodb.github.io/node-mongodb-native/2.1/api/Cursor.html>.

The `Cursor` object is, as the name implies, a pointer into a result set from a query. It has a number of useful functions related to operating on a result set. For example, you can skip the first few items in the results, or limit the size of the result set, or perform the `filter` and `map` operations.

The `Cursor.forEach` method takes two callback functions. The first is called on every element in the result set. In this case, we can use that to record just the `notekey` into an array. The second callback is called after all elements in the result set have been processed. We use this to indicate success or failure, and to pass the `keyz` array to the Promise chain.

```
exports.count = function() {
    return exports.connectDB()
        .then(db => {
            var collection = db.collection('notes');
            return new Promise((resolve, reject) => {
                collection.count({}, (err, count) => {
                    if (err) reject(err);
                    else resolve(count);
                });
            });
        });
};
```

The `count` method takes a query descriptor and, as the name implies, counts the number of matching documents.

Running the Notes application with MongoDB

Now that we have our MongoDB model, we can get ready to run Notes with it.

By now you know the drill; add this to the `scripts` section of `package.json`:

```
"start-mongodb": "MONGO_URL=mongodb://localhost/chap07 NOTES_"
MODEL=models/notes-mongodb node ./bin/www",
```

The `MONGO_URL` environment variable is the URL to connect with your MongoDB database.

You can start the Notes application as follows:

```
$ DEBUG=notes:* npm run start-mongodb

> notes@0.0.0 start-mongodb /Users/david/chap07/notes
> MONGO_URL=mongodb://localhost/chap07 NOTES_MODEL=models/notes-mongodb
node ./bin/www

notes:server Listening on port 3000 +0ms
```

You can browse the application at <http://localhost:3000> and put it through its paces. You can kill and restart the application, and your notes will still be there.

Add this to the `scripts` section of `package.json`:

```
"server1-mongodb": "MONGO_URL=mongodb://localhost/chap07 NOTES_
MODEL=models/notes-mongodb PORT=3001 node ./bin/www",
"server2-mongodb": "MONGO_URL=mongodb://localhost/chap07 NOTES_
MODEL=models/notes-mongodb PORT=3002 node ./bin/www",
```

You will be able to start two instances of the Notes application, and see that both share the same set of notes.

Summary

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 the surface of options for accessing databases and data storage engines from Node.js.

By abstracting the model implementations correctly, we were able to easily switch data storage engines while not changing the rest of the application.

By focusing the model code on the purpose of storing data, both the models and the application should be easier to test. The application can be tested with a mock data module that provides known predictable notes that can be checked predictably. We'll look at this in more depth in *Chapter 11, Unit Testing*.

In the next chapter, we'll focus on deploying our application for real use by real people.

8

Multiuser Authentication the Microservice Way

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. In this chapter, we'll discuss the following three aspects of this phase:

- Creating a microservice to store user profile/authentication data.
- User authentication 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.



It seems 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. HTTP is a stateless protocol, and a web application cannot tell much about one HTTP request versus another. Because HTTP is stateless, HTTP requests do not natively know whether the user driving the web browser is logged in, the user's identity, or even whether the HTTP request was initiated by a human being.

The typical method for user authentication is to send a cookie to the browser containing a token to carry user identity. The cookie needs to contain data identifying the browser and whether that browser is logged in. The cookie will then be sent with every request, letting the application track which user account is associated with the browser.

With Express, the best way to do this is with the `express-session` middleware. It stores data as a cookie and looks for that data on every browser request. It is easy to configure, but is not a complete solution for user authentication. There are several add-on modules that handle user authentication, and some even support authenticating users against third-party websites such as Facebook or Twitter.

One package appears to be leading the pack in user authentication: **Passport** (<http://passportjs.org/>). It supports a long list of services against which to authenticate, making it easy to develop a website that lets users sign up with credentials from another website, for example, Twitter. Another, `express-authentication` (<https://www.npmjs.com/package/express-authentication>), bills itself as the opinionated alternative to Passport.

We will use Passport to authenticate users against both a locally stored user credentials database and using OAuth2 to authenticate against a Twitter account. We'll also take this as an opportunity to explore REST-based microservice implementation with Node.js. The idea is to simulate keeping user identity data in a secured server. When storing user credentials, it's important to use extra caution. You don't want your application to be associated with the latest massive leak of user data, do you?

Let's get started!

The first thing to do is duplicate the code used for the previous chapter. For example, if you kept that code in `chap07/notes`, 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 it would be easy to accomplish, is this what we would do in a real-world production application?

Consider the following considerations:

- It's important to keep user identity data under tight security
- The Notes application may be a small portion of a larger website; therefore, user authentication must be shared between multiple applications
- The microservice approach—dividing the application into small chunks, each deployed as separate services—is a popular architecture choice that increases software development flexibility

It's claimed that the microservice approach is easier to manage than when developing a monolithic large application. By dividing the total system into small services, each service can be developed by a small team, whose engineers can fully grow their service, with each able to proceed on his/her own time schedule independent of every other service in the overall system. Microservice implementations can even be completely replaced as long as the replacement service implements the same API.

Microservices are, of course, not a panacea, meaning we shouldn't try to force-fit every application into the microservice box. By analogy, microservices are like the Unix philosophy of small tools that do one thing well, that 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? A key trade-off of the microservice architecture is that one loses the advantages gained by tight integration of components.

Which path you choose is up to you. While we'll develop the user authentication service as a microservice, it could easily be directly integrated into the Notes application.

While we can use Express to develop REST services, there is another framework that is specialized to developing REST services – **Restify** (<http://restify.com/>).

The server will require two modules: one using Restify to implement the REST interface and the other acting as a data model using Sequelize to store user data objects in a SQL database. To test the service, we'll write a couple of simple scripts for administering user information in the database. Then we'll set about modifying the Notes application to use this service to access user information, while using Passport to handle authentication.

The first step is creating 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 run the following commands:

```
$ cd users
$ npm init
.. answer questions
.. name - user-auth-server
$ npm install debug@2.x --save
$ npm install js-yaml@3.x --save
```

```
$ npm install restify@4.x --save
$ npm install sequelize@3.x --save
$ npm install mysql@2.x sqlite3@3.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/mysql/sqlite3` for database access.

User information model

We'll be storing the user information using a Sequelize-based model in an SQL database. As we go through this, ponder a question: should we integrate the database code directly into the REST API implementation? Doing so would reduce the user information microservice to one module. By separating the REST service from the data storage model, we have the freedom to adopt other data storage systems besides Sequelize/SQL.

Maybe you want to skip creating the microservice and instead integrate the user authentication service into Notes? The first step is to place the module we're about to write into the `users` directory.

Create a new file named `users-sequelize.js` in `users` containing:

```
'use strict';

const Sequelize = require("sequelize");
const jsyaml = require('js-yaml');
const fs = require('fs');
const util = require('util');
const log = require('debug')('users:model-users');
const error = require('debug')('users:error');

var SQUser;
var sequlz;

exports.connectDB = function() {

    if (SQUser) return SQUser.sync();

    return new Promise((resolve, reject) => {
        fs.readFile(process.env.SEQUELIZE_CONNECT, 'utf8',
        (err, data) => {
            if (err) reject(err);
            else resolve(data);
        });
    });
}
```

```
        });
    })
    .then(yamltext => {
      return jsyaml.safeLoad(yamltext, 'utf8');
    })
    .then(params => {
      if (!sequlz) sequlz = new Sequelize(
        params.dbname, params.username, params.password,
        params.params);

      if (!SQUUser) SQUUser = sequlz.define('User', {
        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)
      });
      return SQUUser.sync();
    });
  );
};
```

Like with our Sequelize-based model for Notes, we use a YAML file to store connection configuration. We're even using the same environment variable, `SEQUELIZE_CONNECT`.

One question at this point is: where do we store the data? By using Sequelize, we have our pick of SQL databases to choose from.

It's tempting to simplify the overall system by using the same database instance to store notes and user information, and to use Sequelize for both.

But we've chosen to simulate a secured server for user data. Theoretically, the ideal implementation at least has user data in a separate database. A highly secure application deployment might put the user information service on completely separate servers, perhaps in a physically isolated data center, with carefully configured firewalls, and there might even be armed guards at the door. We won't go quite that far, but at least we can simulate the conditions by maintaining a clean separation.

The user profile schema shown here is derived from the normalized profile provided by Passport; refer to <http://www.passportjs.org/docs/profile> for more information. Passport will harmonize information given by third-party services into a single object definition.

To simplify our code, we're simply using the schema defined by Passport.

As with the models used in the Notes application, we're using `Promise` objects and will return a `Promise` object to the REST interface module:

```
exports.create = function(username, password, provider, familyName,
givenName, middleName, emails, photos) {
    return exports.connectDB().then(SQUser => {
        return SQUser.create({
            username: username,
            password: password,
            provider: provider,
            familyName: familyName,
            givenName: givenName,
            middleName: middleName,
            emails: JSON.stringify(emails),
            photos: JSON.stringify(photos)
        });
    });
};

exports.update = function(username, password, provider, familyName,
givenName, middleName, emails, photos) {
    return exports.find(username).then(user => {
        return user ? user.updateAttributes({
            password: password,
            provider: provider,
            familyName: familyName,
            givenName: givenName,
            middleName: middleName,
            emails: JSON.stringify(emails),
            photos: JSON.stringify(photos)
        }) : undefined;
    });
};
```

Our `create` and `update` functions take user information and either add a new record or update an existing record.

This treatment of the `emails` and `photos` fields is not quite correct. Both fields are provided by Passport as arrays. The correct modeling would be to add another two database tables: one for `emails` and the other for `photos`. Serializing these fields with JSON is a simplification:

```
exports.find = function(username) {
    log('find ' + username);
    return exports.connectDB().then(SQUser => {
```

```

        return SQUser.find({ where: { username: username } });
    })
    .then(user => user ? exports.sanitizedUser(user) : undefined);
}

```

This lets us look up a user information record, and we return a sanitized version of that data.

 Remember that Sequelize returns a `Promise` object, which is resolved, meaning that either the `.then` or `.catch` function is called when the data is retrieved. The `.then` function will receive a `SQUser` object containing the field data, or else `null` if no user was found.

Because we're segregating the user data from the rest of the Notes application, we want to return a sanitized object rather than the actual `SQUser` object. What if there was some information leakage because we simply sent the `SQUser` object back to the caller? The `sanitizedUser` function, shown later, creates an anonymous object with exactly the fields we want exposed to the other modules.

```

exports.destroy = function(username) {
    return exports.connectDB().then(SQUser => {
        return SQUser.find({ where: { username: username } })
    })
    .then(user => {
        if (!user) throw new Error('Did not find requested ' +
            + username + ' to delete');
        user.destroy();
        return;
    });
}

```

This lets us support deleting user information. We do this as we did for the Notes Sequelize model, by first finding the user object then calling its `destroy` method.

```

exports.userPasswordCheck = function(username, password) {
    return exports.connectDB().then(SQUser => {
        return SQUser.find({ where: { username: username } })
    })
    .then(user => {
        if (!user) {
            return { check: false, username: username,
                message: "Could not find user" };
        } else if (user.username === username
            && user.password === password) {

```

```
        return { check: true, username: user.username } ;
    } else {
        return { check: false, username: username,
                 message: "Incorrect password" } ;
    }
});
```

```
};
```

And this lets us support checking user passwords.

The three conditions to handle are as follows:

- Whether there's no such user
- Whether the passwords matched
- Whether they did not match

The object we return lets the caller distinguish between those cases. The `check` field indicates whether to allow this user to be logged in. If `check` is false, there's some reason to deny their request to log in, and the `message` is what should be displayed to the user.

```
exports.findOrCreate = function(profile) {
    return exports.find(profile.id).then(user => {
        if (user) return user;
        return exports.create(profile.id, profile.password,
                             profile.provider, profile.familyName,
                             profile.givenName, profile.middleName,
                             profile.emails, profile.photos);
    });
};
```

This combines two actions in one function: first to verify if the named user exists and, if not, to create that user. Primarily, this will be used while authenticating against third-party services.

```
exports.listUsers = function() {
    return exports.connectDB()
        .then(SQUser => SQUser.findAll({}) )
        .then(userlist => userlist.map(user =>
            exports.sanitizedUser(user)))
        .catch(err => console.error(err));
};
```

List the existing users. The first step is using `findAll` to give us the list of the users as an array of `SQUser` objects. Then we sanitize that list so we don't expose any data we don't want exposed:

```
exports.sanitizedUser = function(user) {
  return {
    id: user.username,
    username: user.username,
    provider: user.provider,
    familyName: user.familyName,
    givenName: user.givenName,
    middleName: user.middleName,
    emails: user.emails,
    photos: user.photos
  };
};
```

This is our utility function to ensure we expose a carefully controlled set of information to the caller. With this service, we're emulating a secured user information service that's walled off from other applications. Therefore, as we said earlier, this function returns an anonymous sanitized object where we know exactly what's in the object.

A REST server for user information

We are building our way toward integrating user information and authentication into the Notes application. The next step is to wrap the user data model we just created into a REST server. After that, we'll create a couple of scripts so that we can add some users, perform other administrative tasks, and generally verify the service works. Finally, we'll extend the Notes application with login and logout support.

In the `package.json` file, change the `main` tag to the following line of code:

```
"main": "user-server.js",
```

And then create a file named `user-server.js` containing the following code:

```
'use strict';

const restify = require('restify');
const util = require('util');
const log = require('debug')('users:server');
const error = require('debug')('users:error');
const usersModel = require('./users-sequelize');
```

```
var server = restify.createServer({
  name: "User-Auth-Service",
  version: "0.0.1"
});

server.use(restify.authorizationParser());
server.use(check);
server.use(restify.queryParser());
server.use(restify.bodyParser({
  mapParams: true
}));
```

This part initializes a Restify server application.

The `createServer` method can take a long list of configuration options. These two may be useful identifying information.

As with Express applications, the `server.use` calls initialize what Express would call middleware functions, but which Restify calls handler functions. These are callback functions whose API is `function (req, res, next)`. Like for Express, this is the request and response objects, and `next` is a function which, when called, proceeds execution to the next handler function.

Unlike Express, every handler function must call the `next` function. In order to tell Restify to stop processing through handlers, the `next` function must be called as `next (false)`. Calling `next` with an error object also causes execution to end, and the error is sent back to the requestor.

The handler functions listed here do two things: authorize requests and handle parsing parameters from both the URL and the post request body. The `authorizationParser` function looks for HTTP basic auth headers. The `check` function is shown later and emulates the idea of an API token to control access.

Refer to <http://restify.com/#bundled-plugins> for more information on the built-in handlers available in Restify.

```
// Create a user record
server.post('/create-user', (req, res, next) => {
  usersModel.create(req.params.username, req.params.password,
    req.params.provider, req.params.lastName,
    req.params.givenName, req.params.middleName,
    req.params.emails,   req.params.photos)
  .then(result => {
    log('created ' + util.inspect(result));
    res.send(result);
  })
});
```

```

        next(false);
    })
    .catch(err => { res.send(500, err); error(err.stack); next(false);
  });
});

```

As for Express, the `server.VERB` functions let us define the handlers for specific HTTP actions. This route handles a POST on `/create-user`, and as the name implies, this will create a user by calling the `usersModel.create` function.

As a POST request, the parameters arrive in the body of the request rather than as URL parameters. Because of the `mapParams` flag on the `bodyParams` handler, the arguments passed in the HTTP body are added to `req.params`.

We simply call `usersModel.create` with the parameters sent to us. When completed, the `result` object should be a user object, which we send back to the requestor using `res.send`.

```

// Update an existing user record
server.post('/update-user/:username', (req, res, next) => {
  usersModel.update(req.params.username, req.params.password,
    req.params.provider, req.params.lastName,
    req.params.givenName, req.params.middleName,
    req.params.emails,   req.params.photos)
  .then(foo => {
    log('updated ' + util.inspect(result));
    res.send(result);
    next(false);
  })
  .catch(err => { res.send(500, err);
    error(err.stack); next(false); });
});

```

The `/update-user` route is handled in a similar way. However, we have put the `username` parameter on the URL. Like Express, Restify lets you put named parameters in the URL like this. Such named parameters are also added to `req.params`.

We simply call `usersModel.update` with the parameters sent to us. That, too, returns an object we send back to the caller with `res.send`.

```

// Find a user, if not found create one given profile information
server.post('/find-or-create', (req, res, next) => {
  usersModel.findOrCreate({
    id: req.params.username, username: req.params.username,
    password: req.params.password,
  })
  .then(result => {
    res.send(result);
  })
  .catch(error => {
    res.send(500, error);
  });
});

```

```
        provider: req.params.provider,
        familyName: req.params.familyName,
        givenName: req.params.givenName,
        middleName: req.params.middleName,
        emails: req.params.emails, photos: req.params.photos
    })
    .then(result => {
        res.send(result);
        next(false);
    })
    .catch(err => { res.send(500, err);
        error(err.stack); next(false); });
});
});
```

This handles our `findOrCreate` operation. We simply delegate this to the model code, as done previously.

As the name implies, we'll look to see if the named user already exists and, if so, simply return that user, otherwise it will be created.

```
// Find the user data (does not return password)
server.get('/find/:username', (req, res, next) => {
    userModel.find(req.params.username).then(user => {
        if (!user) {
            res.send(404, new Error("Did not find " +
                + req.params.username));
        } else {
            res.send(user);
        }
        next(false);
    })
    .catch(err => { res.send(500, err);
        error(err.stack); next(false); });
});
});
```

Here we support looking up the user object for the provided `username`.

If the user was not found, then we return a 404 status code because it indicates a resource that does not exist. Otherwise, we send the object that was retrieved.

```
// Delete/destroy a user record
server.del('/destroy/:username', (req, res, next) => {
    userModel.destroy(req.params.username)
    .then(() => { res.send({}); next(false); } )
    .catch(err => { res.send(500, err);
        error(err.stack); next(false); });
});
});
```

This is how we delete a user from the Notes application. The `DEL` HTTP verb is meant to be used to delete things on a server, making it the natural choice for this functionality.

```
// Check password
server.post('/passwordCheck', (req, res, next) => {
    userModel.userPasswordCheck(req.params.username,
                                req.params.password)
    .then(check => { res.send(check); next(false); })
    .catch(err => { res.send(500, err);
                     error(err.stack); next(false); });
});
```

This is another aspect of keeping the password solely within this server. The password check is performed by this server, rather than in the Notes application. We simply call the `userModel.userPasswordCheck` function shown earlier and send back the object it returns.

```
// List users
server.get('/list', (req, res, next) => {
    userModel.listUsers().then(userlist => {
        if (!userlist) userlist = [];
        res.send(userlist);
        next(false);
    })
    .catch(err => { res.send(500, err);
                     error(err.stack); next(false); });
});
```

Then, finally, in case we send a list of Notes application users back to the requestor. In case no list of users is available, we at least send an empty array.

```
server.listen(process.env.PORT, "localhost", function() {
    log(server.name + ' listening at ' + server.url);
});

// Mimic API Key authentication.

var apiKey = [
    {
        user: 'them',
        key: 'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF'
    }
];

function check(req, res, next) {
    if (req.authorization) {
```

```
var found = false;
for (let auth of apiKey) {
    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"));
    error('Failed authentication check '
        + util.inspect(req.authorization));
    next(false);
}
} else {
    res.send(500, new Error('No Authorization Key'));
    error('NO AUTHORIZATION');
    next(false);
}
}
```

As with the Notes application, we listen to the port named in the `PORT` environment variable. By explicitly listening only on `localhost`, we'll limit the scope of systems that can access the user authentication server. In a real deployment, we might have this server behind a firewall with a tight list of host systems allowed to have access.

This last function, `check`, implements authentication for the REST API itself. This is the handler function we added earlier.

It requires the caller to provide credentials on the HTTP request using the basic auth headers. The `authorizationParser` handler looks for this and gives it to us on the `req.authorization.basic` object. The `check` function simply verifies that the named user and password combination exists in the local array.

This is meant to mimic assigning an API key to an application. There are several ways of doing so; this is just one.

This approach is not limited to just authenticating using HTTP basic auth. The Restify API lets us look at any header in the HTTP request, meaning we could implement any kind of security mechanism we like. The `check` function could implement some other security method, with the right code.

Because we added `check` with the initial set of `server.use` handlers, it is called on every request. Therefore, every request to this server must provide the HTTP basic auth credentials required by this check.

This strategy is good if you want to control access to every single function in your API. For the user authentication service, that's probably a good idea. Some REST services in the world have some API functions that are open to the world and others protected by an API token. To implement that, the `check` function should not be configured among the `server.use` handlers. Instead it should be added to the appropriate route handlers like so:

```
server.get('/request/url', authHandler, (req, res, next) => {  
  ..  
});
```

Such an `authHandler` would be coded similarly to our `check` function. A failure to authenticate is indicated by sending an error code and using `next(false)` to end the routing function chain.

We now have the complete code for the user authentication server. It defines several request URLs, and for each, the corresponding function in the user model is called.

Now we need a YAML file to hold the database credentials, so create `sequelize-sqlite.yaml` containing the following code:

```
dbname: users  
username:  
password:  
params:  
  dialect: sqlite  
  storage: users-sequelize.sqlite3
```

Since this is Sequelize, it's easy to switch to other database engines simply by supplying a different configuration file. Remember that the filename of this configuration file must appear in the `SEQUELIZE_CONNECT` environment variable.

And, finally, `package.json` should look like the following:

```
{  
  "name": "user-auth-server",  
  "version": "0.0.1",  
  "description": "",  
  "main": "user-server.js",  
  "scripts": {  
    "start": "DEBUG=users:* PORT=3333 SEQUELIZE_CONNECT=sequelize-  
sqlite.yaml node user-server"
```

```
},
"author": "",
"license": "ISC",
"dependencies": {
  "debug": "^2.2.0",
  "js-yaml": "^3.5.3",
  "mysql": "^2.10.2",
  "restify": "^4.0.4",
  "sequelize": "^3.19.3",
  "sqlite3": "^3.1.1"
}
}
```

We configure this server to listen on port 3333 using the database credentials we just gave and with debugging output for the server code.

You can now start the user authentication server:

```
$ npm start
```

```
> user-auth-server@0.0.1 start /Users/david/chap08/users
> DEBUG=users:* PORT=3333 SEQUELIZE_CONNECT=sequelize-mysql.yaml node
user-server

users:server User-Auth-Service listening at http://127.0.0.1:3333 +0ms
```

But we don't have any way to interact with this server, yet.

Scripts to test and administer the User Authentication server

To give ourselves assurance that the user authentication server works, we can write a couple of scripts to exercise the API. And, because we're not going to take the time to write a full administrative backend to the Notes application, these scripts will let us add and delete users who are allowed access to Notes.

These scripts will live within the user authentication server package directory, but will make calls using the REST API.

The Restify package supports coding not only in REST servers but also in REST clients. We'll use its client code in these scripts, and later when we integrate authentication into the Notes application.

Create a file named `users-add.js` containing the following code:

```
'use strict';

const util = require('util');
const restify = require('restify');

var client = restify.createJsonClient({
  url: 'http://localhost:' + process.env.PORT,
  version: '*'
});

client.basicAuth('them', 'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF');

client.post('/create-user', {
  username: "me", password: "w0rd", provider: "local",
  familyName: "Einarrsdottir", givenName: "Ashildr",
  middleName: "", emails: [], photos: []
},
(err, req, res, obj) => {
  if (err) console.error(err.stack);
  else console.log('Created ' + util.inspect(obj));
});
});
```

This is the basic structure of a Restify client. We create the `Client` object—we have a choice between the `JsonClient`, as here, the `StringClient`, and the `HttpClient`. The HTTP basic auth credentials are easy to set, as shown here.

Then we make the request, in this case a `POST` request on `/create-user`. Because it is a `POST` request, the object we specify here is formatted by Restify into `HTTP POST` body parameters. As we saw earlier, the server has the `bodyParser` handler function configured, which converts those body parameters into the `req.param` object.

In the Restify client, as for the Restify server, we use the various HTTP methods by calling `client.METHOD`. Because it is a `POST` request, we use `client.post`.

When the request finishes, the callback function is invoked.

Before running these scripts, start the authentication server in one window using the following command:

```
$ npm start
```

Now run the test script using the following command:

```
$ PORT=3333 node users-add.js
Created { id: 1,
  username: 'me',
  password: 'w0rd',
  provider: 'local',
  familyName: 'Einarrsdottir',
  givenName: 'Ashildr',
  middleName: '',
  emails: '[]',
  photos: '[]',
  updatedAt: '2016-02-24T02:34:41.661Z',
  createdAt: '2016-02-24T02:34:41.661Z' }
```

And we can inspect our handiwork using the following command:

```
$ sqlite3 users-sequelize.sqlite3
SQLite version 3.10.2 2016-01-20 15:27:19
Enter ".help" for usage hints.
sqlite> .schema users
CREATE TABLE `Users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `username` VARCHAR(255) UNIQUE, `password` VARCHAR(255), `provider` VARCHAR(255), `familyName` VARCHAR(255), `givenName` VARCHAR(255), `middleName` VARCHAR(255), `emails` VARCHAR(2048), `photos` VARCHAR(2048), `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, UNIQUE (username));
sqlite> select * from users;
1|me|w0rd|local|Einarrsdottir|Ashildr||[]|[]|2016-02-24 02:34:41.661
+00:00|2016-02-24 02:34:41.661 +00:00
sqlite> ^D
```

Now let's write a script, `users-find.js`, to look up a given user:

```
'use strict';

const util = require('util');
const restify = require('restify');

var client = restify.createJsonClient({
  url: 'http://localhost:' + process.env.PORT,
  version: '*'
});
```

```
client.basicAuth('them', 'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF');

client.get('/find/' + process.argv[2],
  (err, req, res, obj) => {
    if (err) console.error(err.stack);
    else console.log('Found ' + util.inspect(obj));
  });
});
```

This simply calls the `/find` URL, specifying the `username` the user supplies as a command-line argument. Note that the `get` operation does not take an object full of parameters. Instead, any parameters would be added to the URL.

It's run as so:

```
$ PORT=3333 node users-find.js me
Found { username: 'me',
  provider: 'local',
  familyName: 'Einarssdottir',
  givenName: 'Ashildr',
  middleName: '',
  emails: '',
  photos: '' }
```

Similarly, we can write scripts against the other REST functions. But we need to get on with the real goal of integrating this into the Notes application.

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.

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. We could have created user model code that directly accesses the database but, for reasons already discussed, we've decided to segregate user authentication into a separate service.

Let us now turn to the Notes application, which you may have stored as `chap08/notes`.

Because we're using Restify for its client library, install its package as so:

```
$ npm install restify@4.x --save
```

Create a new file, `models/users-rest.js`, containing the following code:

```
'use strict';

const restify = require('restify');
const log    = require('debug')('notes:users-rest-client');
const error  = require('debug')('notes:error');

var connectREST = function() {
    return new Promise((resolve, reject) => {
        try {
            resolve(restify.createJsonClient({
                url: process.env.USER_SERVICE_URL,
                version: '*'
            }));
        } catch (err) {
            reject(err);
        }
    })
    .then(client => {
        client.basicAuth('them',
            'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF');
        return client;
    });
};

As we did for the Notes model code, we start with a connectREST function that sets up the REST client connection to the server. The restify.createJsonClient function gives us a client object that we're using to connect to the server. The USER_SERVICE_URL environment variable specifies the URL for our server. With the configuration we put in users/package.json, this URL would be http://localhost:3333.
```

The client is configured with HTTP basic auth credentials matching what we programmed into the server:

```
exports.create = function(username, password, provider, familyName,
givenName, middleName, emails, photos) {
    return connectREST().then(client => {
        return new Promise((resolve, reject) => {
            client.post('/create-user', {
```

```

        username, password, provider,
        familyName, givenName, middleName, emails, photos
    },
    (err, req, res, obj) => {
        if (err) return reject(err);
        resolve(obj);
    });
});
};

exports.update = function(username, password, provider, familyName,
givenName, middleName, emails, photos) {
    return connectREST().then(client => {
        return new Promise((resolve, reject) => {
            client.post('/update-user/'+ username, {
                password, provider,
                familyName, givenName, middleName, emails, photos
            },
            (err, req, res, obj) => {
                if (err) return reject(err);
                resolve(obj);
            });
        });
    });
};
}
;
```

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.

 These anonymous objects are a little different than usual. We're using a new 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 field names of the object to match parameter names used by the server. By 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.

```

exports.find = function(username) {
    return connectREST().then(client => {
        return new Promise((resolve, reject) => {

```

```
        client.get('/find/' + username,
        (err, req, res, obj) => {
            if (err) return reject(err);
            resolve(obj);
        });
    });
});
};
```

Our find operation lets us look up user information.

```
exports.userPasswordCheck = function(username, password) {
    return connectREST().then(client => {
        return new Promise((resolve, reject) => {
            client.post('/passwordCheck', {
                username, password
            },
            (err, req, res, obj) => {
                if (err) return reject(err);
                resolve(obj);
            });
        });
    });
};
```

And we're sending the request to check passwords to the server.

```
exports.findOrCreate = function(profile) {
    return connectREST().then(client => {
        return new Promise((resolve, reject) => {
            client.post('/find-or-create', {
                username: profile.id,
                password: profile.password,
                provider: profile.provider,
                familyName: profile.familyName,
                givenName: profile.givenName,
                middleName: profile.middleName,
                emails: profile.emails, photos: profile.photos
            },
            (err, req, res, obj) => {
                if (err) return reject(err);
                resolve(obj);
            });
        });
    });
};
```

Following is the `findOrCreate` function. The `profile` object will come from Passport, but notice carefully 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.

```
exports.listUsers = function() {
    return connectREST().then(client => {
        return new Promise((resolve, reject) => {
            client.get('/list', (err, req, res, obj) => {
                if (err) return reject(err);
                resolve(obj);
            });
        });
    });
};
```

And, finally, we can retrieve a list of users.

Login and logout routing functions

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. 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:

```
$ npm install passport@0.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, correspond to the various third-party services that support authentication, such as using OAuth2 to authenticate against services like Facebook, Twitter, or GitHub.

The `LocalStrategy` authenticates solely using data stored local to the application, for example, our user authentication information service.

Let's start by creating the routing module, `routes/users.js`:

```
'use strict';

const path  = require('path');
const log   = require('debug')('notes:router-users');
```

```
const error = require('debug')('notes:error');
const express = require('express');
const router = express.Router();
exports.router = router;
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const usersModel = require(process.env.USERS_MODEL
  ? path.join('..', process.env.USERS_MODEL)
  : '../models/users-rest');
```

This brings in the modules we need for this router. This includes the two Passport modules and the REST-based user authentication model. We've coded this so that this can be overridden with an environment variable, if needed.

Notice that for the other router modules, we had the following line of code:

```
module.exports = router;
```

But for this router, the preceding line reads:

```
exports.router = router;
```

What's going on is we have a few functions to export from this module. Rather than having the Router object be the sole export, we make it one of several exports. In `app.js`, the following line will be there:

```
app.use('/users', users.router);
```

But we haven't gotten there yet, so let's not get ahead of ourselves.

```
exports.initPassport = function(app) {
  app.use(passport.initialize());
  app.use(passport.session());
};

exports.ensureAuthenticated = function(req, res, next) {
  // req.user is set by Passport in the deserialize function
  if (req.user) next();
  else res.redirect('/users/login');
};
```

The `initPassport` function will be called from `app.js`, and it installs the Passport middleware into the Express configuration. We'll discuss the implications of this later when we get to `app.js` changes, but Passport uses sessions to detect whether this HTTP request is authenticated or not. It looks at every request coming into the application, looks for clues about whether this browser is logged in or not, and attaches data to the request object as `req.user`.

The `ensureAuthenticated` function will be used by other routing modules and 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.js` must use `ensureAuthenticated`. If the user is not logged in, this function redirects them to `/users/login` so that they can do so.

```
router.get('/login', function(req, res, next) {
  // log(util.inspect(req));
  res.render('login', {
    title: "Login to Notes",
    user: req.user,
  });
});

router.post('/login',
  passport.authenticate('local', {
    successRedirect: '/',
    failureRedirect: 'login', // FAIL: Go to /user/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 to the `/users/login` page.

```
router.get('/logout', function(req, res, next) {
  req.logout();
  res.redirect('/');
});
```

When the user requests to log out of Notes, they are to be sent to `/users/logout`. When we get to changes in the templates, we'll add a button for this purpose. The `req.logout` function instructs Passport to erase their login credentials, and they are then redirected to the home page.

```
passport.use(new LocalStrategy(
  function(username, password, done) {
    usersModel.userPasswordCheck(username, password)
      .then(check => {
        if (check.check) {
          done(null, { id: check.username,
```

```
        username: check.username }) ;  
    } else {  
      done(null, false, check.message) ;  
    }  
    return check;  
})  
.catch(err => done(err)) ;  
}  
));
```

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 they're logged in or not.

A successful login is indicated when `check.check` is `true`. For this case, we tell Passport to use an object containing the `username` in the session object. Otherwise, we have two ways to tell Passport the login attempt was unsuccessful.

```
passport.serializeUser(function(user, done) {  
  done(null, user.username) ;  
}) ;  
  
passport.deserializeUser(function(username, done) {  
  usersModel.find(username)  
.then(user => done(null, user))  
.catch(err => done(err)) ;  
});
```

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 requests and is where we look up the user profile data. Passport will attach this to the request object.

Login/logout changes to app.js

We have a few changes required in `app.js`, some of which we've already touched on. We did carefully isolate the Passport module dependencies to `routes/users.js`. The changes required in `app.js` support the code in `routes/users.js`.

It's now time to uncomment a line we told you to comment out way back in *Chapter 5, Your First Express Application*:

```
var users = require('../routes/users');
```

We need to use the routes we just implemented to support user login/logout:

```
const session = require('express-session');
const FileStore = require('session-file-store')(session);
```

Because Passport uses sessions, we need to enable session support in Express. 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.

Use the following command to install the modules:

```
$ npm install express-session@1.x session-file-store@0.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>:

```
app.use(session({
  store: new FileStore({ path: "sessions" }),
  secret: 'keyboard mouse',
  resave: true,
  saveUninitialized: true
}));
users.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 string `keyboard cat` 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.

The `FileStore` will store its session data records in a directory named `sessions`. This directory will be autogenerated as needed:

```
app.use('/', routes);
app.use('/users', users.router);
app.use('/notes', notes);
```

The preceding are the three routers used in the Notes application. As we saw previously, the user login/logout router is on `users.router`, whereas for the other router modules, the router is the solitary export.

Login/logout changes in routes/index.js

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:

```
router.get('/', function(req, res, next) {
  notes.keylist()
    .then(keylist => {
      var keyPromises = keylist.map(key => {
        return notes.read(key).then(note => {
          return { key: note.key, title: note.title };
        });
      });
      return Promise.all(keyPromises);
    })
    .then(notelist => {
      res.render('index', {
        title: 'Notes',
        notelist: notelist,
        user: req.user ? req.user : undefined,
        breadcrumbs: [
          { href: '/', text: 'Home' }
        ]
      });
    })
    .catch(err => { error(err); next(err); });
});
```

Remember that we ensured `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.js

The changes required here are more significant, but still straightforward:

```
const usersRouter = require('./users');
```

We need to use the `ensureAuthenticated` function to protect certain routes from being used by users who are not logged in. Since that function is in the user router module, we need to require it here:

```
router.get('/add', usersRouter.ensureAuthenticated, (req, res, next)
=> {
  res.render('noteedit', {
    title: "Add a Note",
    docreate: true,
    notekey: "",
    note: undefined,
    user: req.user ? req.user : undefined,
    breadcrumbs: [
      { href: '/', text: 'Home' },
      { active: true, text: "Add Note" }
    ],
    hideAddNote: true
  });
});
```

The first thing we added is to call `usersRouter.ensureAuthenticated` in the route definition. If the user is not logged in, they'll redirect to `/users/login`, thanks to that function.

Because we've ensured 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:

```
router.post('/save', usersRouter.ensureAuthenticated, (req, res, next)
=> {
  ...
});
```

The `/save` route requires only this change to call `ensureAuthenticated` to make sure the user is logged in:

```
router.get('/view', (req, res, next) => {
  ...
  .then(note => {
    res.render('noteview', {
      ...
      user: req.user ? req.user : undefined,
      ...
    });
  })
});
```

For this route, we don't require the user to be logged in. But we do need the user's profile information sent to the view template:

```
router.get('/edit', usersRouter.ensureAuthenticated, (req, res, next) => {
  ...
  res.render('noteedit', {
    ...
    user: req.user ? req.user : undefined,
    ...
  });
  ...
});

router.get('/destroy', usersRouter.ensureAuthenticated, (req, res, next) => {
  ...
  res.render('notedestroy', {
    ...
    user: req.user ? req.user : undefined,
    ...
  });
  ...
});

router.post('/destroy/confirm', usersRouter.ensureAuthenticated, (req, res, next) => {
  ...
});
```

For these routes, we require the user to be logged in. In most cases, we need to send the `req.user` value to the view template.

View template changes supporting login/logout

We're almost ready, but we've got a number of outstanding changes to make in 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 or not.

In `views/pageHeader.ejs`, make the following additions:

```
<header class="page-header">
<h1><%= title %></h1>
<% if (user && typeof hideAddNote === 'undefined') { %>
<a class="btn btn-primary" href="/notes/add">ADD Note</a>
<% } %>
<% if (user) { %>
```

```

<a class="btn btn-primary" href="/users/logout">
    Log Out <span class="badge"><%= user.username %></span></a>
<% } else { %>
<a class="btn btn-primary" href="/users/login">Log in</a>
<% } %>
<% if (typeof breadcrumbs !== 'undefined' && breadcrumbs) { %>
..
<% } %>
</header>

```

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 or not. What we've done with the earlier changes is ensure that the `user` variable will be `undefined` if the user is logged out, otherwise it will have the user profile object.

A logged-out user doesn't get the **Add Note** button, and gets a **Login** button. Otherwise, the user gets an **Add Note** button and a **Logout** button. The **Login** button takes the user to `/users/login`, while the **Logout** button takes them to `/users/logout`. Both of those 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 cue to the user what identity they have.

We need to create `views/login.ejs`:

```

<!DOCTYPE html>
<html>
<head><% include headerStuff %></head>
<body>
<div class="container-fluid">
<% include pageHeader %>
<div class="row"><div class="col-xs-12">
<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>
<% include footer %>
</div>
</body>
</html>
```

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.

In `views/notedestroy.ejs`, 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:

```
...
<div class="row"><div class="col-xs-12">
<% if (user) { %>
<form method='POST' action='/notes/destroy/confirm'>
<input type='hidden' name='notekey' value='<%= note ? notekey : "" %>'>
<p>Delete <%= note.title %> ?</p>
<br/>
<input type='submit' value='DELETE' /> <a class="btn btn-default" href="/notes/view?key=<%= notekey %>">Cancel</a>
</form>
<% } else { %><% include not-logged-in %><% } %>
</div></div>
...
```

That's straightforward; if the user is logged in, display the form, otherwise display this other thing, `not-logged-in.ejs`.

In `views/noteedit.ejs`, we need a similar change:

```
...
<div class="row"><div class="col-xs-12">
<% if (user) { %>
...
<% } else { %><% include not-logged-in %><% } %>
</div></div>
...
```

And then we need to create `views/not-logged-in.ejs`:

```
<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>
```

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 preverified the user is logged in.

A message like 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 like this is another layer of prevention against displaying forms to users who are not allowed to use that functionality.

Running the Notes application with user authentication

Now we're ready to run the Notes application and to try our hand at logging in and out.

We need to change the scripts section of `package.json` as so:

```
"scripts": {
  "start": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_"
  MODEL=models/notes-sequelize USERS_MODEL=models/users-rest USER_
  SERVICE_URL=http://localhost:3333 node ./bin/www",
  "postinstall": "bower install",
  "bootstrapsetup": "cd bower_components/bootstrap && npm install &&
  npm install grunt-cli",
  "buildbootstrap": "cp variables.less bower_components/bootstrap/
  less && cd bower_components/bootstrap && grunt"
},
```

In the previous chapters, we built up quite a few combinations of models and databases for running the Notes application. 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 we wrote earlier. All the other Notes data models are still available just by setting the environment variables appropriately. The **USER_SERVICE_URL** needs to match the port number we designated for that service.

In one window, start the user authentication service as so:

```
$ cd users  
$ npm start
```

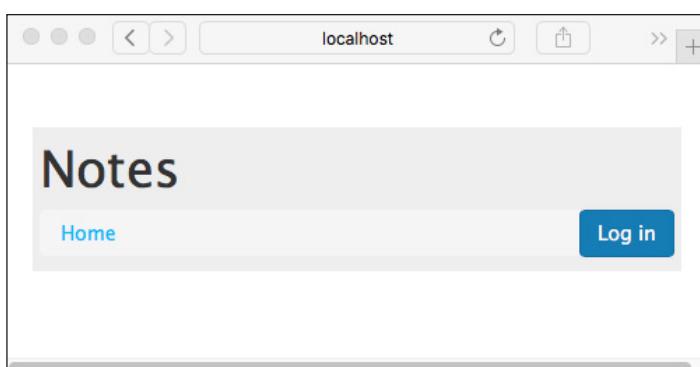
```
> user-auth-server@0.0.1 start /Users/david/chap08/users  
> DEBUG=users:* PORT=3333 SEQUELIZE_CONNECT=sequelize-sqlite.yaml node  
user-server
```

```
users:server User-Auth-Service listening at http://127.0.0.1:3333 +0ms
```

Then, in another window, start the Notes application:

```
$ cd notes  
$ DEBUG=notes:* npm start  
  
> notes@0.0.0 start /Users/david/chap08/notes  
> SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_MODEL=models/  
notes-sequelize USERS_MODEL=models/users-rest USER_SERVICE_URL=http://  
localhost:3333 node ./bin/www  
  
notes:server Listening on port 3000 +0ms
```

And you'll be greeted with the following:



Notice the new button, **Log in**, and the lack of an **Add Note** button.

Click on the **Log in** button, and you will see the login screen:

The screenshot shows a web browser window with the URL 'localhost' in the address bar. The main content is a login form with a title 'Login to Notes'. It contains two text input fields: one for 'User name:' with the value 'me' and another for 'Password:' with the value '****'. Below these is a 'Submit' button. To the right of the input fields is a blue 'Log in' button. The browser interface includes standard navigation buttons (back, forward, search) and a tab bar.

This is our login form from `views/login.ejs`. You can now log in, create a note or three, and you might end up with the following on the home page:

The screenshot shows a web browser window with the URL 'localhost'. The main content is a notes application with a title 'Notes'. It features a 'Home' link in the top left, a 'Log Out' button with the user name 'me' in the top right, and an 'ADD Note' button next to it. Below the header is a large text area containing the message 'Hi Mom'. The browser interface includes standard navigation buttons and a tab bar.

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 or not, and which user is logged in. This might drive the user experience team nuts, and you won't know if this user interface design works until its tested with users, but it's good enough for our purpose now.

The implementation of this is in `views/pageHeader.ejs`.

Twitter login support for the Notes application

If you want your application to hit the big time, it's a great idea to allow users to register using third-party credentials. Websites all over the Internet allow you to log in using Facebook, Twitter, or accounts from other services. Doing so removes hurdles to prospective users signing up for your service. Passport makes it extremely easy to do this.

Supporting Twitter requires installing **TwitterStrategy**, registering a new application with Twitter, and adding a couple of routes into `routes/user.js` and a small change in `views/pageHeader.ejs`. Integrating other third-party services requires similar steps.

Registering an application with Twitter

Twitter, as with every other third-party service, uses **OAuth2** to handle authentication and requires an authentication key to write software using their API. It's their service, so you have to play by their rules, of course.

To register a new application with Twitter, go to <https://apps.twitter.com/>.

The signup process asks for two URLs (website and callback URL) that may be problematic for the Notes application. The problem is that `http://localhost:3000` is not an acceptable URL to register with Twitter. But this is the URL we've used so far to use the Notes application. It seems that Twitter wants to see a proper domain name.

One option is, if you have a server available that has a domain name, to install Node.js on that server and then run the Notes application stack on that server. That is, if you have a web server hosting a domain, for instance, `example.com`, it may be possible to copy the Notes application to the command-line environment on that server and then run the Notes application on the server just as you've been doing on your laptop. You would access the application at `http://example.com:3000` and you can then tell Twitter to use that domain name.

Mac OS X users have another option because of the `.local` domain name which is automatically assigned to their laptop. All along, we could have used a URL similar to this to access the Notes application at `http://MacBook-Pro-2.local:3000/`. Fortunately, Twitter accepts that domain name in its application registration form. You can also enter an IP address such as `http://192.168.0.100:3000/`, which, of course, must match the IP address of your laptop:

Application Details

Name *
Node Web Development Example App
Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *
Example app for Node.js applications accessing Twitter API
Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *
http://MacBook-Pro-2.local:3000
Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL
http://MacBook-Pro-2.local:3000/users/auth/twitter/callback
Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Enable Callback Locking (It is recommended to enable callback locking to ensure apps cannot overwrite the callback url)
 Allow this application to be used to [Sign in with Twitter](#)

For the website URL, use the domain name you've chosen.

For the callback URL, add /users/auth/twitter/callback to that URL, such as <http://MacBook-Pro-2.local:3000/users/auth/twitter/callback>.

Implementing TwitterStrategy

Like with many web applications, we have decided to allow our users to log in using Twitter credentials. The OAuth2 protocol is widely used for this purpose and is the basis for authenticating on one website using credentials maintained by another website.

Within the Passport ecosystem are dozens of Strategy packages for various third-party services. Let's install the package required to use TwitterStrategy:

```
$ npm install passport-twitter@1.x --save
```

And in `routes/users.js`, let's start making some changes:

```
const TwitterStrategy = require('passport-twitter').Strategy;
```

To bring in the package we just installed, add the following:

```
passport.use(new TwitterStrategy({
  consumerKey: "... consumer key",
  consumerSecret: "... consumer secret",
```

```
    callbackURL: "... your callback URL"
  },
  function(token, tokenSecret, profile, done) {
    userModel.findOrCreate({
      id: profile.username, username: profile.username, password:
      "", provider: profile.provider, familyName: profile.displayName,
      givenName: "", middleName: "", photos: profile.photos, emails:
      profile.emails
    })
    .then(user => done(null, user))
    .catch(err => done(err));
  }
));
```

This registers `TwitterStrategy` 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.

We defined the `userModel.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 autoreate that user account in Notes.



The `consumerKey` and `consumerSecret` values are supplied by Twitter. These are used in the OAuth2 protocol as proof of identity to Twitter.

It was found while debugging that the `profile` object supplied by the `TwitterStrategy` did not match the documentation on the Passport website. Therefore, we have mapped the object actually supplied by Passport into something Notes can use:

```
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`. 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 OAuth2 dance begins. Passport sends the browser over to the correct URL at Twitter, where Twitter asks the user if 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 OAuth2 dance concludes, the browser lands here:

```
router.get('/auth/twitter/callback',
  passport.authenticate('twitter', { successRedirect: '/',
    failureRedirect: '/users/login' }));
```

This route handles the callback URL. Depending on whether it indicates a successful registration or not, Passport will redirect the browser to either the home page or back to the /users/login page.

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 `views/pageHeader.ejs`, make the following changes in the code:

```
<% if (user) { %>
<a class="btn btn-primary" href="/users/logout">
  Log Out <span class="badge"><%= user.username %></span></a>
<% } else { %>
<a class="btn btn-primary" href="/users/login">Log in</a>
<a class="btn btn-primary" href="/users/auth/twitter">Log in with Twitter</a>
<% } %>
```

This adds a new button that, when clicked, takes the user to `/users/auth/twitter`, which, of course, kicks off the Twitter authentication process.

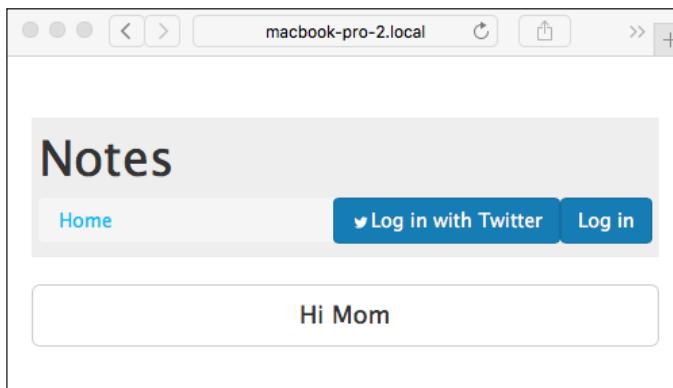
The image being used is from the official Twitter Brand Assets page at <https://about.twitter.com/company/brand-assets>. Twitter recommends using these branding assets for a consistent look across all services using Twitter.

With these changes, we're ready to try logging in with Twitter.

Start the Notes application server as done previously:

```
$ npm start
```

And then use a browser to visit `http://localhost:3000`:

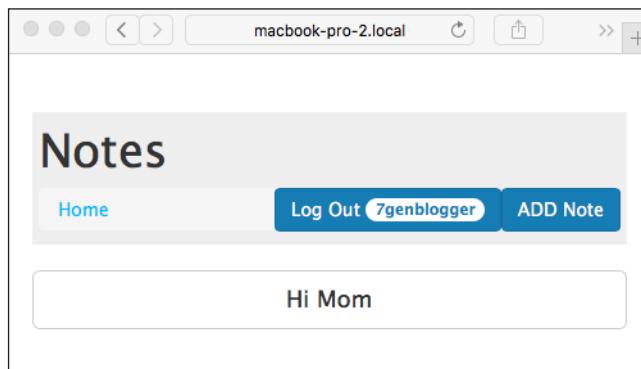


Notice the new button. It looks about right, thanks to having used the official Twitter branding image. The button is a little large, so maybe you want to consult a designer. Obviously, a different design is required if you're going to support dozens of authentication services.

Clicking on this button takes the browser to `/users/auth/twitter`, which starts Passport running the OAuth2 protocol transactions necessary to authenticate.

When this is finished, the browser lands at `/users/auth/twitter/callback`, which we've configured to redirect the user to the Notes home page.

And then, once you're logged in with Twitter, you'll see something like the following screenshot:



We're now logged in, and 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.

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:

```
"scripts": {
  "start": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_MODEL=models/notes-sequelize USERS_MODEL=models/users-rest USER_SERVICE_URL=http://localhost:3333 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:3333 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:3333 PORT=3002 node ./bin/www",
  "postinstall": "bower install",
  "bootstrapsetup": "cd bower_components/bootstrap && npm install && npm install grunt-cli",
  "buildbootstrap": "cp variables.less bower_components/bootstrap/less && cd bower_components/bootstrap && grunt"
},
```

Then, in one command window, run the following command:

```
$ npm run start-server1
```

And in another command window, run the following command:

```
$ npm run start-server2
```

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`. And, as previously, 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:

```
$ ls -l sessions/
total 8
-rw-r--r-- 1 david  staff  139 Feb 27 18:16
vMwgX8i9rs7QqWDeHCF7Ok63iKidwMhj.json
```

```
$ cat sessions/vMwgX8i9rs7QqWDeHCF7Ok63iKidwMhj.json
{"cookie": {"originalMaxAge": null, "expires": null, "httpOnly": true, "path": "/"}, "__lastAccess": 1456625814823, "passport": {"user": "7genblogger"}}
```

This is after logging in using a Twitter account; you can see the Twitter account name is stored here in the session data.

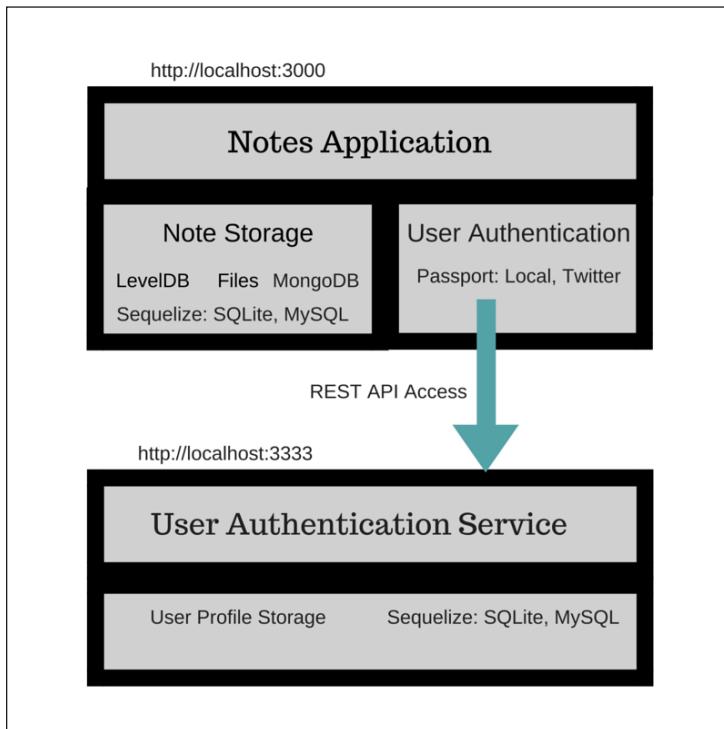
What if you want to clear a session? It's just a file in the filesystem. Deleting the session file erases the session, and the user's browser will be forcefully logged out.

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 (1 hour). With a timed-out session, the application reverts to a logged-out state.

The Notes application stack

Did you notice earlier when we said *run the Notes application stack*? It's time to ask the marketing team to explain what they meant by that phrase.

We've created enough of a system that a diagram might be helpful, in other words:



Summary

You've covered a lot of ground in this chapter, looking at not only user authentication in Express applications but also microservice development.

Specifically, you covered session management in Express, using Passport for user authentication, including Twitter/OAuth2, using router middleware to limit access, creating a REST service with Restify, and when to create a microservice.

In the next chapter, we'll take the Notes application to a new level—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 Interaction between Client and Server 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 screen. While web browsers can show more complex information than old-school dumb terminals, the interaction pattern in both cases is a back and forth of user requests, each resulting in a *screen* of data sent by the server screen after screen or, in the case of web browsers, page after page.

We are progressing rapidly. Web browsers with modern JavaScript engines can do so much more than the dumb terminals of yesteryear, or even the web browsers of just a couple years ago. One new technique is to keep a connection open with the server for continual data exchange between server and client, representing a change in the web application model; some call this the real-time web.

With JavaScript on both the server and client, we can now implement a dream that dates back to the days when Java applets tried to convince us that the browser could host interactive stuff on web pages. Modern JavaScript browser-side engines make it possible to build amazingly rich applications. In some cases, this means keeping the server connection open for exchanging data with the server. Having JavaScript on both the server and client means more facile and fluid data sharing between the frontend and backend of the application.

One observation we can make is that traditional web applications can untruthfully display their data; that is, if two people are looking at a page, such as a wiki page, and one person edits that page, that person's browser will update with the correct copy of the page, while the other browser will not update. The two browsers show different versions of the page, one of which is untruthful. The second browser can even show a page that no longer exists if the user at the first browser deletes that page. Some think it would be better if the other person's browser is refreshed to show the new content as soon as the page is edited.

This is one possible role of the real-time web; pages that update themselves as the page content changes. We're about to implement this behavior in the Notes application.

The more common role of the real-time web is the way some of the social networking websites now dynamically update the page as other people make posts or comments, or when there is a pop-up activity notification for an activity within the social network. We're about to implement a tiny messaging system inside the Notes application as well.

Instead of a page-by-page interaction between the browser and the server, the real-time web maintains a continual stream of data going between the two, with application code living in the browser, to implement the browser-side behavior.

One of the original purposes for inventing Node.js was to support real-time web implementation. The **Comet** application architecture (Comet is related to AJAX, and both happen to be names of household cleaning products) involves holding the HTTP connection open for a long time with data flowing back and forth between browser and server over that channel. The term Comet was introduced by Alex Russell on his blog in 2006 (<http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/>) as a general term for the architectural pattern to implement this real-time two-way data exchange between the client and server. That blog post called for development of a programming platform very similar to Node.js.

To explore this, we will add some real-time web functionality to the Notes application.

To simplify the task, 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.

A common companion to Socket.IO is Backbone.js, because it supports data models, allowing the programmer to exchange high-level data over Socket.IO connections. Because the data in the Notes application is so simple, we won't use Backbone. However, it is worth exploring (<http://backbonejs.org/>).

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.

If you were to implement your application with WebSocket, it would be limited to the modern browsers supporting that protocol. Because Socket.IO falls back on so many alternate protocols (WebSockets, Flash, XHR, and JSONP), it supports a wide range of web browsers, including some old crusty browsers. Older Socket.IO versions claimed to support back to Internet Explore 5.5, but that claim is no longer printed on their website.

As the application author, you don't have to worry about the specific protocol Socket.IO uses in a given browser. Instead, you can implement the business logic and the library takes care of the details for you.

Socket.IO requires that a client library make its way into the browser. That library is provided, and it is easy to instantiate. You'll be writing code on both the browser side and server side using similar Socket.IO API's at each end.

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. The emitted events are sent between the browser and the server with the Socket.IO library taking care of sending them back and forth.

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 – A Web Applications First Steps*, where we wrote a module which 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@1.x passport.socketio@3.x --save
```

We will incorporate user authentication with the Passport module, used in *Chapter 8, Multiuser Authentication the Microservice Way*, into some of the real time interactions we'll implement.

To initialize Socket.IO, we must do some major surgery on how the Notes application is started. So far, we used the bin/www script along with app.js, with each script hosting different steps of launching Notes. Socket.IO initialization requires that these steps occur in a different order than what we've been doing, and that we must merge these two scripts into one. What we'll do is copy the contents of the bin/www script into appropriate parts of app.js, and from there, we'll use app.js to launch Notes.

At the beginning of app.js, insert this:

```
#!/usr/bin/env node
```

Add this to the require statements:

```
const passportSocketIo = require("passport.socketio");
const http = require('http');
const log    = require('debug')('notes:server');
const error = require('debug')('notes:error');
```

The last three lines are copied from bin/www. The passport.socketio module integrates Socket.IO with PassportJS-based user authentication. We'll configure this support shortly.

```
const sessionCookie = 'notes.sid';
const sessionSecret = 'keyboard mouse';
const sessionStore  = new FileStore({ path: "sessions" });
```

The configuration for session management is now shared between Socket.IO, Express, and Passport. These lines centralize that configuration to one place in app.js, so we can change it once to affect every place it's needed.

Use this to initialize the HTTP Server object:

```
var app = express();

var server = http.createServer(app);
var io = require('socket.io')(server);

io.use(passportSocketIo.authorize({
  cookieParser: cookieParser,
```

```
key:           sessionCookie,  
secret:        sessionSecret,  
store:         sessionStore  
});  
  
var port = normalizePort(process.env.PORT || '3000');  
app.set('port', port);
```

The `io` object is our entry point into the Socket.IO API. We need to pass this object to any code that needs to use that API. It won't be enough to simply require the `socket.io` module in other modules because the `io` object is what wraps the `server` object.

The `io.use` function installs in Socket.IO functions similar to Express middleware. In this case, we integrate Passport authentication into Socket.IO:

```
app.use(session({  
  store: sessionStore,  
  secret: sessionSecret,  
  key: sessionCookie,  
  resave: true,  
  saveUninitialized: true  
}));
```

This changes the configuration of Express session support to match the configuration variables we set up earlier.

Use this to initialize Socket.IO code in the router modules:

```
app.use('/', routes);  
app.use('/users', users.router);  
app.use('/notes', notes);  
  
routes.socketio(io);  
// notes.socketio(io);
```

We haven't written these functions yet. This injects the object into the `routes/index.js` and `routes/notes.js` modules. The call to `notes.socketio` is commented out so that we can get to it later.

Then, at the end of `app.js`, we'll copy in the code so the HTTP Server starts listening on our selected port:

```
module.exports = app;  
  
server.listen(port);
```

```
server.on('error', onError);
server.on('listening', onListening);

function normalizePort(val) {
  var port = parseInt(val, 10);
  if (isNaN(port)) {
    // named pipe
    return val;
  }
  if (port >= 0) {
    // port number
    return port;
  }
  return false;
}

function onError(error) {
  if (error.syscall !== 'listen') {
    throw error;
  }
  var bind = typeof port === 'string'
    ? 'Pipe ' + port
    : 'Port ' + port;

  // handle specific listen errors with friendly messages
  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;
  }
}

function onListening() {
  var addr = server.address();
  var bind = typeof addr === 'string'
    ? 'pipe ' + addr
    : 'port ' + addr.port;
  log('Listening on ' + bind);
}
```

Then, in package.json, we must start app.js rather than bin/www:

```
"scripts": {
  "start": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_MODEL=models/notes-sequelize USERS_MODEL=models/users-rest USER_SERVICE_URL=http://localhost:3333 node ./app",
  "start-server1": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_MODEL=models/notes-sequelize USERS_MODEL=models/users-rest USER_SERVICE_URL=http://localhost:3333 PORT=3000 node ./app",
  "start-server2": "SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_MODEL=models/notes-sequelize USERS_MODEL=models/users-rest USER_SERVICE_URL=http://localhost:3333 PORT=3002 node ./app",
  ...
},
```

At this point, you can delete bin/www if you like. You can also try starting the server, but it'll fail because the routes.socketio function does not exist yet.

Real time updates on the Notes home page

The first thing we'll do with Socket.IO is change the Notes home page to automatically update the list of notes as notes are edited or deleted. It means a little bit of work in routes/index.js and views/index.ejs, and a lot of work in the Notes model.

Where the Notes model so far has been a passive repository of documents, it now needs to emit events to any interested parties. This is the listener pattern, and, in theory, there will be code that is interested in knowing when notes are created, edited, or destroyed. At the moment, we'll use that knowledge to update the Notes home page, but there are many potential other uses of that knowledge.

The Notes model as an EventEmitter class

The EventEmitter is the class that implements listener support. Let's create a new module, models/notes-events.js, containing the following:

```
'use strict';

const EventEmitter = require('events');
class NotesEmitter extends EventEmitter {}
module.exports = new NotesEmitter();

module.exports.noteCreated = function(note) {
```

```
module.exports.emit('notecreated', note);
};

module.exports.noteUpdate = function(note) {
    module.exports.emit('noteupdate', note);
};

module.exports.noteDestroy = function(data) {
    module.exports.emit('notedestroy', data);
};
```

This module maintains for us the listeners to notes-related events. Because we have multiple Notes models, it is useful to handle listener maintenance in a common module.

This module exports an `EventEmitter` object, meaning that the module exports two functions: `.on` and `.emit`. We then add additional methods to emit specific events related to the notes life cycle.

Let's now update `models/notes-sequelize.js` to use `notes-events` to emit events. Similar changes can be made in the other Notes models. Add the following code to it:

```
exports.events = require('./notes-events');
```

This technique adds the content of one module to the exports of another module. Not every export from a module needs to be implemented by that module. A Node.js module is simply an object, `module.exports`, that is returned by the `require` statement. The contents of that module object can come from a variety of sources. In this case, the `events` export will come from the `notes-events` module.

In the `create` function, add the following:

```
exports.create = function(key, title, body) {
    return exports.connectDB()
        .then(SQNote => {
            return SQNote.create({
                notekey: key,
                title: title,
                body: body
            });
        })
        .then(newnote => {
            exports.events.noteCreated({
                key: newnote.key,
                title: newnote.title,
                body: newnote.body
            });
        })
};
```

```

    });
    return newnote;
});
};

}
;

```

This is the pre-existing `create` function but with a second step to emit the `notecreated` event. We'll make similar changes throughout this module, adding an additional step to emit events.

Remember that the `exports.events` object came from `require('./notes-events')`. That object exported a set of functions, one of which is `noteCreated`. This is how we'll use that module.

In this case, we add a clause to the promise to emit an event for every note that is created. The Promise still returns the `SQNote` object that was created. We simply add this side effect of emitting an event.

Then, make a similar change for updating a note:

```

exports.update = function(key, title, body) {
    return exports.connectDB()
        .then(SQNote => {
            return SQNote.find({ where: { notekey: key } })
        })
        .then(note => {
            if (!note) {
                throw new Error("No note found for key " + key);
            } else {
                return note.updateAttributes({
                    title: title,
                    body: body
                });
            }
        })
        .then(newnote => {
            exports.events.noteUpdate({
                key,
                title: newnote.title,
                body: newnote.body
            });
            return newnote;
        });
};

```

Here, we send a notification that a note was edited.

Finally, the destroy method:

```
exports.destroy = function(key) {
    return exports.connectDB()
        .then(SQNote => {
            return SQNote.find({ where: { notekey: key } })
        })
        .then(note => note.destroy())
        .then(() => exports.events.noteDestroy({ key }));
};
```

This sends a notification that a note was destroyed.

We only changed the notes-sequelize.js module. The other notes models will require similar changes should you want to use one of them to store notes.

Real-time changes in the Notes home page

The changes in the Notes model are only the first step. Making the events visible to our users means the controller and view portions of the application must consume those events.

Let's start making changes to routes/index.js:

```
router.get('/', function(req, res, next) {
    getKeyTitlesList().then(notelist => {
        var user = req.user ? req.user : undefined;
        res.render('index', {
            title: 'Notes',
            notelist: notelist,
            user: user,
            breadcrumbs: [{ href: '/', text: 'Home' }]
        });
    })
    .catch(err => { error('home page '+ err); next(err); });
});
```

We need to reuse part of the code that had been in this router function. Let's now add this function:

```
module.exports = router;

var getKeyTitlesList = function() {
    return notes.keylist()
        .then(keylist => {
```

```

var keyPromises = keylist.map(key => {
    return notes.read(key).then(note => {
        return { key: note.key, title: note.title };
    });
});
return Promise.all(keyPromises);
);
};

```

This is the same as earlier, just in its own function. It generates an array of items containing the `key` and `title` for all existing notes:

```

module.exports.socketio = function(io) {
    var emitNoteTitles = () => {
        getKeyTitlesList().then(notelist => {
            io.of('/home').emit('notetitles', { notelist });
        });
    };
    notes.events.on('notecreated', emitNoteTitles);
    notes.events.on('noteupdate', emitNoteTitles);
    notes.events.on('notedestroy', emitNoteTitles);
};

```

This is where `app.js` injects the `io` object into the home page routing module.

Now, we get to some actual Socket.IO code. Remember that this function is called from `app.js`, and it is given the `io` object we created there.



The `io.of` method defines what Socket.IO calls a **namespace**. Namespaces limit the scope of messages sent through Socket.IO. The default namespace is `/`, and namespaces look like pathnames; in that, they're a series of slash-separated names. An event emitted into a namespace is delivered to any socket listening to that namespace.

The code in this case is fairly straightforward. It listens to the events we just implemented, `notecreated`, `noteupdate`, and `notedestroy`. For each of these events, it emits an event, `notetitles`, containing the list of note keys and titles.

That's it!

As notes are created, updated, and destroyed, we ensure that the home page will be refreshed to match. The home page template, `views/index.ejs`, will require code to receive that event and rewrite the page to match.

Changing the home page template

Socket.IO runs on both client and server, with the two communicating back and forth over the HTTP connection. This requires loading the client JavaScript library into the client browser. Each page of the Notes application in which we seek to implement Socket.IO services must load the client library and have custom client code for our application.

In `views/index.ejs`, add the following code:

```
<% include footer %>

<script src="/socket.io/socket.io.js"></script>
<script>
$(document).ready(function () {
  var socket = io('/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-default" href="/notes/view?key=' +
          notedata.key + '">' + notedata.title + '</a>');
    }
  });
});
</script>
```

The first line is where we load the Socket.IO client library. 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.

Because we've already loaded jQuery (to support Bootstrap), we can easily ensure that this code is executed once the page is fully loaded using `$(document).ready`.

This code first connects a `socket` object to the `/home` namespace. That namespace is being used for events related to the Notes homepage. We then listen for the `notetitles` events, for which some jQuery DOM manipulation erases the current list of notes and renders a new list on the screen.

That's it. Our code in `routes/index.js` 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.

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:

```
$ cd chap09/users
$ npm start
> user-auth-server@0.0.1 start /Users/david/chap08/users
> DEBUG=users:* PORT=3333 SEQUELIZE_CONNECT=sequelize-sqlite.yaml node
user-server

users:server User-Auth-Service listening at http://127.0.0.1:3333 +0ms
```

Then, in another window, start the Notes application:

```
$ cd chap09/notes
$ npm start
> notes@0.0.0 start /Users/david/chap09/notes
> SEQUELIZE_CONNECT=models/sequelize-sqlite.yaml NOTES_MODEL=models/
notes-sequelize USERS_MODEL=models/users-rest USER_SERVICE_URL=http://
localhost:3333 node ./app

notes:server 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.

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 homepage 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. The third feature will require a messaging subsystem, so we'll get to that later in this chapter.

In `routes/notes.js`, add this to the end of the module:

```
module.exports.socketio = function(io) {
  var nspView = io.of('/view');
  var forNoteViewClients = function(cb) {
    nspView.clients((err, clients) => {
      clients.forEach(id => {
        cb(nspView.connected[id]);
      });
    });
  };

  notes.events.on('noteupdate', newnote => {
    forNoteViewClients(socket => {
      socket.emit('noteupdate', newnote);
    });
  });
  notes.events.on('notedestroy', data => {
    forNoteViewClients(socket => {
      socket.emit('notedestroy', data);
    });
  });
};
```

Now is the time to uncomment that line of code in `app.js` because we've now implemented the function we said we'd get to later.

```
routes.socketio(io);
notes.socketio(io);
```

First, we will connect to the `/view` namespace. This means any browser viewing any note in the application will connect to this namespace. Every such browser will receive events about any note being changed, even those notes that are not being viewed. This means that the client code will have to check the key, and only take action if the event refers to the note being displayed.

What we're doing is listening to the `noteupdate` and `notedestroy` messages from the Notes model and sending them to the `/view` Socket.IO namespace. However, the code shown here doesn't use the method documented on the Socket.IO website.

We should be able to write the code as follows:

```
notes.events.on('noteupdate', newnote => {
    io.of('/view').emit('noteupdate', newnote);
});
```

However, with this, there were difficulties initializing communications that meant reloading the page twice.

With this code, we dig into the data that each namespace uses to track the connected clients. We simply loop through those clients and invoke a callback function on each. In this case, the callback emits the message we need to be sent to the browser.

Changing the note view template for real-time action

As we did earlier, to make these events visible to the user, we must add client code to the template; in this case, `views/noteview.ejs`:

```
<div class="row"><div class="col-xs-12">
<h3 id="notetitle"><%= note ? note.title : "" %></h3>
<p id="notebody"><%= note ? note.body : "" %></p>
<p>Key: <span id="notekey"><%= notekey %></span></p>
</div></div>

<% if (notekey && user) { %>
    <div class="row">
        <div class="btn-group col-sm-12">
            <a class="btn btn-default" href="/notes/destroy?key=<%= notekey %>" role="button">Delete</a>
            <a class="btn btn-default" href="/notes/edit?key=<%= notekey %>" role="button">Edit</a>
        </div>
    </div>
<% } %>

<% include footer %>

<% if (notekey) { %>
<script src="/socket.io/socket.io.js"></script>
<script>
$(document).ready(function () {
    io('/view').on('noteupdate', function(note) {
        if (note.key === "<%= notekey %>") {
```

```
$('h3#notetitle').empty();
$('h3#notetitle').text(note.title);
$('p#notebody').empty();
$('p#notebody').text(note.body);
}
});
io('/view').on('notedestroy', function(data) {
if (data.key === "<%= notekey %>") {
window.location.href = "/";
}
});
</script>
<% } %>
```

We connect to the namespace where the messages are sent. As noteupdate or notedestroy messages arrive, we check the key to see whether it matches the key for the note being displayed. There is a technique used here that's important to understand; it mixes JavaScript executed on the server with JavaScript executed in the browser.

For example, remember that the code within the `<% .. %>` or `<%= .. %>` delimiters is interpreted by the EJS template engine on the server. Consider the following:

```
if (note.key === "<%= notekey %>") {
..
}
```

This comparison is between the `notekey` value in the browser, which arrived inside the message from the server, against the `notekey` variable on the server. That variable contains the key of the note being displayed. Therefore, in this case, we are able to ensure these code snippets are executed only for the note being shown on the screen.

For the noteupdate event, we take the new note content and display it on the screen. For this to work, we had to add `id=` attributes to the HTML so we could use jQuery selectors in manipulating the DOM.

For the notedestroy event, we simply redirect the browser window back to the home page. The note being viewed has been deleted, and there's no point for the user to continue looking at a note which no longer exists.

Running Notes with real-time updates while viewing a note

At this point, you can now rerun the Notes application and try this out.

Launch the user authentication server and the Notes application as before. Then, in the browser, open multiple windows on the Notes application. This time, have one viewing the home page, and two viewing a note. In one of those windows, edit the note to make a change, and see the text change on both the homepage and note viewing page.

Then delete the note, and watch it disappear from the homepage, and the browser window which had viewed the note is now on the homepage.

Inter-user chat and commenting for Notes

This is cool, we now have real-time updates in Notes as we edit or delete or create notes. Let's now take it to the next level and implement something akin to inter-user chatting.

It's possible to pivot our Notes application concept and take it in the direction of a social network. In most 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 user name, a time stamp, 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 actions dynamically.

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 the message is sent to, their message, and finally a timestamp for when the message was sent. As messages are received or deleted, events must be emitted from the model so we can do the right thing on the webpage.

This model implementation will be written for Sequelize. If you prefer a different storage solution, you can re-implement the same API on other data storage systems by all means.

Create a new file, `models/messages-sequelize.js`, containing the following:

```
'use strict';

const Sequelize = require("sequelize");
const jsyaml = require('js-yaml');
const fs = require('fs');
const util = require('util');
const EventEmitter = require('events');

class MessagesEmitter extends EventEmitter {}

const log = require('debug')('messages:model-messages');
const error = require('debug')('messages:error');

var SQMessage;
var sequlz;

module.exports = new MessagesEmitter();
```

This sets up the modules being used and also initializes the `EventEmitter` interface. The module itself will be an `EventEmitter` interface because the object is assigned to `module.exports`.

```
var connectDB = function() {
    if (SQMessage) return SQMessage.sync();

    return new Promise((resolve, reject) => {
        fs.readFile(process.env.SEQUELIZE_CONNECT, 'utf8',
        (err, data) => {
            if (err) reject(err);
            else resolve(data);
        });
    })
    .then(yamltext => jsyaml.safeLoad(yamltext, 'utf8'))
    .then(params => {
        if (!sequlz) sequlz = new Sequelize(params.dbname, params.username, params.password, params.params);

        if (!SQMessage) SQMessage = sequlz.define('Message', {
            id: { type: Sequelize.INTEGER, autoIncrement: true,
```

```

        primaryKey: true },
    from: Sequelize.STRING,
    namespace: Sequelize.STRING,
    message: Sequelize.STRING(1024),
    timestamp: Sequelize.DATE
}) ;
return SQMessage.sync();
});
};

This defines our message schema in the database. We'll use the same database that we used for Notes, but the messages will be stored in their own table.
```

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:

```

module.exports.postMessage = function(from, namespace, message) {
    return connectDB()
        .then(SQMessage => SQMessage.create({
            from, namespace, message, timestamp: new Date()
        }))
        .then(newmsg => {
            var toEmit = {
                id: newmsg.id,
                from: newmsg.from,
                namespace: newmsg.namespace,
                message: newmsg.message,
                timestamp: newmsg.timestamp
            };
            module.exports.emit('newmessage', toEmit);
        });
};

This is to be called when a user posts a new comment/message. We first store it in the database, and then we emit an event saying the message was created:
```

```

module.exports.destroyMessage = function(id, namespace) {
    return connectDB()
        .then(SQMessage => SQMessage.find({ where: { id } }))
        .then(msg => msg.destroy())
        .then(result => {
            module.exports.emit('destroymessage', { id, namespace });
        });
};

[ 251 ]
```

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. Once that's done, we emit a message saying the message was destroyed:

```
module.exports.recentMessages = function(namespace) {
    return connectDB().then(SQMessage => {
        return SQMessage.findAll({
            where: { namespace },
            order: 'timestamp DESC',
            limit: 20
        });
    })
    .then(messages => {
        return messages.map(message => {
            return {
                id: message.id,
                from: message.from,
                namespace: message.namespace,
                message: message.message,
                timestamp: message.timestamp
            };
        });
    });
};
```

While this is meant to be called when viewing a note, it is generalized to work for any Socket.IO namespace. It finds the most recent 20 messages associated with the given namespace and returns a cleaned up list to the caller.

Adding messages to the Notes router

Now that we can store messages in the database, let's integrate this into the Notes router module.

In `routes/notes.js`, add this to the require statements:

```
const messagesModel = require('../models/messages-sequelize');
```

If you wish to implement a different data storage model for messages, you'll need to change this `require` statement. One should consider using an environment variable to specify the module name, as we've done elsewhere:

```
// Save incoming message to message pool, then broadcast it
router.post('/make-comment', usersRouter.ensureAuthenticated,
(req, res, next) => {
```

```

messagesModel.postMessage(req.body.from,
    req.body.namespace, req.body.message)
.then(results => { res.status(200).json({}); })
.catch(err => { res.status(500).end(err.stack); });
};

// Delete the indicated message
router.post('/del-message', usersRouter.ensureAuthenticated,
(req, res, next) => {
    messagesModel.destroyMessage(req.body.id, req.body.namespace)
    .then(results => { res.status(200).json({}); })
    .catch(err => { res.status(500).end(err.stack); });
});

```

This pair of routes, `/notes/make-comment` and `/notes/del-message`, is used to post a new comment or delete an existing one. Each calls the corresponding data model function and then sends an appropriate response back to the caller.

Remember that `postMessage` stores a message in the database, and then it turns around and emits that message to other browsers. Likewise, `destroyMessage` deletes the message from the database, then emits a message to other browsers saying that the message has been deleted. Finally, the results from `recentMessages` will reflect the current set of messages in the database.

Both of these will be called by AJAX code in the browser.

```

module.exports.socketio = function(io) {
    var nspView = io.of('/note/view');
    nspView.on('connection', function(socket) {
        // 'cb' is a function sent from the browser, to which we
        // send the messages for the named note.
        socket.on('getnotemessages', (namespace, cb) => {
            messagesModel.recentMessages(namespace)
            .then(cb)
            .catch(err => console.error(err.stack));
        });
    });
    ..

    messagesModel.on('newmessage', newmsg => {
        forNoteViewClients(socket => {
            socket.emit('newmessage', newmsg);
        });
    });
    messagesModel.on('destroymessage', data => {
        forNoteViewClients(socket => {

```

```
        socket.emit('destroymessage', data);
    });
});
..;
};
```

This is the Socket.IO glue code, which we will add to the code we looked at earlier.

The `getnotemessages` message from the browser requests the list of messages for the given note. This calls the `recentMessages` function in the model. This uses a feature of Socket.IO where the client can pass a callback function, and server-side Socket.IO code can invoke that callback giving it some data.

We also listen to the `newmessage` and `destroymessage` messages emitted by the messages model, sending corresponding messages to the browser. These are sent using the method described earlier.

Changing the note view template for messages

We need to dive back into `views/noterview.ejs` 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, several AJAX calls to communicate with the server, and of course, more Socket.IO stuff.

Using a Modal window to compose messages

The Bootstrap framework has a Modal component that serves a similar purpose to Modal dialogs in desktop applications. You pop up the Modal, it prevents interaction with other parts of the webpage, you enter stuff into fields in the Modal, and then click a button to make it close.

This new segment of code replaces the existing segment defining the **Edit** and **Delete** buttons:

```
<% if (notekey && user) { %>
<div class="row">
<div class="btn-group col-sm-12">
  <a class="btn btn-default" href="/notes/destroy?key=<%= notekey %>" role="button">Delete</a>
  <a class="btn btn-default" href="/notes/edit?key=<%= notekey %>" role="button">Edit</a>
  <button type="button" class="btn btn-default"
    data-toggle="modal"
```

```
        data-target=".notes-comment-modal">Comment</button>
    </div>
</div>

<div class="modal fade notes-comment-modal" tabindex="-1"
role="dialog" aria-labelledby="noteCommentModalLabel">
    <div class="modal-dialog modal-lg">
        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-
dismiss="modal" aria-label="Close">
                    <span aria-hidden="true">&times;</span>
                </button>
                <h4 class="modal-title" id="noteCommentModalLabel">Lea
ve a Comment</h4>
            </div>
            <div class="modal-body">
                <form id="submit-comment" class="well"
                    data-async data-target="#rating-modal"
                    action="/notes/make-comment" method="POST">
                    <input type="hidden" name="from"
                        value="<% user.id %>">
                    <input type="hidden" name="namespace"
                        value="/view-<% notekey %>">
                    <input type="hidden" name="key"
                        value="<% notekey %>">
                    <fieldset>
                        <div class="form-group">
                            <label for="noteCommentTextArea">
                                Your Excellent Thoughts, Please</label>
                            <textarea id="noteCommentTextArea" name="message"
                                class="form-control" rows="3"></textarea>
                        </div>

                        <div class="form-group">
                            <div class="col-sm-offset-2 col-sm-10">
                                <button id="submitNewComment" type="submit"
                                    class="btn btn-default">Make Comment</button>
                            </div>
                        </div>
                    </fieldset>
                </form>
            </div>
        </div>
    </div>
<% } %>
```

This adds support for posting comments on a note. The user will see a Modal pop-up window in which they write their comment.

We added a new button labeled **Comment** that the user will click to start the process of posting a message. This button is connected to the Modal by way of the class name specified in the `data-target` attribute. Note that the class name matches one of the class names on the outermost `div` wrapping the Modal. This structure of `div` elements and class names are from the Bootstrap website at <http://getbootstrap.com/javascript/#modals>.

The key portion of this is the HTML form contained within the `div.modal-body` element. It's a straightforward normal Bootstrap augmented form with a normal **Submit** button at the bottom. A few hidden `input` elements are used to pass extra information inside the request.

With the HTML set up this way, Bootstrap will ensure that this Modal is triggered when the user clicks on the **Comment** button. The user can close the Modal by clicking on the **Close** button. Otherwise, it's up to us to implement code to handle the form submission using AJAX so that it doesn't cause the page to reload:

```
<% if (user) { %>
<div id="noteMessages" style="display: none"></div>
<% } %>
<% include footer %>
```

This gives us a place to display the messages.

Sending, displaying, and deleting messages

Note that these code snippets are wrapped with `if` statements. The effect is so that certain user interface elements are displayed only to sufficiently privileged users. A user that isn't logged in perhaps shouldn't see the messages, and they certainly shouldn't be able to post a message.

Now we have a lot of code to add to this block:

```
<% if (notekey) { %>
<script src="/socket.io/socket.io.js"></script>
<script>
$(document).ready(function () { .. });
</script>
```

We need to handle the form submission for posting a new comment, get the recent messages when first viewing a note, listen for events from the server about new messages or deleted messages, render the messages on the screen, and handle requests to delete a message:

```

$(document).ready(function () {
    io('/view').on('noteupdate', function(note) { .. });
    io('/view').on('notedestroy', function(data) { .. });
    <% if (user) { %>
        // Request the recent list of messages
        io('/view').emit('getnotemessages', '/view-<%= notekey %>',
        function(msgs) {
            console.log("RECEIVE getnotemessages reply");
            $('#noteMessages').empty();
            if (msgs.length > 0) {
                msgs.forEach(function(newmsg) {
                    $('#noteMessages').append(formatMessage(newmsg));
                });
                $('#noteMessages').show();
                connectMsgDelButton();
            } else $('#noteMessages').hide();
        });
        // Handler for the .message-del-button's
        var connectMsgDelButton = function() {
            $('.message-del-button').on('click', function(event) {
                $.post('/notes/del-message', {
                    id: $(this).data("id"),
                    namespace: $(this).data("namespace")
                },
                function(response) { });
                event.preventDefault();
            });
        };
        // Emit the code to show a message, and the
        // buttons that will sit next to it.
        var formatMessage = function(newmsg) {
            return '<p id="note-message-' +
                newmsg.id +'" class="well"><strong>' +
                newmsg.from +'</strong>: ' +
                newmsg.message +
                '<small style="float: right">' +
                newmsg.timestamp +'</small>' +
                '<button style="float: right" type="button" class="btn
                btn-primary btn-xs message-del-button" data-id="' +

```

```
+ newmsg.id +'" data-namespace=''
+ newmsg.namespace +'>Delete</button></p>';
};

// Act on newmessage and destroymessage events
io('/view').on('newmessage', function(newmsg) {
    if (newmsg.namespace === '/view-<%= notekey %>') {
        $('#noteMessages').prepend(formatMessage(newmsg));
        connectMsgDelButton();
    }
});

io('/view').on('destroymessage', function(data) {
    if (data.namespace === '/view-<%= notekey %>') {
        $('#noteMessages #note-message-' + data.id).remove();
    }
});

// Handle form submission for the comment form
$('form#submit-comment').submit(function(event) {
    // Abort any pending request
    if (request) { request.abort(); }

    var $form = $('form#submit-comment');
    var $target = $($form.attr('data-target'));

    var request = $.ajax({
        type: $form.attr('method'),
        url: $form.attr('action'),
        data: $form.serialize()
    });

    request.done(function (response, textStatus, jqXHR) { });
    request.fail(function (jqXHR, textStatus, errorThrown) {
        alert("ERROR " + jqXHR.responseText);
    });
    request.always(function () {
        $('.notes-comment-modal').modal('hide');
    });

    event.preventDefault();
});
<% } %>
});

});
```

The code within `$('form#submit-comment').submit` handles the form submission for the comment form. Because we already have jQuery available, we can use its AJAX support to POST a request to the server without causing a page reload.

Using `event.preventDefault`, we ensure that the default action does not occur. For the FORM submission, that means the browser page does not reload. What happens is an HTTP POST is sent to `/notes/make-comment` with a data payload consisting of the values of the form's input elements. Included in those values are three hidden inputs, `from`, `namespace`, and `key`, providing useful identification data.

If you refer to the `/notes/make-comment` route definition, this calls `messagesModel.postMessage` to store the message in the database. That function then posts an event, `newmessage`, which our server-side code forwards to any browser that's connected to the namespace. Shortly, we'll show the response to that event.

When the page is first loaded, we want to retrieve the current messages. This is kicked off with `io('/view').emit('getnotemessages', ...)`. This function, as the name implies, sends a `getnotemessages` message to the server. We showed the implementation of the server-side handler for this message earlier, in `routes/notes.js`.

If you remember, we said that Socket.IO supports providing a callback function that is called by the server in response to an event. You simply pass a function as the last parameter to a `.emit` call. That function is made available at the other end of the communication to be called when appropriate. To make this clear, we have a callback function on the browser being invoked by server-side code.

In this case, our callback function takes a list of messages and displays them on the screen. It uses jQuery DOM manipulation to erase any existing messages, then render each message into the messages area using the `formatMessage` function. The message display area (`#noteMessages`) is also hidden or shown depending on whether we have messages to display.

The message display template, in `formatMessage`, is straightforward. It uses a Bootstrap well to give a nice visual effect. And, there is a button for deleting messages.

In `formatMessage` we created a **Delete** button for each message. Those buttons need an event handler, and the event handler is set up using the `connectMsgDelButton` function. In this case, we send an HTTP POST request to `/notes/del-message`. We again use the jQuery AJAX support again to post that HTTP request.

The `/notes/del-message` route in turn calls `messagesModel.destroyMessage` to do the deed. That function then emits an event, `destroymessage`, which gets sent back to the browser. As you see here, the `destroymessage` event handler causes the corresponding message to be removed using jQuery DOM manipulation. We were careful to add an `id=` attribute to every message to make removal easy.

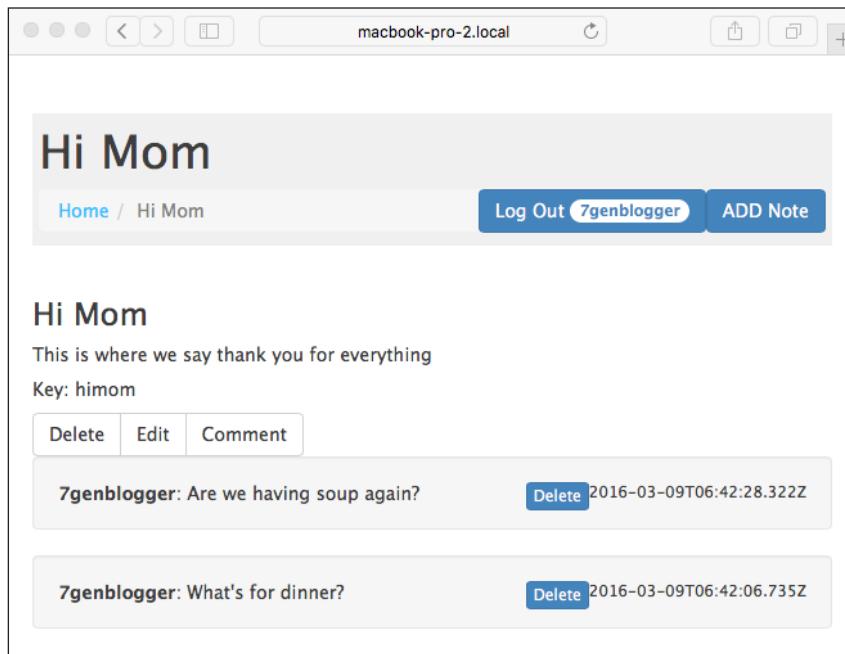
Since the flip side of destruction is creation, we need to have the `newmessage` event handler sitting next to the `destroymessage` event handler. It also uses jQuery DOM manipulation to insert the new message into the `#noteMessages` area.

Running Notes and passing messages

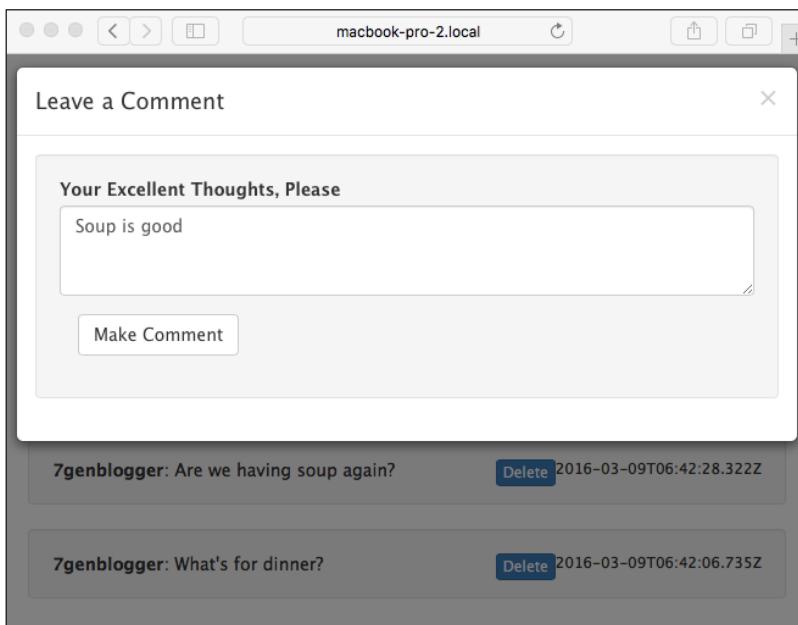
That was a lot of code, but we now have the ability to compose messages, display them on the screen, and delete them, all with no page reloads. Watch out Facebook! We're coming for you now that the Notes application is becoming a social network!

Okay, maybe not. We do need to see what this can do.

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:



This is what a note might look like after entering a few comments.



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.

Other applications of Modal windows

We used a Modal and some AJAX code to avoid one page reload due to a form submission. In the Notes application, as it stands, a similar technique could be used when creating a new note, editing existing notes, and deleting existing notes. In each case, we would use a Modal, some AJAX code to handle the form submission, and some jQuery code to update the page without causing a reload.

But wait, that's not all. Consider the sort of dynamic real-time user interface wizardry on the popular social networks. Imagine what events and/or AJAX calls are required.

When you click on an image in Twitter, it pops up, you guessed it, a Modal window to show a larger version of the image. The Twitter **Compose new Tweet** window is also a Modal window. Facebook uses many different Modal windows, such as when sharing a post, reporting a spam post, or while doing a lot of other things Facebook's designers deem to require a pop-up window.

Socket.IO, as we've seen, gives us a rich foundation of event passing between server and client, which can build multiuser multichannel communication experiences for your users.

Summary

While we came a long way in this chapter, maybe Facebook doesn't have anything to fear from the baby steps we took toward converting the Notes application into a social network. This chapter gave us the opportunity to explore some really cool technology for pseudo-real-time communication between browser sessions.

In this chapter, you learned about using Socket.IO for pseudo-real-time web experiences, using the `EventEmitter` class to send messages between parts of an application, jQuery, AJAX, and other browser-side JavaScript technologies, and avoiding page reloads while making AJAX calls.

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.

10

Deploying Node.js Applications

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 get input and advice from others, Notes needs to escape our laptop and live on a real server. Therefore, the goal of this chapter to look at a couple of deployment methods for Node.js applications.

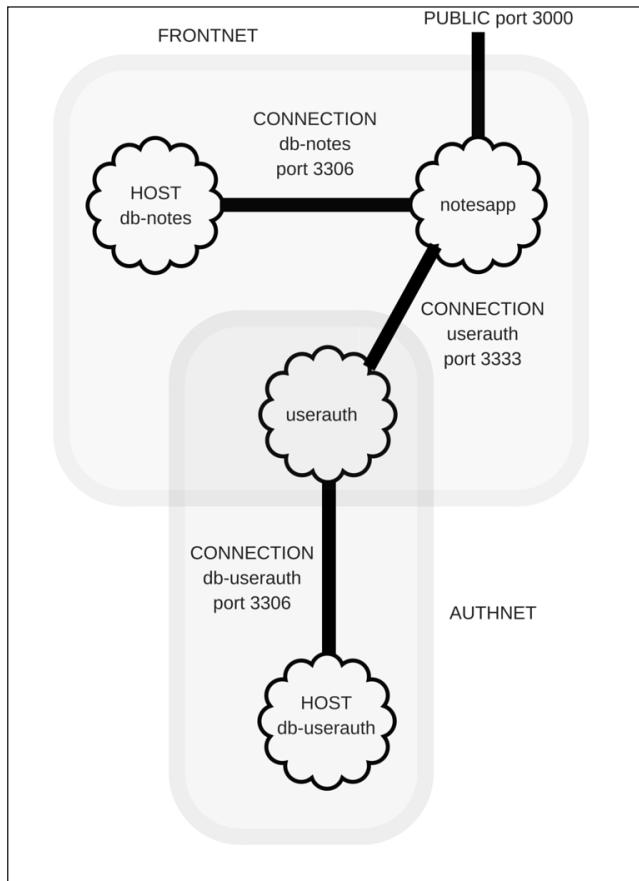
In this chapter, we will cover the following topics:

- Traditional LSB-compliant Node.js deployment
- Using PM2 to improve reliability
- Deployment to a Virtual Private Server provider
- Microservice deployment with Docker (we have four distinct services to deploy)
- Deployment to a Docker hosting provider

The first task is to duplicate the source code from the previous chapter. It's suggested to create a new directory, `chap10`, as a sibling of the `chap09` directory and copy everything from `chap09` to `chap10`.

Notes application architecture

Before we get into deploying the Notes application, we need to review its architecture. We put together several pieces in the last several chapters without giving ourselves the chance to step back a few feet and gather some perspective on the results. To deploy the Notes application, we must understand what we're planning to do:



It's been our goal to segment the services into two groups. The user authentication server should be the more secure portion of the system so that we can earn the trust of our users. On our laptop, we weren't able to create the envisioned protective wall around that service, but we're about to implement such protection.

One strategy to enhance security is to expose as few ports as possible to the outside. That reduces the avenues miscreants can probe for vulnerabilities, thus 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, the two for MySQL servers and the user authentication service port, should be hidden from public view.

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. As currently envisaged, no service outside the Notes application requires access to either database or the authentication service.

Implementation of this segmentation requires either two or three subnets, depending on the lengths you wish to go. The first subnet, FrontNet, contains the Notes application and its database. The second subnet, AuthNet, contains the authentication service and its database. The subnets must be configured to limit the hosts that can access that subnet. It's this subnet configuration that creates the security wall that will let us breathe a little easier because miscreants will have a much harder time doing any damage.

But if the authentication service and Notes application are on different subnets, they will be unable to talk to each other. One could create a third subnet, NotesAuthConnector maybe, so that those services can talk to each other. Or one could do as shown here and allow the authentication service to connect to both FrontNet and AuthNet.

That picture is the desired end result of the chapter. First, we need to get our feet wet with traditional deployment methodologies.

Traditional Linux Node.js service deployment

In a normal server, application deployment on Linux, and other Unix-like systems, is to write an **init script** to manage any needed daemon processes. The required daemons are to start every time the system boots and cleanly shut down when the system is halted. While it's a simple model, the specifics of this vary widely from one operating system (OS) to another.

A common method is for the **init** daemon to manage background processes using shell scripts in the `/etc/init.d` directory. Other OSes use other daemon managers such as `upstart` or `launchd`.

The Node.js project itself does not include any scripts to manage server processes on any OS. Node.js is more like a construction kit with the pieces and parts to build servers and is not a complete polished server framework itself. Implementing a complete web service based on Node.js means creating the scripting to integrate with daemon process management on your OS. It's up to us to develop those scripts.

Web services have to be:

- Reliable, for example, to auto-restart when the server process crashes
- Manageable, meaning it integrates well with system management practices
- Observable, meaning the administrator must be able to get status and activity information from the service

To demonstrate a little of what's involved, let's use PM2 along with an LSB-style init script (<http://wiki.debian.org/LSBInitScripts>) to implement background server process management for the Notes application. These scripts are pretty simple; they take a command-line argument saying whether the service should start or stop the service and do whatever is necessary to do so. Because LSB-style init scripts are not used on all Linux systems, you'll need to adjust the scripts shown in this chapter to match the specific needs of your system. Fortunately, PM2 makes that easy for us.

PM2 generates scripts for a long list of OSes. Additionally, PM2 will handle bringing a multiprocess service up and down for us. Since we have two Node.js processes to administer, it will be a big help.

For this deployment, we'll set up a single Ubuntu 15.10 server. You should secure a Virtual Private Service (**VPS**) from a hosting provider and do all installation and configuration there. Renting a small machine instance from one of the major providers for the time needed to go through this chapter will only cost a couple of dollars.

You can also do the tasks in this section using **VirtualBox** on your laptop. Simply install Debian or Ubuntu and pretend that it's hosted on a remote VPS hosting provider.

Both the Notes and User Authentication services will be on that server, along with a single MySQL instance. While our goal is a strong separation between FrontNet and AuthNet, with two MySQL instances, we won't do so at this time.

Prerequisite – provisioning the databases

The Linux package management systems don't allow us to install two MySQL instances. We can implement some separation in the same MySQL instance by using separate databases with different usernames and access privileges for each database.

The first step is to ensure that MySQL is installed on your server. For Ubuntu, **Digital Ocean** has a fairly good tutorial: <https://www.digitalocean.com/community/tutorials/how-to-install-mysql-on-ubuntu-14-04>.

The MySQL server must support TCP connections from localhost. Edit the configuration file, `/etc/mysql/my.cnf`, to have the following line:

```
bind-address = 127.0.0.1
```

This limits MySQL server connections to the processes on the server. A miscreant would have to break into the server to access your database.

Now that our database server is available, let's set up two databases.

In the `chap10/notes/models` directory, create a file named `mysql-create-db.sql` containing the following:

```
CREATE DATABASE notes;
CREATE USER 'notes'@'localhost' IDENTIFIED BY 'notes';
GRANT ALL PRIVILEGES ON notes.* TO 'notes'@'localhost'
    WITH GRANT OPTION;
```

And in the `chap10/users` directory, create a file named `mysql-create-db.sql` containing the following:

```
CREATE DATABASE userauth;
CREATE USER 'userauth'@'localhost' IDENTIFIED BY 'userauth';
GRANT ALL PRIVILEGES ON userauth.* TO 'userauth'@'localhost'
    WITH GRANT OPTION;
```

When the Notes application source code is copied to the server, we'll run the scripts as:

```
$ mysql -u root -p <chap10/users/mysql-create-db.sql
$ mysql -u root -p <chap10/notes/models/mysql-create-db.sql
```

This will create the two databases, `notes` and `userauth`, with associated usernames and passwords. Each user can access only its associated database. Later we'll set up Notes and the User Authentication service with YAML configuration files to access these databases.

Installing Node.js on Ubuntu

According to the Node.js documentation (<https://nodejs.org/en/download/package-manager/>), the recommended installation method for Debian or Ubuntu Linux distributions is the following:

```
$ curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash -
$ sudo apt-get install -y nodejs
$ sudo apt-get install -y build-essential
```

Installing this way means that as new Node.js releases are issued, upgrades are easily accomplished with the normal package management procedures. The `build-essential` package brings in all the compilers and other development tools required to build native code Node.js packages.

Setting up Notes and User Authentication on the server

Before copying the Notes and User Authentication code to this server, let's do a little coding to prepare for the move. We know that the Notes and Authentication services must access the MySQL instance on `localhost` using the usernames and passwords given earlier.

Using the approach we've followed so far, this means a pair of YAML files for `Sequelize` parameters, and changing environment variables in the `package.json` files to match.

Create a `chap10/notes/models/sequelize-server-mysql.yaml` file containing:

```
dbname: notes
username: notes
password: notes
params:
  host: localhost
  port: 3306
  dialect: mysql
```

In `chap10/notes/package.json`, add the following line to the `scripts` section:

```
"on-server": "SEQUELIZE_CONNECT=models/sequelize-server-mysql.yaml
NOTES_MODEL=models/notes-sequelize USERS_MODEL=models/users-rest USER_
SERVICE_URL=http://localhost:3333 PORT=3000 node ./app",
```

Then create a `chap10/users/sequelize-server-mysql.yaml` file containing:

```
dbname: userauth
username: userauth
password: userauth
params:
  host: localhost
  port: 3306
  dialect: mysql
```

And in `chap10/users/package.json`, add the following line to the `scripts` section:

```
"on-server": "PORT=3333 SEQUELIZE_CONNECT=sequelize-server-mysql.yaml
node ./user-server",
```

This configures the authentication service to access the databases just created.

Now we need to select a place on the server to install the application code:

```
# ls /opt
```

This empty directory looks to be as good a place as any. Simply upload `chap10/notes` and `chap10/users` to your preferred location, so you have the following:

```
# ls /opt
notes  users
```

Then, in each directory, run these commands:

```
# rm -rf node_modules
# npm install
```

Note that we're running these commands as `root` rather than a user ID that can use the `sudo` command. The machine offered by the hosting provider is configured to be used in this way. Other VPS hosting providers will provide machines where you log in as a regular user and then use `sudo` to perform privileged operations. As you read these instructions, pay attention to the command prompt we show. We've followed the convention where `$` is used for commands run as a regular user and `#` is used for commands run as `root`. If you're running as a regular user, and need to run a `root` command, then run the command with `sudo`.



When uploading the source to the server, you may have included the `node_modules` directory. That directory was built on your laptop and probably isn't suitable for the server you're running. It's unlikely your laptop is running the same OS as the server, so any native code modules will fail. The `node_modules` directory should be created on the target machine so that native code modules are compiled on the target machine.

The simplest method to do so is to just delete the whole `node_modules` directory and then let `npm install` do its job.

Remember that we set up the `PATH` environment variable the following way:

```
# export PATH=./node_modules/.bin:${PATH}
```

Also remember that in the `chap10/notes` directory, the `npm install` step will run `bower install`. This is likely to fail because `bower` insists on not being run with root privileges, for good reasons. If necessary, you can run `bower` the following way:

```
# apt-get install git  
# bower --allow-root install
```

It's useful to record this in the package `.json` script as well:

```
"postinstall": "bower --allow-root install"
```

Finally, at this time, you can now run the SQL scripts written earlier to set up the database instances:

```
# mysql -u root -p <users/mysql-create-db.sql  
# mysql -u root -p <notes/models/mysql-create-db.sql
```

Then you should be able to start up the services by hand to check that everything is working correctly. The MySQL instance has already been tested, so we just need to start the User Authentication and Notes services:

```
# cd /opt/users  
# npm run on-server  
  
> users@0.0.1 on-server /opt/users  
> PORT=3333 SEQUELIZE_CONNECT=sequelize-server-mysql.yaml node ./user-server
```

Then log in to the server on another terminal session and run the following:

```
# cd /opt/users/
# PORT=3333 node users-add.js
Created { id: 1,
  username: 'me', password: 'w0rd', provider: 'local',
  familyName: 'Einarrsdottir', givenName: 'Ashildr',
  middleName: '', emails: '[]', photos: '[]',
  updatedAt: '2016-03-22T01:28:21.000Z',
  createdAt: '2016-03-22T01:28:21.000Z' }

# PORT=3333 node users-list.js
List [ { id: 'me', username: 'me', provider: 'local',
  familyName: 'Einarrsdottir', givenName: 'Ashildr', middleName: '',
  emails: '[]', photos: '[]' } ]
```

The preceding command both tests the backend user authentication service is functioning and gives us a user account we can use to log in. The `users-list` command demonstrates that it works.

Now we can start the Notes service:

```
# cd ../notes
# npm run on-server

> notes@0.0.0 on-server /opt/notes
> SEQUELIZE_CONNECT=models/sequelize-server-mysql.yaml NOTES_
MODEL=models/notes-sequelize USERS_MODEL=models/users-rest USER_SERVICE_
URL=http://localhost:3333 PORT=3000 node ./app
```

And then we can use our web browser to connect to the application. Since you probably do not have a domain name associated with this server, Notes can be accessed via the IP address of the server, such as: `http://104.131.55.236:3000`.

The Twitter application we set up for Notes previously won't work because the authentication URL is incorrect for the server. For now, we can log in using the user profile created previously.

By now you know that the drill of verifying Notes is working. Create a few notes, open a few browser windows, see that real-time notifications work, and so on. Once you're satisfied that Notes is working on the server, kill the processes and move on to the next section, where we'll set this up to run when the server starts.

Setting up PM2 to manage Node.js processes

There are many ways to manage server processes, to ensure restarts if the process crashes, and so on. **Supervisord** (<http://supervisord.org/>) is a likely candidate for this purpose. However, we'll instead use **PM2** (<http://pm2.keymetrics.io/>) because it's optimized for Node.js processes. It bundles process management and monitoring into one application whose purpose is managing background processes.

Install it as so (using sudo if needed):

```
# npm install -g pm2@1.x
```

In `users/package.json`, we can add the following line to the `scripts` section:

```
"on-server-pm2": "PORT=3333 SEQUELIZE_CONNECT=sequelize-server-mysql.yaml pm2 start --env PORT --env SEQUELIZE_CONNECT user-server.js",
```

In normal `pm2` usage, we launch scripts with `pm2 start script-name.js`.

Because our services use environment variables for configuration, we have to add the `--env` options so that PM2 knows it should pay attention to these variables. We could go ahead and add a similar line in `notes/package.json`, but let's try something else instead. PM2 offers a better way for us to describe the processes it should start.

In the `/opt` directory, create a file named `ecosystem.json` containing the following:

```
{
  apps : [
    {
      name      : "User Authentication",
      script    : "user-server.js",
      "cwd"     : "/opt/users",
      env: {
        PORT: "3333",
        SEQUELIZE_CONNECT: "sequelize-server-mysql.yaml"
      },
      env_production : {
        NODE_ENV: "production"
      }
    },
    {
      name      : "Notes",
      script    : "app.js",
      "cwd"     : "/opt/notes",
      env: {
```

```

    PORT: "3000",
    SEQUELIZE_CONNECT: "models/sequelize-server-mysql.yaml",
    NOTES_MODEL: "models/notes-sequelize",
    USERS_MODEL: "models/users-rest",
    USER_SERVICE_URL: "http://localhost:3333"
  },
  env_production : {
    NODE_ENV: "production"
  }
}
]
}

```

This file describes the directories containing both services, the script to run each service, and the environment variables to use.

We then start the services as so: pm2 start ecosystem.json, which looks like the following on the screen:

```

[root@ubuntu-2gb-nyc3-01:/opt# pm2 start ecosystem.json
[PM2] Process launched
[PM2] Process launched



| App name                     | id     | mode         | pid            | status           | restart | uptime   | memory                 | watching             |
|------------------------------|--------|--------------|----------------|------------------|---------|----------|------------------------|----------------------|
| User Authentication<br>Notes | 0<br>1 | fork<br>fork | 11840<br>11845 | online<br>online | 0<br>0  | 0s<br>0s | 21.313 MB<br>17.648 MB | disabled<br>disabled |



Use `pm2 show <id|name>` to get more details about an app
[root@ubuntu-2gb-nyc3-01:/opt# pm2 status


| App name                     | id     | mode         | pid            | status           | restart | uptime       | memory                 | watching             |
|------------------------------|--------|--------------|----------------|------------------|---------|--------------|------------------------|----------------------|
| User Authentication<br>Notes | 0<br>1 | fork<br>fork | 11840<br>11845 | online<br>online | 0<br>0  | 103s<br>103s | 44.453 MB<br>59.113 MB | disabled<br>disabled |



Use `pm2 show <id|name>` to get more details about an app

```

You can again navigate your browser to the URL for your server, such as <http://104.131.55.236:3000>, and check that the Notes application is working.

Once started, some useful commands are as follows:

```
# pm2 list
# pm2 describe 1
# pm2 logs 1
```

These commands let you query the status of the services.

The pm2 monit command gives you a pseudo-graphical monitor of system activity, as shown in the following screenshot:



As you access Notes, watch how those red bars grow and shrink showing the changes in CPU consumption. This is all cool, but if we restart the server, these processes don't start with the server. How do we handle that? It's very simple because PM2 can generate an init script for us:

```
# pm2 startup ubuntu
[PM2] Generating system init script in /etc/init.d/pm2-init.sh
[PM2] Making script booting at startup...
[PM2] -ubuntu- Using the command:
      su -c "chmod +x /etc/init.d/pm2-init.sh && update-rc.d pm2-init.sh
defaults"
[PM2] Done.
```

Because we're using an Ubuntu server, we generated an Ubuntu startup script. Make sure to generate this script for the platform (**Gentoo** and so on) you are using. If you simply leave off the platform name, PM2 will autodetect the platform for you.

The script generated doesn't include specifics about the currently executing set of processes. Instead, it simply runs some of the PM2 commands needed to start or stop a list of processes it has been told to manage. PM2 keeps its own data about the processes you've asked it to manage.

It is necessary to run the following command:

```
# pm2 save
```

This saves the current process state so that the processes can be restarted later, such as when the system restarts.

Twitter support for the hosted Notes app

The only thing we're lacking is support for logging in with Twitter. When we registered the Notes application with Twitter, we registered a domain such as `http://MacBook-Pro-2.local:3000` for our laptop. The Notes application is now deployed on the server and is used by a URL that Twitter doesn't recognize. We therefore have a couple of things to change.

On `apps.twitter.com` in the Notes application registration screen, we can enter the IP address of the website as so: `http://104.131.55.236:3000`.

Then, in `notes/routes/users.js`, we need to make a corresponding change to the `TwitterStrategy` definition:

```
passport.use(new TwitterStrategy({
    consumerKey: "... KEY",
    consumerSecret: "... SECRET",
    callbackURL:
        "http://104.131.55.236:3000/users/auth/twitter/callback"
},
) );
```

Upon making those changes, restart the services:

```
root@ubuntu-2gb-nyc3-01:/opt/notes# /etc/init.d/pm2-init.sh restart
Restarting pm2
[PM2] Deleting all process
[PM2] deleteProcessId process id 0
[PM2] deleteProcessId process id 1
```

App name	id	mode	pid	status	restart	uptime	memory	watching
----------	----	------	-----	--------	---------	--------	--------	----------

```
Use `pm2 show <id|name>` to get more details about an app
```

```
[PM2] Stopping PM2...
[PM2] [WARN] No process found
[PM2] All processes have been stopped and deleted
[PM2] PM2 stopped
```

```
Starting pm2
[PM2] Spawning PM2 daemon
[PM2] PM2 Successfully daemonized
[PM2] Resurrecting
Process /opt/users/user-server.js launched
Process /opt/notes/app.js launched
```

App name	id	mode	pid	status	restart	uptime	memory	watching
User Authentication Notes	0 1	fork fork	1301 1306	online online	0 0	0s 0s	23.031 MB 21.809 MB	disabled disabled

```
Use `pm2 show <id|name>` to get more details about an app
```

And then you'll be able to log in using your Twitter account.

We now have the `Notes` application under a fairly good management system. We can easily update its code on the server and restart the service. If the service crashes, PM2 will automatically restart it. Log files are automatically kept for our perusal.

PM2 also supports deployment from the source on our laptop, which we can push to staging or production environments. To support this, we must add deployment information to the `ecosystem.json` file and then run the `pm2 deploy` command to push the code to the server. See the PM2 website for more information.

While PM2 does a good job at managing server processes, the system we've developed is insufficient for an *Internet-scale* service. What if the `Notes` application were to become a viral hit and suddenly we need to deploy a million servers spread around the planet? Deploying and maintaining servers one at a time, like this, is not scalable.

We also skipped over implementing the architectural decisions at the beginning. Putting the user authentication data on the same server is a security risk. We want to deploy that data on a different server, under tighter security.

In the next section, we'll explore a new system, **Docker**, that solves these problems and more.

Node.js microservice deployment with Docker

Docker (<http://docker.com>) is the new attraction in the software industry. Interest is taking off like crazy, spawning many projects, many with names containing puns around shipping containers.

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.

Docker automates the application deployment within software containers. The basic concepts of Linux containers date back to `chroot` jail's first implemented in the 1970s, and other systems like Solaris Zones. The Docker implementation creates a layer of software isolation and virtualization based on Linux cgroups, kernel namespaces, and union-capable filesystems, which blend together to make Docker what it is. That was some heavy geek-speak, so let's try a simpler explanation. A Docker container is a running instantiation of a Docker image. An image is a given Linux OS and application configuration designed by developers for whatever purpose they have in mind. Developers describe an image using a **Dockerfile**. The Dockerfile is a fairly simple-to-write script showing Docker how to build an image.

A running container will make you feel like you're inside a virtual server running on a virtual machine. But Docker containerization is very different from a virtual machine system such as VirtualBox. The processes running inside the container are actually running on the host OS. The containerization technology (cgroups, kernel namespaces, and so on) create the illusion of running on the Linux variant specified in the Dockerfile, even if the host OS is completely different. Your host OS could be Ubuntu and the container OS could be Fedora or OpenSUSE; Docker makes it all work. The only requirement is that both host OS and container OS must be a Linux variant.

By contrast, with Virtual Machine software (VirtualBox, VMWare, and so on), you're using what feels like a real computer. There is a virtual BIOS and system hardware, and you must install a full-fledged guest OS inside this virtual computer. You must follow every ritual of computer ownership, including securing licenses, if it's a closed source system like Windows. Processes running inside the virtual machine are running solely on the guest OS, and the host OS has no clue what's going on inside that machine.

It means Docker has much lower overhead than a virtual machine. For example, network traffic on Docker is handled by the host OS, whereas in a virtual machine, it has to be translated back and forth between host and guest OS network drivers. The same is true for access to any other resource, such as disk space, USB devices, and so on.

While Docker is primarily targeted at x86 flavors of Linux, some teams support Docker on ARM and other processors. You can even run Docker on single-board computers such as Raspberry Pis for hardware-oriented Internet of Things projects.

The Docker ecosystem contains many tools, and their number is quickly increasing. For our purposes, we'll be focusing on the following three specific 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 Machine:** This is a client application performing functions around provisioning Docker Engine instances on host computers.
- **Docker Compose:** This helps you define, in a single file, a multicontainer application, with all its dependencies defined.

With the Docker ecosystem, you can create a whole universe of subnets and services to implement your dream application. That universe can run on your laptop or be deployed to a globe-spanning network of cloud-hosting facilities around the world. The surface area through which miscreants can attack is small and is strictly defined by the developer. A multicontainer application will even limit access so strongly between services that even miscreants who manage to break into a container will find it difficult to break out of the container.

Using Docker, we'll first design on our laptop the system shown in the earlier diagram. Then we'll migrate that system to a Docker instance on a server.

Installing Docker on your laptop

The best place to learn how to install Docker on your laptop is the Docker documentation website: <https://docs.docker.com/engine/installation/>.

For users on Linux distributions, you can directly use the Docker command-line tools to manage local Docker containers, with no additional configuration required. The Linux Docker tools are easily installed using the package management systems on several Linux distros.

Because Docker runs on Linux, installing it on Mac OS X or Windows requires installing Linux inside a virtual machine and then running Docker tools within that Linux. The days when you had to craft that yourself are long gone. The Docker team has made this easy by developing easy-to-use Docker applications for Mac and Windows. They bundle together Docker tools and virtual machine software, letting you run containers on your laptop and use the Docker command-line tools.

Docker Toolbox bundles together VirtualBox, the full set of Docker command-line tools, and the **Kitematic GUI**. It replaces an earlier application named **boot2docker**. When installed (<https://docs.docker.com/toolbox/overview/>), it automatically installs a VirtualBox instance behind the scenes, along with all the Docker command-line tools set up, so that they easily manage Docker infrastructure inside VirtualBox. With the command-line tools, you can also use any Docker Engine instance anywhere, whether on your laptop or in a server room on the other side of the planet.

Two new applications, Docker for Windows and Docker for Mac, are native Docker implementations for their respective platforms. They integrate more tightly with the platform, making Docker work similarly to other native applications. Most importantly, it does not require VirtualBox, which fixes several problems. It is claimed to have higher performance by using lighter-weight virtual machine technology. It's so lightweight that it's feasible to leave Docker for Windows or Mac running full-time in the background.

As of this writing, Docker for Windows and Docker for Mac are in a limited-access Beta.

After installing Docker or Docker Toolbox or Docker for Windows/Mac and following the instructions for your OS, you can try a couple of commands to get your feet wet.

Starting Docker using Docker Toolbox and Docker Machine

If you installed Docker Toolbox, there is a prerequisite step before you can run Docker commands. That is to use the local Docker Machine tool to create a local Docker instance, and initialize some environment variables so the Docker commands know which Docker Engine to access.

Even if the Docker team ends up discontinuing Docker Toolbox, you must know how to use Docker Machine. That's because Docker Machine is used for installing Docker Engine onto host machine instances, either local or remote, and to manage Docker instances using `docker-machine` command-line operations.

Starting `docker-machine` on your local machine boots up the VirtualBox instance, and as we see in the following, the first time it runs, it provisions the Docker environment inside VirtualBox. This is required when using Docker Toolbox, but not when using Docker for Windows or Mac:

```
$ docker-machine start
starting "default"...
(default) Check network to re-create if needed...
(default) Waiting for an IP...
Machine "default" was started.
Waiting for SSH to be available...
Detecting the provisioner...
```

Started machines may have new IP addresses. You may need to rerun the `docker-machine env` command.

This starts a Docker Engine instance on the localhost, and behind the scenes, Docker is running in a Linux instance inside VirtualBox. If you look at background processes, you'll see that a VirtualBox instance has started:

```
$ docker-machine env
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
```

```
export DOCKER_CERT_PATH="/Users/david/.docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
# Run this command to configure your shell:
# eval $(docker-machine env)
```

These are the Docker environment variables describing how the Docker commands should access the local Docker instance. These variables obviously include a TCP address and a directory full of PEM certificates. It also helpfully suggests an easy way to add these variables to the environment of your shell.

The `docker-machine` command lets us start multiple Docker Engine instances, either on the localhost or, as we'll see later, on remote servers. The `docker-machine create` command is used for creating these instances. Later, we'll see how this command can even reach out to specific cloud-hosting providers and create, from the command line of your laptop, Docker hosts on the cloud.

In the meantime, the following is what we have:

```
MacBook-Pro-2:users david$ eval $(docker-machine env default)
MacBook-Pro-2:users david$ docker-machine ls
NAME      ACTIVE   DRIVER      STATE      URL
default    *        virtualbox  Running    tcp://192.168.99.100:2376
MacBook-Pro-2:users david$
```

The name for this newly created Docker instance is `default`. When creating a Docker host with `docker-machine`, you can specify any name you like.

Because VirtualBox is fairly heavyweight, you probably won't want to leave it running full-time in the background. For that and many other reasons, you may want to shut down a Docker instance. This is done using the following `docker-machine` command:

```
$ docker-machine stop
```

This contacts the Docker instance identified by the `DOCKER` environment variables, sending it a command to shut down containers and the Docker service.

Starting Docker with Docker for Windows/Mac

If you instead installed Docker for Windows or Mac, the startup process is much simpler. You simply find and double-click on the application icon. It launches as would any other native application, and when started, it manages a virtual machine (not VirtualBox) within which is a Linux instance running the Docker Engine. On Mac OS X, a menu bar icon shows up with which you control `Docker.app`, and on Windows, an icon is available in the system tray.

The `docker-machine` command is not used to provision this instance. Instead, you use the `docker` and `docker-compose` command-line tools, and they interact directly with this Docker instance with no configuration required.

Kicking the tires of 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 this amazing system.

As in so many software journeys, this one starts with saying Hello to the world:

```
MacBook-Pro-2:chap10 david$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
03f4658f8b78: Already exists
a3ed95caeb02: Already exists
Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com

For more examples and ideas, visit:
https://docs.docker.com/userguide/
```

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.

You can query your computer to see that while the `hello-world` container has executed and finished, it still exists:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
MacBook-Pro-2:chap10 davids	docker ps					
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
a801788a914a	hello-world	"/hello"	7 minutes ago	Exited (0) 7 minutes ago		hopeful_almeida
61656bb35a11	hello-world	"/hello"	10 minutes ago	Exited (0) 10 minutes ago		amazing_boyd

The `docker ps` command lists the running Docker containers. As we see here, the `hello-world` container is no longer running, but with the `-a` switch, `docker ps` also shows those containers that exist but are not currently running.

When you're done using a container, you can clean up with the following command:

```
$ docker rm amazing_boyd hopeful_almeida  
amazing_boyd  
hopeful_almeida
```

We used these names because these are the container names. While the image name was `hello-world`, that's not the container name. Docker generated these two container names so you have a more user-friendly identifier for the containers than the hex ID shown in the container ID column. When creating a container, it's easy to specify any container name you like.

Creating the AuthNet for the User Authentication service

With all that theory spinning around our heads, it's time to do something practical. Let's start by setting up the User Authentication service. In the diagram shown earlier, this will be the box labeled AuthNet containing a MySQL instance and the authentication server.

MySQL for the Authentication service

Create a directory named `db-auth` as a sibling to the `users` and `notes` directories. In that directory, create a file named `Dockerfile` containing the following:

```
FROM mysql:5.7  
  
ENV MYSQL_RANDOM_ROOT_PASSWORD=yes  
ENV MYSQL_DATABASE=userauth  
ENV MYSQL_USER=userauth  
ENV MYSQL_PASSWORD=userauth
```

```
RUN sed -i "s/^#bind-address.*$/bind-address = 0.0.0.0/" /etc/mysql/my.cnf
RUN sed -i "s/^pid-file/# pid-file/" /etc/mysql/my.cnf
RUN sed -i "s/^socket/# socket/" /etc/mysql/my.cnf

VOLUME /var/lib/mysql

EXPOSE 3306
CMD ["mysqld"]
```

Dockerfiles describe the installation of an application on a server (see <https://docs.docker.com/engine/reference/builder/>). The purpose is to document how to assemble the bits which go into a Docker container image. Docker uses the instructions in a Dockerfile to build a Docker image.

The `FROM` command specifies a pre-existing image from which to derive this image. The `mysql` image (https://hub.docker.com/_/mysql/) used here was created by the MySQL team. While there are plenty of MySQL images available in the Docker Hub, this is the official image. The `:5.7` attribute is a version tag, specifying we'll be using MySQL 5.7. If it's important to you, the `mysql:5.7` image itself is built on the `debian:jessie` image, meaning that we'll ultimately be executing under Debian.

The subsequent commands tailor the MySQL container to be useful for the authentication service. The environment variable definitions (`ENV` command) instruct the MySQL image to create the named user identity, database, and passwords the first time this container is executed. The `RUN sed` commands modify the `my.cnf` file so that we can use a TCP port to communicate with the MySQL instance.

The `VOLUME` command says that the MySQL data directory, `/var/lib/mysql`, will be stored outside the container and then mapped so that it's visible inside the container. Generally speaking, what happens inside a Docker container stays within the Docker container. Normally, that's useful not just for bad puns on lines from movies but also for improved security. Sometimes, however, we need to reuse data generated by a container in other contexts. For example, a database might need to be migrated elsewhere or preserved when the container is deleted.

Docker containers are designed to be cheap and easy to create, and to be thrown away and recreated whenever needed. This includes throwing away all data inside the container. Databases, on the other hand, are meant to live on for years sometimes, making it useful for the database files to exist separately from 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.

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.

Make sure that Docker is running on your machine. If necessary, run the `docker-machine` command or start the Docker for Windows or Mac application.

Now we can build the `db-auth` image as follows:

```
$ cd db-auth  
$ docker build -t node-web-development/db-auth .
```

This builds the image and gives it a full name by which we can refer to this image. The image can even be pushed, under this name, to a Docker image repository and used by others.

And then execute it as so:

```
$ docker run -it --name db-auth node-web-development/db-auth
```

Voluminous information will be printed about the initialization of the database. If you look carefully, the randomly generated password is printed out and you're told that the server is listening to `0.0.0.0` on port `3306`. Normally, this means that this MySQL instance is open to access from the entire Internet, but because it is inside a Docker container, we are in control of where and when that port is visible. At the moment, the port is not published outside the container.

The `-it` option tells Docker to run this in an interactive (`-i`) terminal (`-t`) session. That's why the voluminous output is printed on the terminal.

The `--name db-auth` option assigns a user-friendly name to the container. Otherwise the container is identified by its `sha256` hash, the shortened version of which might be an utterly forgettable identifier like `c1d731d589ae`.

The final argument is the name of the image to execute. The image will be retrieved from a Docker repository or from the local machine.

Once the image is running, we can't do much with it because the MySQL port is invisible outside the container. But we can get into the container and run a MySQL command there. In another command window, run the following:

```
[MacBook-Pro-2:compose david$ docker exec -it db-auth bash
[root@c1d731d589ae:/# mysql -u userauth -p
[Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.11 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| userauth       |
+-----+
2 rows in set (0.00 sec)

mysql> use userauth;
Database changed
mysql> show tables;
Empty set (0.00 sec)

mysql> ]
```

The `exec` command will execute a process inside a running container. Notice that the shell prompt changes from the laptop to root on a host named `c1d731d589ae` because we ran `bash` inside the container. We can then run the MySQL client and examine the database. We could also poke around the filesystem and see what's there.

At the moment, though, we don't need this container image. Let's remove it:

```
$ docker stop db-auth
db-auth
$ docker rm db-auth
db-auth
```

We'll be recreating the `db-auth` container later with some additional options. As implied by the command names, the first stops the running container, while the second deletes the container.

Dockerizing the Authentication service

In the `users` directory, create a file named `Dockerfile` containing the following:

```
FROM node:5.9

ENV DEBUG="users:__":
ENV PORT="3333"
ENV SEQUELIZE_CONNECT="sequelize-docker-mysql.yaml"
ENV REST_LISTEN="0.0.0.0"

RUN mkdir -p /usr/src/app
COPY . /usr/src/app/
WORKDIR /usr/src/app
RUN apt-get update -y \
    && apt-get -y install curl python build-essential git ca-
certificates \
    && npm install --unsafe-perm

EXPOSE 3333
CMD npm run docker
```

We're deriving this image from the official Node.js Docker image maintained by the Node.js team (https://hub.docker.com/_/node/). Like the MySQL container used earlier, this one is also derived from `debian:jessie`.

The authentication service code is copied to the `/usr/src/app` directory of the container. The `build-essential` package and other tools are installed to support building native code Node.js modules.

It's recommended to always combine `apt-get update` with `apt-get install` in the same command line, like this, because of the Docker build cache. When building a Docker image, not all the lines are executed. Instead, the first change in the file is the first line executed. By putting those two together, you ensure that `apt-get update` is executed any time you change the list of packages to be installed. For a complete discussion, see the documentation at https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/.

At the end of this command is `npm install --unsafe-perm`. We obviously need to run `npm install` inside the container so that Node.js dependencies are installed and that native code Node.js modules are compiled for the OS running inside the container. What's the `--unsafe-perm` flag, however?

The issue here is that these commands are being run as `root`. Normally, when `npm` is run as `root`, it changes its user ID to a nonprivileged user. This can cause failure, however, and the `--unsafe-perm` option prevents changing the user ID.

The environment variables used are the same as we used in `package.json`, with two differences.

First, we're using a different file for `SEQUELIZE_CONNECT`. Create a new file named `sequelize-docker-mysql.yaml` containing the following:

```
dbname: userauth
username: userauth
password: userauth
params:
  host: db-auth
  port: 3306
  dialect: mysql
```

We'll explain the hostname in a little bit, and yes it purposely does not have a top-level domain name. The database name, user name, and password must match with what we gave in the `db-auth` Dockerfile.

Second is a new variable named `REST_LISTEN`. Previously, the Authentication Server had listened only to `http://localhost:3333`. We'd done this for security purposes, that is, to limit which processes could connect to the service. Under Docker, we need to connect to this service from outside its container so that other containers can connect to this service. Therefore, it must listen to connections from outside the localhost.

In `users-server.js`, we need to make the following change:

```
server.listen(process.env.PORT,
  process.env.REST_LISTEN ? process.env.REST_LISTEN : "localhost",
  () => { log(server.name + ' listening at ' + server.url); });
```

That is, if the `REST_LISTEN` variable exists, the REST server is told to listen to whatever it says, otherwise the service is to listen to `localhost`. With the environment variable in the Dockerfile, the authentication service will listen to the world (`0.0.0.0`). Are we throwing caution to the wind and abrogating our fiduciary duty in keeping the sacred trust of storing all this user identification information? No. Be patient. We'll describe momentarily how to connect this service and its database to AuthNet and will prevent access to AuthNet by any other process.

In `package.json`, add the following line to the `scripts` section:

```
"docker": "node user-server"
```

Previously, we've put the configuration environment variables into `package.json`. In this case, the configuration environment variables are in the Dockerfile. This means we need a way to run the server with no environment variables other than those in the Dockerfile. With this `scripts` entry, we can do `npm run docker` and then the Dockerfile environment variables will supply all configuration.

We can build the authentication service as so:

```
$ docker build -t node-web-development/userauth .
```

Putting Authnet together

We can't go ahead and run the `userauth` container because we have a couple of missing pieces. For example, the domain name `db-auth` specified in `sequelizer-docker-mysql.yaml` doesn't exist. And we have an authentication service that listens to the whole world when its scope needs to be limited. What we need is to create the `AuthNet` box shown on the earlier diagram.

With Docker, we can create virtual networks with strictly limited access. We can then attach Docker containers to those networks, and Docker even includes an embedded DNS server to set up the hostnames required so containers can find each other.

Type the following:

```
$ docker network create --driver bridge authnet
```

A bridge network is a virtual network that exists solely within the host computer. Docker manages this network, and bridge networks can only be accessed by containers attached to the bridge network. A DNS server is configured for that network, whose job is to publish domain names for the containers. Those domain names match the container name given when creating the container. Containers are meant to find other containers, a.k.a. "service discovery", by using the container name as a domain name.

In older days of Docker, we were told to link containers using the `--link` option. With that option, Docker would create entries in `/etc/hosts` so that one container can refer to another container by its host name. That option also arranged access to TCP ports and volumes between linked containers. This allowed creation of multicontainer services, using private TCP ports for communication that exposed nothing to processes outside the containers. Today, we are told that the `--link` option is a legacy feature, and that instead we should use bridge networks.

This is "service discovery" using host names or domain names. In our case, the service named db-auth contains a MySQL instance, and we desire to use the db-auth service from the userauth service. We'll use AuthNet to connect the two. The containers find each other by looking for hosts with a preconfigured hostname. With a Docker bridge network, the container hostnames are found using the embedded DNS server, while with the earlier --link option, hostnames were found via the /etc/hosts files created by Docker.

How do we attach a container to a bridge network?

```
$ docker run -it --name db-auth --net=authnet \
    node-web-development/db-auth
$ docker run -it --name userauth --net=authnet \
    node-web-development/userauth
```

One way to attach a container to a network is with the --net=authnet option. We'll look at a second method later.

Let's explore what we just created:

```
$ docker network inspect authnet
```

This prints out a large JSON object describing the network, and its attached containers. The db-auth container might be at the 172.19.0.2 IP address. But try pinging that address and we'll see failure:

```
$ ping 172.19.0.2
PING 172.19.0.2 (172.19.0.2): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
```

If we execute a shell in the userauth container, we can now ping the IP address of the db-auth container:

```
$ docker exec -it userauth bash
root@1f2e3de3be6d:/usr/src/app# ping 172.19.0.2
PING 172.19.0.2 (172.19.0.2): 56 data bytes
64 bytes from 172.19.0.2: icmp_seq=0 ttl=64 time=0.126 ms
64 bytes from 172.19.0.2: icmp_seq=1 ttl=64 time=0.153 ms
```

That is Docker container isolation in practice. Access to the container is strictly limited by the container and network topology we create. AuthNet is a virtual subnet living solely inside the computer. Making a TCP/IP connection to processes inside the container is limited to whatever we connect to AuthNet.

It's useful to look at the DNS resolver configuration inside the container:

```
root@1f2e3de3be6d:/usr/src/app# cat /etc/resolv.conf
search gateway.sonic.net
nameserver 127.0.0.11
options ndots:0
```

The `sonic.net` line comes from the laptop on which this is being executed, meaning domain name resolution will fall back to the Internet at large. The nameserver at `127.0.0.11` is the one implemented within Docker itself. It's also configured to support domain names with no dots in them, such as `db-auth` or `userauth`. This means we can ping `db-auth` and that domain name will work.

That's how service discovery works in Docker. We give a meaningful name to each service using the `--name` option when building the container. Docker then uses that meaningful name as the container's domain name. The embedded DNS server contains the name and IP address of each container. The DNS resolver configuration uses that internal DNS server to look up names. The resolution algorithm is set up to not require dots in domain names, thus simplifying setup of the service-naming structure.

Since we are conveniently inside the container, we can easily test whether the authentication service can reach the database:

```
root@1f2e3de3be6d:/usr/src/app# PORT=3333 node users-add
Created { id: 1, username: 'me', password: 'w0rd', provider: 'local',
  familyName: 'Einarssdottir', givenName: 'Achildr',
  middleName: '', emails: '[]', photos: '[]',
  updatedAt: '2016-03-23T04:45:29.000Z',
  createdAt: '2016-03-23T04:45:29.000Z' }
```

This behaves exactly as it has in every other instance we've run this script, proving that the Authentication service code in the `userauth` container accessed the MySQL instance in the `db-auth` container. In all the previous examples, the MySQL instance was on localhost, but now it's on what is effectively a separate computer. It's possible with Docker to deploy `db-auth` to a completely separate computer, if you like.

As a side effect, we created a user account we can later use to log in to the Notes application for testing.

Creating FrontNet for the Notes application

We have the back half of our system set up in Docker container, as well as the private bridge network to connect the backend containers. We now need to set up another private bridge network, `frontnet`, and attach the other half of our system to that network.

Let's go ahead and create the `frontnet` bridge network:

```
$ docker network create --driver bridge frontnet
```

MySQL for the Notes application

Dockerizing the front half of the Notes application will more or less follow the pattern we've already discussed.

Create a directory, `db-notes`, as a sibling of the `users` and `notes` directories. In that directory, create a Dockerfile:

```
FROM mysql:5.7

ENV MYSQL_RANDOM_ROOT_PASSWORD=yes
ENV MYSQL_DATABASE=notes
ENV MYSQL_USER=notes
ENV MYSQL_PASSWORD=notes

RUN sed -i "s/^#bind-address.*$/bind-address = 0.0.0.0/" /etc/mysql/my.cnf
RUN sed -i "s/^pid-file/# pid-file/" /etc/mysql/my.cnf
RUN sed -i "s/^socket/# socket/" /etc/mysql/my.cnf

VOLUME /var/lib/mysql

EXPOSE 3306
CMD ["mysqld"]
```

This is almost exactly what we used before, except for the difference in user name, database name, and password.

Now we build the image:

```
$ docker build -t node-web-development/db-notes .
```

And then run it:

```
$ docker run -it --name db-notes --net=frontnet \
node-web-development/db-notes
```

This database will be available at the db-notes domain name on frontnet. Because it's attached to frontnet, it won't be reachable by containers connected to authnet.

Dockerizing the Notes application

In the notes directory, create a file named Dockerfile containing the following:

```
FROM node:5.9.0

ENV DEBUG="notes:* ,messages:*
```

```
ENV SEQUELIZE_CONNECT="models/sequelize-docker-mysql.yaml"
ENV NOTES_MODEL="models/notes-sequelize"
ENV USERS_MODEL="models/users-rest"
ENV USER_SERVICE_URL="http://userauth:3333"
ENV PORT="3000"
ENV NOTES_SESSIONS_DIR="/sessions"

RUN mkdir -p /usr/src/app
COPY . /usr/src/app/
WORKDIR /usr/src/app
RUN apt-get update -y \
    && apt-get -y install curl python build-essential git ca-certificates \
    && npm install --unsafe-perm

VOLUME /sessions
EXPOSE 3000
CMD npm run docker
```

This is similar to the Dockerfile we used for the authentication service. We're using the environment variables from notes/package.json, and the application is installed in the same way.

Because this npm install also runs bower install, we face the same permissions problem discussed earlier. We're using the --unsafe-perm flag so that npm doesn't change its user ID, but what about bower? Earlier, we discussed Bower's --allow-root option and the need to add that option in the postinstall script.

We also have a new SEQUELIZE_CONNECT file. Create models/sequelize-docker-mysql.yaml containing the following:

```
dbname: notes
username: notes
password: notes
params:
```

```
host: db-notes
port: 3306
dialect: mysql
```

This will access a database server on the db-notes domain name using the named database, user name, and password. These parameters of course must match the environment variables defined in db-notes/Dockerfile.

Notice that the `USER_SERVICE_URL` variable no longer accesses the authentication service at `localhost`, but at `userauth`. The `userauth` domain name is currently only advertised by the DNS server on AuthNet, but the Notes service is on FrontNet. This means we'll have to connect the `userauth` container to the `FrontNet` bridge network so that its name is known there as well. We'll get to that in a minute.

A new variable is `NOTES_SESSIONS_DIR` and the matching `VOLUME` declaration. If we were to run multiple `Notes` instances, they could share session data by sharing this volume.

Supporting the `NOTES_SESSIONS_DIR` variable requires one change in `app.js`:

```
const sessionStore = new FileStore({
  path: process.env.NOTES_SESSIONS_DIR ?
    process.env.NOTES_SESSIONS_DIR : "sessions"
});
```

Instead of a hardcoded directory name, we can use an environment variable to define the location where session data is stored. Alternatively, there are `sessionStore` implementations for various servers such as `REDIS`, enabling session data sharing between containers on separate host systems.

Now we can build the container image:

```
$ docker build -t node-web-development/notesapp .
```

And then run it as so:

```
$ docker run -it --name notes --net=frontnet -p 3000:3000 \
node-web-development/notesapp
```

This container is the one we want to publish on the Internet. The `-p` option is what lets us publish a port so the world can beat a path to our doorstep. We could also map the port number to the usual HTTP port (80) by using `-p 80:3000`.

You can inspect FrontNet's details now:

```
$ docker network inspect frontnet
```

Putting FrontNet together

We're almost ready to test this, but for connecting the `userauth` container to FrontNet. So far we've created two bridge networks, with no connection between them. For Notes to use the authentication service, it must be able to access the `userauth` container.

The `userauth` container is already on AuthNet. We can add it to FrontNet in the following way:

```
$ docker network connect frontnet userauth
```

The `userauth` container is now attached to both bridge networks. This means the notes service will be able to access `userauth` for its authentication needs.

Now we can finally turn to our web browser and use the Notes application. The issue is which access URL to use.

Previously we'd accessed Notes using `http://localhost:3000` because it was running via the laptop OS. But it's now running inside a virtualized Linux container, and that container has its own IP address. Depending on the configuration of this virtualized Linux, the IP address might not even be visible outside our laptop.

If you are using Docker Toolbox, first find out the IP address of the Notes application service using the following:

```
$ docker-machine ip
```

Then use that IP address in the URL, such as: `http://192.168.55.101:3000`. If instead you're using Docker for Windows or Mac, you should run `docker ps`. The output of that command is another way to learn the IP address of the container, and which container has an exposed port:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
30e541489c70	compose_notesapp	"/bin/sh -c 'npm run "	8 minutes ago	Up 21 seconds	192.168.64.5:3000->3000/tcp	notesapp
8c2b2b70e338	compose_userauth	"/bin/sh -c 'npm run "	8 minutes ago	Up 22 seconds	3333/tcp	userauth
66553ae50f17	compose_db-notes	"/entrypoint.sh mysql"	8 minutes ago	Up 23 seconds	3306/tcp	db-notes
c40fef3c0c74	compose_db-auth	"/entrypoint.sh mysql"	8 minutes ago	Up 23 seconds	3306/tcp	db-auth

In the **PORTS** column, it shows which ports are published from which container. When the listing shows `0.0.0.0` for the IP address, that port is visible to the world. In such a case, you can access Notes via `http://localhost:3000` as previously. It may instead show a fixed IP address, in which case you use that to access Notes. Twitter login will work if you reconfigure the Twitter application registration to use the IP address we just determined. But we did create a local user profile earlier while testing AuthNet, so we could use that to log in and test Notes.

But let's explore the current system. Compare the output of the following commands:

```
$ docker network inspect frontnet  
$ docker network inspect authnet
```

You'll see that userauth is listed in both networks, as expected. This means that from userauth, you can reach hosts on both networks and, for example, pinging both db-auth and db-notes will work:

```
$ docker exec -it userauth bash  
# ping db-auth  
PING db-auth (172.19.0.2): 56 data bytes  
64 bytes from 172.19.0.2: icmp_seq=0 ttl=64 time=0.087 ms  
# ping db-notes  
PING db-notes (172.20.0.3): 56 data bytes  
64 bytes from 172.20.0.3: icmp_seq=0 ttl=64 time=0.104 ms
```

This is the only container in our system where this is true. Try accessing the other containers and try the same test. Whether this is acceptable depends on you. If a miscreant were able to break into the userauth container, they could access the rest of the system. Maybe we need to erect a barrier against that possibility.

There is a fairly simple solution, which is to create a third bridge network solely for the purpose of connecting notesapp to userauth:

```
$ docker network create --driver bridge notesauth  
$ docker network connect notesauth notesapp  
$ docker network disconnect frontnet userauth  
$ docker network connect notesauth userauth  
$ docker network inspect authnet  
$ docker network inspect notesauth  
$ docker network inspect frontnet
```

The notesauth network serves as the connector between notes and userauth. Those containers can contact each other, and now userauth cannot connect to db-notes, a service which it had no need to connect with:

```
$ docker exec -it userauth bash  
# ping db-notes  
ping: unknown host
```

If we use our browser to peruse the Notes application, it still works perfectly.

Accessing the Dockerized Notes app from another device In your testing, you might have whipped out a mobile computer, or another computer, and tried to use the application. This is likely to have failed, however, because of the virtual machine configuration on your laptop.

The virtual machine Docker has its own OS and network stack. By default, Docker configures this virtual machine so that it cannot be accessed from outside your laptop.

It's recommended to routinely test the application on mobile devices.

Over the next couple of sections, we'll see what configuration changes to make, depending on what you've installed on your laptop.

Configuring remote access on Docker for Windows or Mac

Unfortunately, at the time of writing, Docker for Windows and Docker for Mac products are rapidly changing and therefore we cannot give precise steps to follow. You should consult the previously mentioned documentation instead.

There are two behaviors to consider:

- Whether ports published by the Docker host are visible only to a private IP address on your laptop or if those ports are forwarded to your laptop's Internet connection
- Whether software inside a Docker container connects out to the Internet as if it's behind a NAT firewall or if the Docker process proxies that traffic

At the time of writing, the out-of-the-box behavior appears to be to publish ports from Docker via your laptop's IP address. This means you can use `http://localhost` from browsers running on your laptop, and the IP address of your laptop from other devices.

The Docker application can be configured so that ports are not exported from your laptop. You'll only have access from the laptop's browser, and not from any other device.

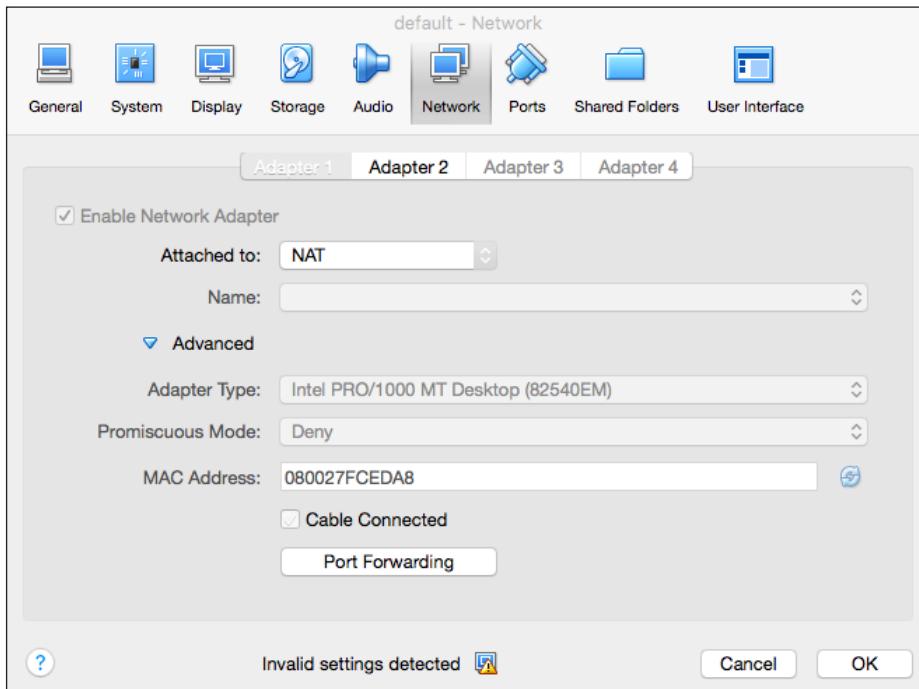
Configuring remote access in VirtualBox on Docker toolbox

The same considerations exist when using VirtualBox. It has different defaults, and fortunately the actions to take are well documented.

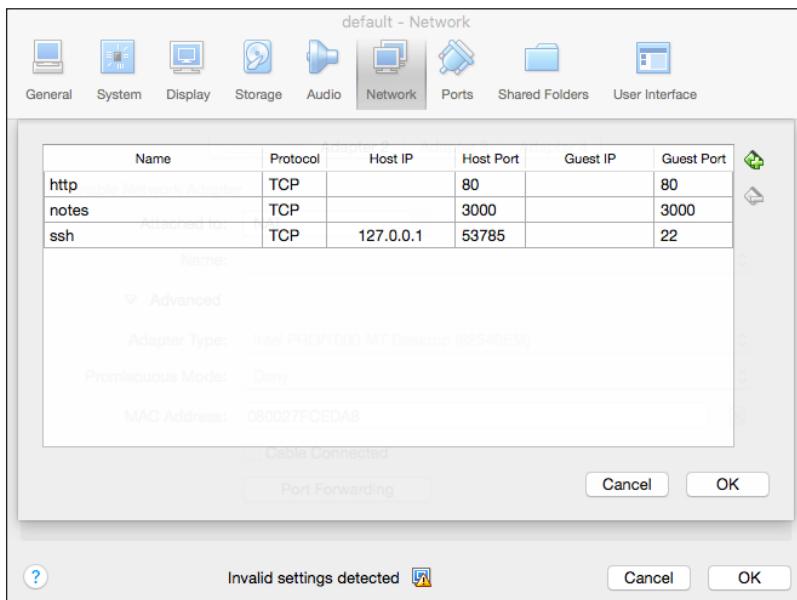
By default, with VirtualBox, TCP ports from guest machines are not exposed outside your laptop, the guest machine has a private IP address that is not available from outside the laptop, and software inside the guest machine can access the Internet as if it's behind a NAT firewall.

The issues are explained in the VirtualBox user guide. We're told that, by default, guest machines running inside VirtualBox are configured as if they are behind a NAT router. Machines behind a NAT router are able to access services outside their network, but it is difficult to go the other direction, unless you configure *port forwarding* in the NAT router.

You might do this with your home Wi-Fi router so that, for example, you can host a website on your home network connection. In this case, the NAT router is embedded in VirtualBox, and we need to configure port forwarding in its NAT router. To do so, you start the VirtualBox GUI, select the Docker host instance, and then click on the **Settings** button:



At the bottom of the screen is a **Port Forwarding** button. Click on that:



Then add an entry so that host port 3000 maps to guest port 3000. Presto, you'll now be able to access Notes, hosted inside the Docker instance on your laptop, from other computers.

Any ports you wish to expose outside your laptop need to be configured here.

Exploring the Docker Toolbox VirtualBoxMachine

As long as we're poking at VirtualBox, let's take a peek under the covers of the virtual machine we just hosted these containers on. Fortunately, Docker makes this easy:

We can easily log in to the Docker Machine host and see that, yes, indeed, boot2docker still lives behind the scenes:

```
1239 ? Sl 0:03 /usr/local/bin/docker daemon -D -g /var/lib/docker -H unix://
-H tcp://0.0.0.0:2376 --label provider=virtualbox --tlsverify --tlscacert=/var/lib/boot2d
ocker/ca.pem --tlscert=/var/lib/boot2docker/server.pem --tlskey=/
1601 pts/0 Ssl+ 0:01 \_ mysqld
1654 pts/1 Ss+ 0:00 \_ /bin/sh -c npm run docker
1664 pts/1 Sl+ 0:00 | \_ npm
1677 pts/1 S+ 0:00 | \_ sh -c node user-server
1678 pts/1 Sl+ 0:01 | \_ node user-server
1692 pts/2 Ssl+ 0:01 \_ mysqld
1746 ? Sl 0:00 \_ docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 3000 -
container-ip 172.20.0.4 -container-port 3000
1752 pts/3 Ss+ 0:00 \_ /bin/sh -c cd /usr/src/app && npm run docker
1761 pts/3 Sl+ 0:00 \_ npm
1775 pts/3 S+ 0:00 \_ sh -c node ./app
1776 pts/3 Sl+ 0:02 \_ node ./app
```

Running a `ps` command tells us that there's a full complement of Linux background processes. Most important is this group of processes, which are exactly what we started in these containers. We see the two MySQL instances and two Node.js processes started by the containers.

This innocently looks like any process list, doesn't it? But we've demonstrated that access to processes inside a container is strictly limited. How can that be if these are host OS processes?

That's the containerization technology at work, the details of which are not visible in this process listing.

If instead you've installed Docker for Windows or Mac, you also have a virtual Linux system running inside virtualization software that is not VirtualBox. There will be a command to `ssh` into that virtual Linux box. Once inside that virtual machine, you'll see a similar process list.

Controlling the location of MySQL data volumes

The `db-auth` and `db-notes` Dockerfiles contain `VOLUME /var/lib/mysql`.

Doing this is the first step to ensuring that the database files persist even if we delete the container. As said previously, Docker containers are quick and cheap to create and delete, while databases are meant to last for years or decades.

The `VOLUME` instruction instructs Docker to create a directory outside the container and to map that directory so that it's mounted inside the container on the named path. The `VOLUME` instruction by itself doesn't control the directory name on the host computer. But at least the data is available outside the container.

The first step is to discover where Docker put the volume:

```
$ docker inspect --format '{{json .Mounts}}' db-notes
[{
  "Name": "e05b81e4dee1c71f1e1a48e119c48a35bfdc9e0bec742b20577b71fbc71d294b",
  "Source": "/mnt/sda1/var/lib/docker/volumes/e05b81e4dee1c71f1e1a48e119c48a35bfdc9e0bec742b20577b71fbc71d294b/_data",
  "Destination": "/var/lib/mysql",
  "Driver": "local",
  "Mode": "",
  "RW": true,
  "Propagation": ""
}]
```

That's not exactly a user-friendly pathname, but you can snoop into that directory and see that indeed the MySQL database is stored there. You'll have to do it this way if you've installed Docker Toolbox:

```
$ docker-machine ssh
$ sudo ls /mnt/path/to/volume/location
```

If you've installed Docker for Windows or Mac, consult the documentation on how to access the Docker host.

We make these user-friendly pathnames with the following command:

```
$ docker volume create --name db-auth-data
$ docker volume create --name db-notes-data
```

As the command string implies, this creates a volume. We can then recreate the container so it references the volume like so:

```
$ docker run -it --name db-auth --net=authnet \
  --volume=db-auth-data:/var/lib/mysql:rw \
  node-web-development/db-auth
$ docker run -it --name db-notes --net=frontnet \
  --volume=db-notes-data:/var/lib/mysql:rw \
  node-web-development/db-notes
```

The `--volume` argument is what connects the exported volume to a specific location in the filesystem.

And now the volume location has a nicer pathname:

```
$ docker volume inspect db-auth-data
[{
  "Name": "db-auth-data", "Driver": "local",
  "Mountpoint": "/mnt/sda1/var/lib/docker/volumes/db-auth-data/_data"
}]
```

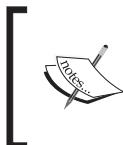
This is improved, but all we've done is make the pathname easier to read. We haven't controlled the directory location.

Docker allows us to specify the volume mapping in another way:

```
docker .. --volume=/Host/Path:/Container/Path ..
```

This says to map a specific directory on the host into the container.

Unfortunately, we run into a problem with the MySQL container and mapping its volume from inside a VirtualBox Docker Engine. Docker does configure VirtualBox so that any path under /Users is automatically available for containers to share data on the host computer filesystem. But this only works for certain user IDs. Many containers, like MySQL, execute their processes under a specific user ID, which then does not map correctly to the host computer filesystem.



It's a known issue which you can read about; refer to <https://github.com/boot2docker/boot2docker/issues/581> and <https://github.com/boot2docker/boot2docker/issues/1101>.

This exact issue is one of the primary reasons why Docker for Windows or Mac was created. It fixes this problem and allows you to use host paths in the volume mapping, with the Docker software correctly mapping file permissions.

When running with Docker for Windows or Mac, you do not have to run the `docker volume create` command shown earlier, and can instead run the following command:

```
$ docker run -it --name db-auth --net=authnet \
  --volume=./db-auth/data:/var/lib/mysql:rw \
  node-web-development/db-auth

$ docker run -it --name db-notes --net=frontnet \
  --volume=./db-notes/data:/var/lib/mysql:rw \
  node-web-development/db-notes
```

This uses db-auth/data and db-notes/data for the data directories.

Because the volumes exist on the host filesystem, we can read and write these files using normal Linux tools. That makes it easy to perform backups, copy volume data to other systems, and so on. But do this with care since the processes running inside the container may not be compatible with their files being manipulated.

For example, the MySQL server should be shut down before attempting to copy the raw database files. And importing raw MySQL database files into another server has to be done with care.

Deploying to the cloud with Docker compose

This is cool, but we need to take all this learning and apply it to the task we set ourselves. Namely, to deploy the Notes application on a public Internet server with a fairly high degree of security. We've demonstrated that, with Docker, the Notes application can be decomposed into four containers that have a high degree of isolation from each other and from the outside world. It's claimed that this greatly improves security, while giving huge deployability benefits. Deployment containers are described with Dockerfiles that are used to build deployment images, which can be easily deployed to target systems.

Docker Compose (<https://docs.docker.com/compose/overview/>) 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 the deployment onto a single host machine, Compose can be used for multimachine deployments, especially when combined with Docker Swarm.

The docker-compose command is used not only to build a set of containers and intercontainer configuration, but also to deploy the whole lot to a local or remote Docker server.

Docker compose files

Let's start by creating a directory, compose, as a sibling to the users and notes directories. In that directory, create a file named docker-compose.yml:

```
version: '2'  
services:  
  
  db-auth:  
    build: ../db-auth  
    container_name: db-auth
```

```
networks:
  - authnet
volumes:
  - db-auth-data:/var/lib/mysql

userauth:
  build: ../users
  container_name: userauth
  networks:
    - authnet
    - notesauth
  expose:
    - 3333
  depends_on:
    - db-auth
  restart: always

db-notes:
  build: ../db-notes
  container_name: db-notes
  networks:
    - frontnet
  volumes:
    - db-notes-data:/var/lib/mysql

notesapp:
  build: ../notes
  container_name: notesapp
  networks:
    - frontnet
    - notesauth
  expose:
    - 3000
  ports:
    - "3000:3000"
  depends_on:
    - db-notes
    - userauth
  restart: always

networks:
  authnet:
    driver: bridge
frontnet:
```

```
driver: bridge
notesauth:
  driver: bridge

volumes:
  db-auth-data:
  db-notes-data:
```

That's the description of the entire `Notes` application deployment. It's at a fairly high level of abstraction, roughly equivalent to the options on the command-line tools we've used so far. Further details are located inside the Dockerfiles, which are referenced from this `Compose` file.

The `version` line says that this is a version 2 `Compose` file. Previous versions of `Compose` did not have this line and had a number of differences to the contents of the `Compose` file.

There are three major sections used here: *services*, *volumes*, and *networks*. The *services* section describes the containers being used, the *networks* section, of course, describes the networks, and the *volumes* section, of course, describes the volumes. The contents of each section match exactly the commands we ran earlier.

For example, each of the three networks in our system is a bridge network. This fact is described in the `Compose` file.

The `build` attribute on containers specifies a directory containing a Dockerfile. That directory contains everything related to building the specific container. We have four such containers: `db-auth`, `db-notes`, `userauth`, and `notesapp`.

It's possible to directly use a Docker image without having to write a Dockerfile. Instead of using the `build` attribute, you use the `image` attribute listing the image name. We could have used this approach for the two MySQL containers, except there were customizations required which could not be described in the `Compose` file.

The `container_name` attribute is equivalent to the `--name` attribute and specifies a user-friendly name for the container.

The `networks` attribute lists the networks to which this container must be connected and is exactly equivalent to the `--net` argument. The networks in question are, of course, listed in the *networks* section later in the file.

The `expose` attribute declares which ports are exposed from the container. The exposed ports are not published outside the host machine, however. The `ports` attribute declares the ports which are to be published. 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 startup order. A container that depends on another will wait to start until the depended-upon 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-auth-data` and `db-notes-data`, and then used them for the volume mapping. You can, of course, inspect the volume location using the `docker` command line:

```
$ docker volume inspect compose_db-notes-data  
$ docker volume inspect compose_db-auth-data
```

The preceding volume configuration is correct when running under Docker Toolbox. Because of limitations while using VirtualBox, `docker-compose.yml` cannot specify a host-side pathname for the volume. But, if you're instead using Docker for Windows or Mac, you can take a different approach in `docker-compose.yml`.

First, comment out the `volumes` section because you won't need it. Then change the `volumes` attribute of `db-auth` and `db-notes` as follows:

```
db-auth:  
  ...  
  volumes:  
    # - db-auth-data:/var/lib/mysql  
    - ./db-auth/data:/var/lib/mysql  
  
db-notes:  
  ...  
  volumes:  
    # - db-notes-data:/var/lib/mysql  
    - ./db-notes/data:/var/lib/mysql
```

This is the same configuration change we made earlier. It uses `db-auth/data` and `db-notes/data` as the data directories for their respective database containers.

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 it should restart the process? We could use a background process supervisor, like Supervisord or PM2. But, we can also use the Docker `restart` option.

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

Running the Notes application with Docker Compose

Before deploying this to a server, let's run it on our laptop using `docker-compose`:

```
$ docker stop db-notes userauth db-auth notesapp
db-notes
userauth
db-auth
notesapp
$ docker rm db-notes userauth db-auth notesapp
db-notes
userauth
db-auth
notesapp
```

We first needed to stop and delete the existing containers. Because the Compose file wants to launch containers with the same names as we'd built earlier, we also have to remove the existing containers:

```
$ docker-compose build
Building db-auth
.. lots of output
$ docker-compose up
Creating db-auth
Recreating compose_db-notes_1
Recreating compose_userauth_1
```

```
Recreating compose_notesapp_1
Attaching to db-auth, db-notes, userauth, notesapp
```

Once that's done, we can build the containers, docker-compose build, and then start them running, docker-compose up.

The first test is to execute a shell in userauth to run our user database script:

```
$ docker exec -it userauth bash
# PORT=3333 node users-add.js
Created { id: 1, username: 'me', password: 'w0rd', provider: 'local',
  familyName: 'Einarssdottir', givenName: 'Ashildr',
  middleName: '', emails: '[]', photos: '[]',
  updatedAt: '2016-03-24T01:22:38.000Z',
  createdAt: '2016-03-24T01:22:38.000Z' }
```

Now that we've proved that the authentication service will work, and by the way created a user account, you should be able to browse to the Notes application and run it through its paces.

You can also try pinging different containers to ensure that the application network topology has been created correctly.

 If you use Docker command-line tools to explore the running containers and networks, you'll see they have new names. The new names are similar to the old names, but prefixed with the string compose_. This is a side effect of using Docker Compose.

Deploying to cloud hosting with Docker Compose

Now that we've verified on our laptop that the services described by the Compose file work as intended, we can deploy the whole thing to cloud hosting.

So far we've used Docker Machine to interact with a Docker instance on the localhost. The docker-machine command comes with *drivers* supporting a long list of cloud-hosting providers. With these commands, we can create host systems preconfigured as Docker Machine from the convenience of the command line on our laptop.

In this section, we'll use the Digital Ocean service to deploy the Notes application. If you prefer a different cloud host, by all means use it. You'll simply have to translate certain details over to your chosen hosting provider.

After signing up for a Digital Ocean account, click on the API link in the dashboard. We need an API token to grant docker-machine access to the account. Go through the process of creating a token and save away the token string you're given. The Docker website has a tutorial at <https://docs.docker.com/machine/examples/ocean/>.

With the token in hand, type the following:

```
$ docker-machine create --driver digitalocean --digitalocean-size 2gb  
--digitalocean-access-token ..TOKEN.. sandbox  
  
Running pre-create checks...  
  
Creating machine...  
  
(sandbox) Creating SSH key...  
  
(sandbox) Creating Digital Ocean droplet...  
  
(sandbox) Waiting for IP address to be assigned to the Droplet...  
  
Waiting for machine to be running, this may take a few minutes...  
  
Detecting operating system of created instance...  
  
Waiting for SSH to be available...  
  
Detecting the provisioner...  
  
Provisioning with ubuntu(systemd)...  
  
Installing Docker...  
  
Copying certs to the local machine directory...  
  
Copying certs to the remote machine...  
  
Setting Docker configuration on the remote daemon...  
  
Checking connection to Docker...  
  
Docker is up and running!
```

To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: `docker-machine env sandbox`.

This reaches out over the Internet to Digital Ocean's servers, causes Digital Ocean to create what they call a **Droplet**, and initializes that Droplet as a Docker Engine host. Where we would use `-d virtualbox` to create another Docker Machine on our laptop, we use `-d digitalocean` to do so on Digital Ocean.

We gave this machine the name `sandbox`.

Once the command finishes, you can see the new Droplet in your Digital Ocean dashboard.

As we do with the Docker Machine on our laptop, set up the environment variables:

```
$ eval $(docker-machine env sandbox)
```

With this method, we can create multiple Docker Machine instances. We switch between the machines by rerunning this command, thus specifying the desired machine instance name.

The next step is to build our containers for the new machine. Because we've switched the environment variables to point to the new server, these commands cause action to happen there rather than inside our laptop:

```
$ docker-compose build
```

This time, because we changed the environment variables, the build occurs on the `sandbox` machine rather than on our laptop as previously.

This will take a while. The Docker image cache on the remote machine is empty, meaning that every container build starts from scratch. Additionally, building the `notesapp` and `userauth` containers copies the entire source tree to the server and reruns `npm install` there.

The build may fail if the default memory size is 500 MB, the default on Digital Ocean at the time this was written. If so, the first thing to try is resizing the memory on the host to at least 2 GB.

Once the build is finished, launch the containers on the remote machine:

```
$ docker-compose up
```

Once the containers start, you should test the `userauth` container as we've done previously. Execute a shell in `userauth` to test and set up the user database:

```
$ docker exec -it userauth bash
# PORT=3333 node users-add.js
Created { id: 1, username: 'me', password: 'w0rd', provider: 'local',
familyName: 'Einarssdottir', givenName: 'Achildr',
middleName: '', emails: '[]', photos: '[]',
updatedAt: '2016-03-24T01:22:38.000Z',
createdAt: '2016-03-24T01:22:38.000Z' }
```

As mentioned previously, this verifies that the `userauth` service works, that the remote containers are set up, and that we can proceed to using the Notes application.

The question is: What's the URL to use? We don't have a domain name assigned, but there is an IP address for the server.

Run the following command:

```
$ docker-machine ip sandbox  
159.203.105.135
```

Docker tells you the IP address, which you should use as the basis of the URL.

This is much simpler than when we deployed to Docker containers on our laptop. In this case, there's no virtual machine between the containers and the host system. The ports exposed from Docker containers on the Docker host are simply visible to the Internet.

With Notes deployed to the remote server, you should check out all the things we've looked at previously. The three bridge networks should exist, as shown previously, with the same limited access between containers. The only public access should be port 3000 on the notesapp container. If you reconfigure the Twitter application to recognize the server's IP address, you should be able to log in to Notes using a Twitter account. By now, you know the drill.

Because our database containers mount a volume to store the data, let's see where that volume landed on the server:

```
$ docker volume ls  
  
DRIVER          VOLUME NAME  
local           compose_db-auth-data  
local           compose_db-notes-data
```

Those are the expected volumes, one for each container:

```
$ docker volume inspect compose_db-auth-data compose_db-notes-data  
[  
 {  
   "Name": "compose_db-auth-data",  
   "Driver": "local",  
   "Mountpoint":  
     "/var/lib/docker/volumes/compose_db-auth-data/_data",  
   "Labels": null  
,  
 {  
   "Name": "compose_db-notes-data",  
   "Driver": "local",  
   "Mountpoint":
```

```
        "/var/lib/docker/volumes/compose_db-notes-data/_data",
    "Labels": null
}
]
```

Those are the directories, but they're not located on our laptop. Instead they're on the remote server. Accessing these directories means logging into the remote server to take a look:

```
$ docker-machine ssh sandbox
Welcome to Ubuntu 15.10 (GNU/Linux 4.2.0-27-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
New release '16.04 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Sat Apr 30 19:09:01 2016 from ####.####.####.####
root@sandbox:~# docker volume ls
DRIVER          VOLUME NAME
local           compose_db-auth-data
local           compose_db-notes-data
```

From this point, you can inspect the directories corresponding to these volumes and see that indeed they contain MySQL configuration and data files.

You'll also find that the Docker command-line tools will work with no configuration such as setting environment variables.

Once you're satisfied that Notes is working on the remote server, you can shut it down and remove it as so:

```
$ docker-compose stop
Stopping notesapp ... done
Stopping userauth ... done
stopping db-notes ... done
Stopping db-auth ... done
```

This shuts down all the containers at once.

```
$ docker-machine stop sandbox
Stopping "sandbox"...
Machine "sandbox" was stopped.
```

This shuts down the remote machine. The Digital Ocean dashboard will show that the Droplet has stopped.

At this point, you can use the Digital Ocean dashboard to save a snapshot of the Droplet. This way you can go ahead and delete the Droplet, but still reinstate it any time you like.

```
$ docker-machine rm sandbox
About to remove sandbox
Are you sure? (y/n): y
Successfully removed sandbox
```

And, if you're truly certain you want to delete the machine, the preceding command does the deed. As soon as you do this, the Droplet will be erased from your Digital Ocean dashboard.

Summary

This chapter has been quite a journey in learning two ways to deploy Node.js applications to a production server.

You started by reviewing the Notes application architecture and how that will affect deployment. That let you understand what you had to do for server deployment.

Then you learned the traditional way to deploy services on Linux using an init script. The PM2 command is a useful tool for managing background processes. You also learned how to provision a remote server using a virtual machine hosting service.

Then you took a long trip into the land of Docker, a new and exciting system for deploying services on machines. You learned how to write a Dockerfile so that Docker knows how to construct a service image. You learned several ways to deploy Docker images on our laptop or on a remote server. And you learned how to describe a multicontainer application using Docker Compose.

You're almost ready to wrap up this book. You've learned a lot along the way, but you have one final thing to cover.

While a core principle of test-driven development is to write the unit tests before writing the application, we've done it the other way around and put the chapter about unit testing at the end of this book, which is the next and final chapter in this book. That's not to say unit testing is unimportant because it is extremely important. But, to jump-start your knowledge of application development with Node.js and Express, we had to first cover that technology stack.

11

Unit 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 proper functioning. Each unit is theoretically the smallest testable part of an application. In a Node.js application, you might consider each module as a unit.

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. Common technologies to use are mock objects and other methods to present a faked dependency implementation. We will focus on that testing model in this chapter.

Functional testing, on the other hand, doesn't try to test individual components, but instead it tests the whole system. Generally speaking, unit testing is performed by the development team, and functional testing is performed by a 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. We'll also do a little bit of functional testing.

Now that we've written several chunks of software in this book, let's use that software to explore unit test implementation.

Testing asynchronous code

The asynchronous nature of the Node.js platform means accounting for asynchronous behavior in tests. Fortunately, the Node.js unit test frameworks help in dealing with this, but it's worth spending a few moments considering the underlying problem.

Consider a code snippet like this, which you could save in a file named `deleteFile.js`:

```
const fs = require('fs');
exports.deleteFile = function(fname, callback) {
  fs.stat(fname, (err, stats) => {
    if (err)
      callback(new Error(`the file ${fname} does not exist`));
    else {
      fs.unlink(fname, err2 => {
        if (err)
          callback(new Error(`could not delete ${fname}`));
        else callback();
      });
    }
  });
};
```

The nature of asynchronous code is such that its execution order is nonlinear, meaning that there is a complex relationship between time and the lines of code. Since this is the real world, and not science fiction, we don't have a time machine (blue box or not) to help us traverse the web of time, and therefore, we must rely on software tools. The code is not executed one line after another, but each callback is called according to the flow of events. This snippet, like many written with Node.js, has a couple of layers of callback, making the result arrive in an indeterminate amount of time in the future.

Testing the `deleteFile` function can be accomplished with this code which you can save in `test-deleteFile.js`:

```
const df = require('./deleteFile');
df.deleteFile("no-such-file", err => {
  // the test passes if err is an Error with the message
  //     "the file no-such-file does not exist"
  // otherwise the test fails
});
```

This is what we call a negative test. It verifies that the failure path actually executes. It's relatively easy to verify which path was taken within the `deleteFile` function, by inspecting the `err` variable. But, what if the callback is never called? That would also be a failure of the `deleteFile` function, but how do you detect that condition? You might be asking, how could it be that the callback is never called? The behavior of file access over NFS includes conditions where the NFS server is wedged and filesystem requests never finish, in which case the callbacks shown here would never be called.

Assert – the simplest testing methodology

Node.js has a useful testing tool built-in with the `assert` module. It's not any kind of testing framework but simply a tool that can be used to write test code by making assertions of things that must be true or false at a given location in the code.

The `assert` module has several methods providing various ways to assert conditions that must be true (or false) if the code is properly functioning.

The following is an example of using `assert` for testing, providing an implementation of `test-deleteFile.js`:

```
const fs = require('fs');
const assert = require('assert');
const df = require('../deleteFile');
df.deleteFile("no-such-file", (err) => {
    assert.throws(
        function() { if (err) throw err; },
        function(error) {
            if ((error instanceof Error)
                && /does not exist/.test(error)) {
                return true;
            } else return false;
        },
        "unexpected error"
    );
});
```

If you are looking for a quick way to test, the `assert` module can be useful when used this way. If it runs and no messages are printed, then the test passes.

```
$ node test-deleteFile.js
```

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.

There are many styles of assertion libraries available in contributed modules. 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 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? Testing a data model and not mocking out the database means that to run the test, one must first launch the database, making that a dependency of running the test. On the other hand, avoiding the overhead of launching a database engine means that you will have to create a fake Sequelize implementation. That does not look like a productive use of our time. One can argue that testing a data model is really about testing the interaction between your code and the database, that mocking out the database means not testing that interaction, and therefore we should test our code against the database engine used in production.

With that line of reasoning in mind, we'll skip mocking out the database, and test the models against a real database server. Instead of running the test against the live production database, it should be run 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 that's set up for testing.

Mocha and Chai the chosen test tools

If you haven't already done so, duplicate the source tree to use in this chapter. For example, if you had a directory named `chap10`, create one named `chap11` containing everything from `chap10`.

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 Promise's.

While in the `notes` directory, type this to install Mocha and Chai:

```
$ npm install mocha@2.x chai@3.x --save-dev
```

We saw similar commands plenty of times, but this time, we used `--save-dev` rather than the `--save` we saw earlier. This new option saves the package name to the `devDependencies` list in `package.json` rather than the `dependencies` list. The idea is to record which dependencies to use in production and which are used in development or testing environments.

We only want Mocha and Chai installed on a development machine, not on a production machine. With npm, installing, the production version of our code is done this way:

```
$ npm install --production
```

Alternatively, it can be done as follows:

```
$ NODE_ENV=production npm install
```

Either approach causes npm to skip installing packages listed in `devDependencies`.

In `compose/compose-docker.yml`, we should then add this to both the `userauth` and `notesapp` sections:

```
environment:  
  - NODE_ENV="production"
```

When the production deployment Dockerfile executes `npm install`, this ensures that development dependencies are not installed.

Notes model test suite

Because we have several Notes models, the test suite should run against any model. We can write tests using the Notes model API we developed, and an environment variable should be used to declare the model to test.

In the `test` directory, create a file named `test-model.js` containing this as the outer shell of the test suite:

```
'use strict';  
  
const assert = require('chai').assert;  
  
const model = require(process.env.MODEL_TO_TEST);  
  
describe("Model Test", function() {  
  ..  
});
```

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 styles supported by Chai, see <http://chaijs.com/guide/styles/>.

We'll specify the Notes model to test with the `MODEL_TO_TEST` environment variable. For the models that also consult environment variables, we'll need to supply that configuration as well.

With Mocha, a test suite is contained within a `describe` block. The first argument is descriptive text, which you use to tailor presentation of test results.

Rather than maintaining a separate test database, we can create one on the fly while executing tests. Mocha has what are called "hooks" which are functions executed before or after test case execution. The hook functions let you the test suite author, set up and tear down required conditions for the test suite to operate as desired. For example, to create a test database with known test content.

```
describe("Model Test", function() {
  beforeEach(function() {
    return model.keylist().then(keyz => {
      var todel = keyz.map(key => model.destroy(key));
      return Promise.all(todel);
    })
    .then(() => {
      return Promise.all([
        model.create("n1", "Note 1", "Note 1"),
        model.create("n2", "Note 2", "Note 2"),
        model.create("n3", "Note 3", "Note 3")
      ]);
    });
  });
  ...
});
```

This defines a `beforeEach` hook, which is executed before every test case. The other hooks are `before`, `after`, `beforeEach`, and `afterEach`. The `Each` hooks are triggered before or after each test case execution.

This uses our Notes API to first delete all notes from the database (if any) and then create a set of new notes with known characteristics. This technique simplifies tests by ensuring that we have known conditions to test against.

We also have a side effect of testing the `model.keylist` and `model.create` methods.

In Mocha, test cases are written using an `it` block contained within a `describe` block. You can nest the `describe` blocks as deeply as you like. Add the following `beforeEach` block as shown:

```
describe("check keylist", function() {
    it("should have three entries", function() {
        return model.keylist().then(keyz => {
            assert.equal(3, keyz.length, "length 3");
        });
    });
    it("should have keys n1 n2 n3", function() {
        return model.keylist().then(keyz => {
            keyz.forEach(key => {
                assert.match(key, /n[123]/, "correct key");
            });
        });
    });
    it("should have titles Node #", function() {
        return model.keylist().then(keyz => {
            var keyPromises = keyz.map(key => model.read(key));
            return Promise.all(keyPromises);
        })
        .then(notez => {
            notez.forEach(note => {
                assert.match(note.title, /Note [123]/);
            });
        });
    });
});
});
```

The idea is of course to call Notes API functions, then to test the results to check whether they matched the expected results.

This `describe` block is within the outer `describe` block. The descriptions given in the `describe` and `it` blocks are used to make the test report more readable.



It is important with Mocha to not use arrow functions in the `describe` and `it` blocks. By now, you will have grown fond of these because of how much easier they are to write. But, 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.

You'll see that we used a few arrow functions here. It's the function supplied to the `describe` or `it` block, which must be a traditional function declaration. The others can be arrow functions.

How does Mocha know whether the test code passes? How does it know when the test finishes? This segment of code shows one of the three methods.

Namely, the code within the `it` block can return a Promise object. The test finishes when the promise finishes, and it passes or fails depending on whether the Promise concludes successfully or not.

Another method for writing tests in Mocha is with non-asynchronous code. If that code executes without throwing an error then the test is deemed to have passed. Assertion libraries such as Chai are used to assist writing checks for expected conditions. Any assertion library can be used, so long as it throws an error.

In the tests shown earlier, we did use Chai assertions to check values. However, those assertions were performed within a Promise. Remember that Promises catch errors thrown within the `.then` functions, which will cause the Promise to fail, which Mocha will interpret as a test failure.

The last method for writing tests in Mocha is used for asynchronous code. You write the `it` block callback function so that it takes an argument. Mocha supplies a callback function to that argument, and when your test is finished, you invoke that callback to inform Mocha whether the test succeeded or failed.

```
it("sample asynchronous test", function(done) {  
  performAsynchronousOperation(arg1, arg2, function(err, result) {  
    if (err) return done(err); // test failure  
    // check attributes of result against expected result  
    if (resultShowsFail) return done(new Error("describe fail"));  
    done(); // Test passes  
  });  
});
```

Using Mocha to test asynchronous functions is simple. Call the function and asynchronously receive results, verifying that the result is what's expected, then call `done(err)` to indicate a fail or `done()` to indicate a pass.

Configuring and running tests

We have more tests to write, but let's first get set up to run the tests.

The simplest model to test is the in-memory model. Let's add this to the `scripts` section of `package.json`:

```
"test-notes-memory": "MODEL_TO_TEST=../models/notes-memory mocha",
```

Then, we can run it as follows:

```
$ npm run test-notes-memory

> notes@0.0.0 test-notes-memory /Users/david/chap11/notes
> MODEL_TO_TEST=../models/notes-memory mocha

Model Test
  check keylist
    ✓ should have three entries
    ✓ should have keys n1 n2 n3
    ✓ should have titles Node #

  3 passing (23ms)
```

The `mocha` command is used to run the test suite. With no arguments, it looks in the `test` directory and executes everything there. Command-line arguments can be used to tailor this, so you can run a subset of tests or change the reporting format.

More tests for the Notes model

That wasn't enough to test much, so let's go ahead and add the some more tests:

```
describe("read note", function() {
  it("should have proper note", function() {
    return model.read("n1").then(note => {
      assert.equal(note.key, "n1");
      assert.equal(note.title, "Note 1");
      assert.equal(note.body, "Note 1");
    });
  });

  it("Unknown note should fail", function() {
    return model.read("badkey12")
      .then(note => { throw new Error("should not get here"); })
      .catch(err => {
        // this is expected, so do not indicate error
      });
  });
});

describe("change note", function() {
  it("after a successful model.update", function() {
```

```
        return model.update("n1",
    "Note 1 title changed",
    "Note 1 body changed")
    .then(newnote => { return model.read("n1"); })
    .then(newnote => {
        assert.equal(newnote.key, "n1");
        assert.equal(newnote.title, "Note 1 title changed");
        assert.equal(newnote.body, "Note 1 body changed");
    });
})
});
describe("destroy note", function() {
    it("should remove note", function() {
        return model.destroy("n1").then(() => {
            return model.keylist()
            .then(keyz => { assert.equal(2, keyz.length); });
        })
    });
    it("should fail to remove unknown note", function() {
        return model.destroy("badkey12")
        .then(() => { throw new Error("should not get here"); })
        .catch(err => {
            // this is expected, so do not indicate error
        });
    })
});
});
```

And now, the test report:

```
Model Test
check keylist
✓ should have three entries
✓ should have keys n1 n2 n3
✓ should have titles Node #
read note
✓ should have proper note
✓ Unknown note should fail
change note
✓ after a successful model.update
destroy note
✓ should remove note
```

```
✓ should fail to remove unknown note
```

```
8 passing (31ms)
```

In these additional tests, we have a couple of negative tests. In each test that we expect to fail, we supply a `notekey` that we know is not in the database, and we then ensure that the model gives us an error.

Notice how the test report reads well. Mocha's design is what produces this sort of descriptive results report. While writing a test suite, it's useful to choose the descriptions so the report reads well.

Testing database models

That was good, but we obviously won't run Notes in production with the in-memory Notes model. This means that we need to test all the other models. While each model implements the same API, we can easily make a mistake in one of them.

Testing the LevelUP and filesystem models is easy, just add this to the `scripts` section of `package.json`:

```
"test-notes-levelup": "MODEL_TO_TEST=../models/notes-levelup mocha",
"test-notes-fs": "MODEL_TO_TEST=../models/notes-fs mocha",
```

Then run the following command:

```
$ npm run test-notes-fs
$ npm run test-notes-levelup
```

This will produce a successful test result.

The simplest database to test is SQLite3, since it requires zero setup. We have two SQLite3 models to test, let's start with `notes-sqlite3.js`. Add the following to the `scripts` section of `package.json`:

```
"test-notes-sqlite3": "rm -f chap11.sqlite3 && sqlite3 chap11.sqlite3
--init models/schema-sqlite3.sql </dev/null && MODEL_TO_TEST=../
models/notes-sqlite3 SQLITE_FILE=chap11.sqlite3 mocha"
```

This command sequence puts the test database in the `chap11.sqlite3` file. It first initializes that database using the `sqlite3` command-line tool. Note that we've connected its input to `/dev/null` because the `sqlite3` command will prompt for input otherwise. Then, it runs the test suite passing in environment variables required to run against the SQLite3 model.

Running the test suite does find an error:

```
$ npm run test-notes-sqlite3
..
  read note
    ✓ should have proper note
      1) Unknown note should fail
..
  7 passing (385ms)
  1 failing

1) Model Test read note Unknown note should fail:
   Uncaught TypeError: Cannot read property 'notekey' of undefined
     at Statement.<anonymous> (models/notes-sqlite3.js:72:44)
   --> in Database#get('SELECT * FROM notes WHERE notekey = ?', [
     'badkey12' ], [Function])
     at models/notes-sqlite3.js:68:16
     at models/notes-sqlite3.js:67:16
```

The failure indicators in this report are as follows:

- The lack of a checkmark in the report
- The line reading "1 failing"
- The stack trace at the end of the report

The failing test calls `model.read("badkey12")`, which we know does not exist. Writing negative tests paid off.

The failing line of code at `models/notes-sqlite3.js` (line 72) reads as follows:

```
var note = new Note(row.notekey, row.title, row.body);
```

It's easy enough to insert `"util.log(util.inspect(row));"` just before this and learn that, for the failing call, SQLite3 gave us a "row" object with the `undefined` value.

The test suite calls the `read` function multiple times with a `notekey` value that does exist. Obviously, when given an invalid `notekey` value, the query gives an empty results set and SQLite3 invokes the callback with both the `undefined` error and the `undefined` row values. This is common behavior for database modules. An empty result set isn't an error, and therefore we received no error and an `undefined` row.

In fact, we saw this behavior earlier with `models/notes-sequelize.js`. The equivalent code in `models/notes-sequelize.js` does the right thing, and it has a check which we can adapt. Let's rewrite the `read` function in `models/notes-sqlite.js` to this:

```
exports.read = function(key) {
    return exports.connectDB().then(() => {
        return new Promise((resolve, reject) => {
            db.get("SELECT * FROM notes WHERE notekey = ?",
                [ key ], (err, row) => {
                    if (err) reject(err);
                    else if (!row) {
                        reject(new Error("No note found for " + key));
                    } else {
                        var note = new Note(row.notekey,
                            row.title, row.body);
                        log('READ ' + util.inspect(note));
                        resolve(note);
                    }
                });
        });
    });
};
```

This is simple, we just check whether `row` is undefined and, if so, throw an error. While the database doesn't see an empty results set as an error, Notes does. Further, Notes already knows how to deal with a thrown error.

Make this change and the test passes.

This is the bug we referred to in *Chapter 7, Data Storage and Retrieval*. We simply forgot to check for this condition in this particular method. 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.

Now that we've fixed `models/notes-sqlite3.js`, let's also test `models/notes-sequelize.js` using the SQLite3 database. To do this, we need a connection object to specify in the `SEQUELIZE_CONNECT` variable. While we can reuse the existing one, let's create a new one. Create a file named `test/sequelize-sqlite.yaml` containing this:

```
dbname: notestest
username:
password:
params:
```

```
  dialect: sqlite
  storage: notestest-sequelize.sqlite3
  logging: false
```

This way, we don't overwrite the "production" database instance with our test suite. Since the test suite destroys the database it tests, it must be run against a database we are comfortable destroying. The logging parameter turns off the voluminous output Sequelize produces so that we can read the test results report.

Add the following to the scripts section of package.json:

```
"test-notes-sequelize-sqlite": "MODEL_TO_TEST=../models/notes-
sequelize SEQUELIZE_CONNECT=test/sequelize-sqlite.yaml mocha",
```

Then run the test suite:

```
$ npm run test-notes-sequelize-sqlite
..
8 passing (2s)
```

And, we pass with flying colors!

We've been able to leverage the same test suite against multiple Notes models. We even found a bug in one model. But, we have two test configurations remaining to test. Our test matrix reads as follows:

- models-fs: PASS
- models-memory: PASS
- models-levelup: PASS
- models-sqlite3: 1 failure, now fixed
- models-sequelize with SQLite3: PASS
- models-sequelize with MySQL: untested
- models-mongodb: untested

The two untested models both require the setup of a database server. We avoided testing these combinations because setting up the database server makes it more difficult to run the test. But our manager won't accept that excuse because the CEO needs to know we've tested Notes while configured similarly to the production environment.

In production, we'll be using a regular database server, of course, with MySQL or MongoDB—the primary choices. Therefore, we need a way that incurs a low overhead to run tests against those databases because they're the production configuration. Testing against the production configuration must be so easy that we should feel no resistance in doing so. That's to ensure that tests are run against that configuration often enough for testing to make the desired impact.

Using Docker to manage test database servers

One advantage Docker gives is the ability to install the production environment on our laptop. If the production environment is a Docker image, that image can be run just as easily on our laptop as on the cloud hosting environment. Generally speaking, it's important to replicate the production environment when running tests. Docker can make this an easy thing to do.

What we'll do in this section is demonstrate making minimal changes to the Docker environment we defined previously and develop a shell script to automate executing the Notes test suite inside the appropriate containers.

Using Docker, we'll be able to easily test against a database, and have a simple method for starting and stopping a test version of our production environment. Let's get started.

Docker Compose to orchestrate test infrastructure

We had a great experience using Docker Compose to orchestrate the Notes application deployment. The whole system, with four independent services, is easily described in `compose/docker-compose.yml`. What we'll do is duplicate that script, then make a couple small changes required to support test execution.

Let's start by making a new directory, `test-compose`, as a sibling to the `notes`, `users`, and `compose` directories. Copy `compose/docker-compose.yml` to the newly created `test-compose` directory. We'll be making several changes to this file and a couple of small changes to the existing Dockerfiles.

We want to change the container and network names so our test infrastructure doesn't clobber the production infrastructure. We'll constantly delete and recreate the test containers, so to keep the developers happy, we'll leave development infrastructure alone and perform testing on separate infrastructure. By maintaining separate test containers and networks, our test scripts can do anything they like without disturbing the development or production containers.

Consider this change to the db-auth and db-notes containers:

```
db-auth-test:  
  build: ../db-auth  
  container_name: db-auth-test  
  networks:  
    - authnet-test  
  
..  
  
db-notes-test:  
  build: ../db-notes  
  container_name: db-notes-test  
  networks:  
    - frontnet-test
```

This is the same as earlier, but with "-test" appended to container and network names.

That's the first change we must make, append `-test` to every container and network name in `test-compose/docker-compose.yml`. Everything we'll do with tests will run on completely separate containers, hostnames, and networks than that of the development instance.

This change will affect the `notesapp-test` and `userauth-test` services because the database server hostnames are now `db-auth-test` and `db-notest-test`. There are several environment variables or configuration files to update.

Previously, we defined all environment variables in the Dockerfile. But now, we want to reuse the same containers in test and production environments, with slightly tweaked environment variables to reflect where the container is executing. This raises a question over the location, Dockerfile or `docker-compose.yml`, to define a given environment variable. The environment variables which must be different between the production or test environment must be defined in the corresponding `docker-compose.yml`, while all other variables can be defined in the corresponding Dockerfile.

```
userauth-test:  
  build: ../users  
  container_name: userauth-test
```

```
environment:
  DEBUG: ""
  NODE_ENV: "test"
  SEQUELIZE_CONNECT: "userauth-test/sequelize-docker-mysql.yaml"
  HOST_USERS_TEST: "localhost"
networks:
  - authnet-test
  - notesauth-test
depends_on:
  - db-auth-test
volumes:
  - ./reports-userauth:/reports
  - ./userauth:/usr/src/app/userauth-test
.

.

notesapp-test:
  build: ../notes
  container_name: notesapp-test
  environment:
    DEBUG: ""
    NODE_ENV: "test"
    SEQUELIZE_CONNECT: "notesmodel-test/sequelize-docker-mysql.yaml"
    USER_SERVICE_URL: "http://userauth-test:3333"
  networks:
    - frontnet-test
    - notesauth-test
  expose:
    - 3000
  ports:
    - "3000:3000"
  depends_on:
    - db-notes-test
    - userauth-test
  volumes:
    - ./reports-notes:/reports
    - ./notesmodel:/usr/src/app/notesmodel-test
```

Again, we changed the container and network names to append `-test`. We moved some of the environment variables from `Dockerfile` to `docker-compose.yml`. Finally, we added some data volumes to mount host directories inside the container.

The existing variables, corresponding to the ones shown here, must be copied into `compose/docker-compose.yml`. As you do so, delete the variable definition from the corresponding Dockerfile. What we'll end up with is this arrangement:

- `compose/docker-compose.yml` holding environment variables for the production environment
- `test-compose/docker-compose.yml` holding environment variables for the test environment
- Dockerfiles hold the environment variables common to both environments

An option is to not record any environment variables in Dockerfiles, and instead put them all in the two `docker-compose.yml` files. You'd avoid the decision of what goes where but end up with duplicated variables with identical values. Forgetting to update variable definitions in both locations risks potential disasters.

Another thing to do is to set up directories to store test code. A common practice in Node.js projects is to put test code in the same directory as the application code. That would mean avoiding copying the test code to a production server, or when publishing as an npm module. As it stands the Dockerfiles simply copy everything from the notes and users directories into the corresponding containers. We can either change the Dockerfile, or we can mount the test code into the container as is done with the corresponding volume sections of `test-compose/docker-compose.yml`. That way, the test code is injected into the container rather than ignored while creating the container.

Let's start with these shell commands:

```
$ mv notes/test test-compose/notesmodel  
$ mkdir test-compose/userauth
```

The first command moves the Notes models test suite we just created into `test-compose`, and the second sets up a directory for a test suite we're about to write. With the volume definitions shown earlier, `test-compose/notesmodel` appears as `notesmodel-test` in the notes application directory, and `test-compose/userauth` appears as `userauth-test` in the users application directory.

Now add the `test-compose/userauth/sequelize-docker-mysql.yaml` file containing the following:

```
dbname: userauth  
username: userauth  
password: userauth  
params:  
  host: db-auth-test
```

```
port: 3306
dialect: mysql
logging: false
```

This is the same as `users/sequelize-docker-mysql.yaml`, but for the hostname change.

Similarly we add `test-compose/notesmodel/sequelize-docker-mysql.yaml` containing the following:

```
dbname: notes
username: notes
password: notes
params:
  host: db-notes-test
  port: 3306
  dialect: mysql
  logging: false
```

Again, this is the same as `notes/models/sequelize-docker-mysql.yaml` but for the hostname change.

Package.json scripts for Dockerized test infrastructure

Now we can add a few `package.json` lines for the Dockerized test execution. We'll see later how we'll actually run the tests under Docker.

In `notes/package.json`, add the following to the `scripts` section:

```
"test-docker-notes-sequelize-sqlite": "MODEL_TO_TEST=../models/notes-sequelize mocha -R json notesmodel-test/test-model.js >/reports/notes-sequelize-sqlite.json",
"test-docker-notes-sequelize-mysql": "MODEL_TO_TEST=../models/notes-sequelize mocha -R json notesmodel-test/test-model.js >/reports/notes-sequelize-mysql.json",
"test-docker-notes-memory": "MODEL_TO_TEST=../models/notes-memory mocha -R json notesmodel-test/test-model.js >/reports/notes-memory.json",
"test-docker-notes-fs": "MODEL_TO_TEST=../models/notes-fs mocha -R json notesmodel-test/test-model.js >/reports/notes-fs.json",
"test-docker-notes-levelup": "MODEL_TO_TEST=../models/notes-levelup mocha -R json notesmodel-test/test-model.js >/reports/notes-levelup.json",
```

```
"test-docker-notes-sqlite3": "rm -f chap11.sqlite3 && sqlite3 chap11.sqlite3 --init models/schema-sqlite3.sql </dev/null && MODEL_TO_TEST=../models/notes-sqlite3 SQLITE_FILE=chap11.sqlite3 mocha -R json notesmodel-test/test-model.js >/reports/notes-sqlite3.json"
```

We removed the `SEQUELIZE_CONNECT` variable because it's now defined in `test-compose/docker-compose.yml`.

The Mocha invocation is different now. Previously, there'd been no arguments, but now, we're executing it with `-R json notesmodel-test/test-model.js` and then redirecting `stdout` to a file. Because the test is no longer in the `test` directory, we must explicitly instruct Mocha the filename to execute. We're also, with the `-R` option, using a different results reporting format. You can leave the test results reporting as they are, but the test results would be printed on the screen mixed together with a bunch of other output. There'd be no opportunity to collect results data for publishing in a report or showing a success or failure badge on a project dashboard website. The large amount of output might make it hard to spot a test failure.

Mocha supports different reporter modules that print results in different formats. So far, we used what Mocha calls the spec reporter. The HTML reporter is useful for generating test suite documentation. With the JSON reporter (`-R json`), test results are printed as JSON on the `stdout`. We're redirecting that output to a file in `/reports`, a directory which we've defined in the `volumes` section.

The `test-compose/docker-compose.yml` file contains volume declarations connecting the `/reports` container directory to a directory on the host filesystem. What will happen is these files will be stored on the host in the named directories, letting us easily access the JSON test results data so that we can make a report.

In `users/package.json`, let's make a similar addition to the `scripts` section:

```
"test-docker": "mocha -R json userauth-test/test.js >/reports/userauth.json"
```

We still haven't written the corresponding test suite for the user authentication REST service.

Executing tests under Docker Compose

Now we're ready to execute some of the tests inside a container. In `test-compose`, let's make a shell script called `run.sh`:

```
docker-compose up --build --force-recreate -d  
docker exec -it notesapp-test npm install mocha@2.4.5 chai@3.5.0
```

```

docker exec -it notesapp-test npm run test-docker-notes-memory
docker exec -it notesapp-test npm run test-docker-notes-fs
docker exec -it notesapp-test npm run test-docker-notes-levelup
docker exec -it notesapp-test npm run test-docker-notes-sqlite3
docker exec -it notesapp-test \
    npm run test-docker-notes-sequelize-sqlite
docker exec -it notesapp-test \
    npm run test-docker-notes-sequelize-mysql

docker-compose stop

```



[It's common practice to run tests out of a continuous integration system such as Jenkins. Continuous integration systems automatically run builds or tests against software products. The build and test results data is used to automatically generate status pages. Visit <https://jenkins.io/index.html>, which is a good starting point for a Jenkins job.]

After quite a lot of experimentation, these "docker-compose up" options were found to most reliably execute the tests. These options ensure that the images are rebuilt and new containers are built from the images. The "-d" option puts the containers in the background, so the script can go on to the next step and execute the tests.

Next, the script uses "docker exec" to execute commands inside the notesapp-test container. With the first we ensure that Mocha and Chai are installed, and with the subsequent commands, we execute the test suite. We ran these tests earlier outside the container, which was easier to do but at the cost of test execution running in a different environment than we have in production.

We've also been able to add test execution on the Sequelize MySQL combination. If you remember, that combination was left out of our test matrix earlier because it was "too difficult" to set up a test database. With `test-compose/docker-compose.yml`, we no longer have that excuse. But, we're still a little lazy because we've left the MongoDB model untested.

Testing on MongoDB would simply require defining a container for the MongoDB database and a little bit of configuration. Visit https://hub.docker.com/_/mongo/ for the official MongoDB container. We'll leave this as an exercise for you to try.

To run the tests, simply type:

```
$ sh -x run.sh
```

Lots of output will be printed concerning building the containers and executing the test commands. The test results will be left in the directories named as volumes in `test-compose/docker-compose.yml`.

Testing REST backend services

It's now time to turn our attention to the user authentication service. We've mentioned tests of this service, saying that we'll get to them later. "Later" is now, it seems. While we can test this service as we did for the Notes models, which would be to just call the methods and check the results, we have an opportunity to test its REST API instead. The customer of this service, the Notes application, uses it through the REST API, giving us a perfect rationalization to test using REST.

The approach we'll take is to use Mocha and Chai as we did earlier using the `restify` client to make REST calls inside test cases.

We've already made the `test-compose/userauth` directory. In that directory, create a file named `test.js`:

```
'use strict';

const assert = require('chai').assert;
const restify = require('restify');
const url = require('url');

var usersClient;

describe("Users Test", function() {
    before(function() {
        usersClient = restify.createJsonClient({
            url: url.format({
                protocol: 'http',
                hostname: process.env.HOST_USERS_TEST,
                port: process.env.PORT
            }),
            version: '*'
        });
        usersClient.basicAuth('them',
            'D4ED43C0-8BD6-4FE2-B358-7C0E230D11EF');
    });
});
```

This sets up Mocha and the Restify client. The `HOST_USERS_TEST` environment variable specifies the hostname to run the test against. This variable was already set in `test-compose/docker-compose.yml` to "localhost". The `before` hook, used to set up the REST client, is run once at the beginning of the test suite:

```
beforeEach(function() {
    return new Promise((resolve, reject) => {
```

```

usersClient.post('/find-or-create', {
    username: "me", password: "w0rd", provider: "local",
    familyName: "Einarrsdottir", givenName: "Ashildr",
    middleName: "", emails: [], photos: []
},
(err, req, res, obj) => {
    if (err) reject(err);
    else resolve();
});
});
});

afterEach(function() {
    return new Promise((resolve, reject) => {
        usersClient.del('/destroy/me', (err, req, res, obj) => {
            if (err) reject(err);
            else resolve();
        });
    });
});
});

```

This uses the API to create our test account before each test case execution, then used again later to delete that account:

```

describe("List user", function() {
    it("list created users", function() {
        return new Promise((resolve, reject) => {
            usersClient.get('/list', (err, req, res, obj) => {
                if (err) reject(err);
                else if (obj.length <= 0)
                    reject(new Error("no users found"));
                else resolve();
            });
        });
    });
});

```

Now, we can turn to testing some API methods, such as the `/list` operation.

We'd already guaranteed that there is an account, in the `before` method, so `/list` should give us an array with at least one entry.

This particular testcase can be written using Mocha's support for asynchronous test cases rather than using a Promise object.

```
describe("List user", function() {
    it("list created users", function(done) {
        usersClient.get('/list', (err, req, res, obj) => {
            if (err) done(err);
            else if (obj.length <= 0)
                done(new Error("no users found"));
            else done();
        });
    });
});
```

You should take the approach you prefer. Maybe as you grow comfortable with the Promise object, you'll start applying it everywhere.

This approach uses the traditional Node.js asynchronous coding model. The two are equivalent, other than the lines of code. Perhaps, for this test, this implementation is preferable:

```
describe("find user", function() {
    it("find created users", function() {
        return new Promise((resolve, reject) => {
            usersClient.get('/find/me', (err, req, res, obj) => {
                if (err) reject(err);
                else if (!obj)
                    reject(new Error("me should exist"));
                else resolve();
            });
        });
    });
    it("fail to find non-existent users", function() {
        return new Promise((resolve, reject) => {
            usersClient.get('/find/nonExistentUser',
            (err, req, res, obj) => {
                if (err) resolve();
                else if (!obj) resolve();
                else reject(new Error("nonExistentUser should not
exist")));
            });
        });
    });
});
```

We can check the /find operation in two ways, looking for the account we know exists, and for the one we know does not exist. In the first case, we indicate failure if the user account is for some reason not found. In the second, we indicate failure if the user account is found:

```
describe("delete user", function() {
    it("delete nonexistent users", function() {
        return new Promise((resolve, reject) => {
            usersClient.del('/destroy/nonExistentUser',
            (err, req, res, obj) => {
                if (err) resolve();
                else reject(new Error("Should throw error"));
            });
        });
    });
});
```

Finally, we should check the /destroy operation. We already check this operation in the after method, where we destroy a known user account. We need to also perform the negative test and verify its behavior against an account we know does not exist.

We have already added configuration to `test-compose/docker-compose.yml` and the necessary script in `users/package.json`. Therefore, we go ahead and add these lines to the `run.sh` file:

```
docker exec -it userauth-test npm install mocha@2.4.5 chai@3.5.0
docker exec -it userauth-test npm run test-docker
```

Then, execute `run.sh` to rerun the tests, and you'll see this new test run giving its results.

Frontend headless browser testing with CasperJS

A big cost area in testing is manual tests of the user interface. Therefore, a wide range of tools have 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. ZombieJS is a popular tool that implements a simulated browser which we can direct to visit web pages and perform simulated clicks on simulated buttons, verifying application behavior by inspecting the resulting behavior.

However, for this section, we'll use CasperJS (<http://casperjs.org>). This tool runs on top of PhantomJS, which is a headless version of Webkit. Webkit is a real browser engine that's at the heart of the Safari browser. PhantomJS encapsulates Webkit so it runs completely headless, meaning that there is no GUI window popping up on the screen during test execution. It otherwise behaves exactly as a web browser, and CasperJS can even generate screenshots if its desired to verify visual elements.

While CasperJS is installed using npm, and CasperJS scripts are written in JavaScript that looks like Node.js, the CasperJS website repeatedly says that this is not Node.js. ES-2015 constructs do not work, and while there's a require statement it doesn't access Node.js packages. Fortunately, it comes with a fairly rich API that supports a wide range of tasks including screen scraping and UI test automation.

CasperJS scripts are run using the `casperjs` command. When run as `casperjs test`, additional methods are made available that are useful for functional testing.

Setup

Let's first set up the directory and other configurations:

```
$ mkdir test-compose/notesui
```

Then in `test-compose/docker-compose.yml`, update the volumes as follows:

```
notesapp-test:  
  ..  
  volumes:  
    - ./reports-notes:/reports  
    - ./notesui:/usr/src/app/notesui-test  
    - ./notesmodel:/usr/src/app/notesmodel-test
```

This ensures that the directory just created appears as `notesui-test` in the application directory.

In `run.sh`, add this line to install CasperJS and PhantomJS:

```
docker exec -it notesapp-test npm install -g phantomjs-prebuilt@2.1.7  
casperjs@1.1.0-beta5
```

In the script we're about to write, we need a user account that we can use to log in and perform some actions. Fortunately, we already have a script to set up a test account. In `users/package.json`, add this line to the `scripts` section:

```
"setupuser": "PORT=3333 node users-add",
```

Then in `notes/package.json`, add this line to the `scripts` section:

```
"test-docker-ui": "cd notesui-test && casperjs test uitest.js"
```

We're about to write this test script, but let's finish the setup, the final bit of which is adding these lines to `run.sh`:

```
docker exec -it userauth-test npm run setupuser
docker exec -it notesapp-test npm run test-docker-ui
```

When executed, these two lines ensure that the test user is set up, and it then runs the user interface tests.

Improving testability in Notes UI

Adding a few `id` or `class` attributes to HTML elements can improve testability. In the test we're about to write, we'll inspect the HTML to ensure that the browser has gone to the right page, and that the application is in the expected state. We'll also be clicking on buttons. Both of these tasks are made easier if we can refer to certain HTML elements by an `id`.

In `notes/views/pageHeader.ejs`, change these lines:

```
<% if (user && typeof hideAddNote === 'undefined') { %>
<a class="btn btn-primary" id="btnaddnote" href="/notes/add">
  ADD Note</a>
<% } %>

<% if (user) { %>
<a class="btn btn-primary" id="btnlogout" href="/users/logout">
  Log Out <span class="badge"><%= user.username %></span></a>
<% } else { %>
<a class="btn btn-primary" id="btnloginlocal" href="/users/login">Log
in</a>
<a class="btn btn-primary" id="btnlogintwitter" href="/users/
auth/twitter">Log in with Twitter</a>
<% } %>
```

In `notes/views/noteview.ejs`, make these changes:

```
<a class="btn btn-default" id="btndestroynote" href="/notes/
destroy?key=<%= notekey %>" role="button">Delete</a>
<a class="btn btn-default" id="btneditnote" href="/notes/edit?key=<%
notekey %>" role="button">Edit</a>
```

In both cases, we added the `id` attributes. In writing the test code, the `id` attribute made it easier to check for or click on these buttons.

CasperJS test script for Notes

In `test-compose/notesui`, create a file named `uitest.js` containing the following:

```
var notes = 'http://localhost:3000';
casper.test.begin('Can login to Notes application',
    function suite(test) {
        casper.start(notes, function() {
            test.assertTitle("Notes");
            test.assertExists('a#btnloginlocal',
                "Login button is found");
            this.click("a#btnloginlocal");
        });
    ...
    casper.run(function() {
        test.done();
    });
});
```

The `casper.test.begin` function contains a block of tests. You see that the callback is given a `test` object that has methods useful for testing. Inside this block, certain `casper` methods are called, the result of which is to create a queue of test instructions to perform. That queue is executed when `casper.run` is called.

The `casper.start` function must be called, well, at the start of the script. You give it a URL, as shown here, and you can optionally pass in a callback function.

The `casper.run` function must be called at the end of the script. In between these two, you'll write steps of the test scenario.

In callback functions, you can make the `test` assertions or you can call other `casper` functions that are available on `this`. In this case, the browser is on the Notes home page and we're checking a couple things to ensure that this is where the browser is located and that the browser is not logged in.

For example, because of the change we made to `notes/views/pageHeader.ejs`, the presence of `#btnloginlocal` is a sure indication that Notes is not logged in. If it were logged in, that button would not be visible and `#btnlogout` would be present instead.

When you take an action that causes an asynchronous request and response, it's necessary to start a new navigation step, that is the following:

```
casper.then(function() {
    test.assertHttpStatus(200);
    test.assertUrlMatch('/users\\login/');
```

```
'should be on /users/login');

this.fill('form', {
    username: "me",
    password: "wOrd"
});
this.click('button[type="submit"]');
});
```

We have just told the browser to click on the **Login** button. We should then end up on the `/users/login` page. We then put login parameters into the form inputs, and click on the **Submit** button.

This step is why we needed the `setupuser` script to be run:

```
casper.waitForSelector('#btnlogout', function() {
    test.assertHttpStatus(200);
    test.assertTitle("Notes");
    test.assertExists('a#btnlogout', "logout button is found");
    test.assertExists('a#btndonnote', "Add Note button is found");
    this.click("#btndonnote");
});
```

Once we click on the login form, we need to wait for the screen to refresh. It should, of course, navigate to the Notes home page. The `waitForSelector` method waits until an element with that selector is available on the page.

In CasperJS, many functions take selector parameters. These are used to select elements in the DOM of the current page. By default, it takes CSS3 selector strings, but you can also use XPath selectors.

The final step is to click on the **Add Note** button.

```
casper.waitForUrl('/notes/\add/', function() {
    test.assertHttpStatus(200);
    test.assertTitle("Add a Note");
    test.assertField("docreate", "create");
    this.fill('form', {
        notekey: 'testkey',
        title: 'Test Note Title',
        body: 'Test Note Body with various textual delights'
    });
    this.click('button[type="submit"]');
});
```

And, of course, the response to that is to go to /notes/add. We again check a few things to make sure that the browser has gone to the correct page. We then fill in the entry form with a dummy note and click on the **Submit** button:

```
casper.waitForUrl('/notes/view', function() {
  test.assertHttpStatus(200);
  test.assertTitle("Test Note Title");
  test.assertSelectorHasText("p#notebody",
    'Test Note Body with various textual delights');
  this.click('#btndestroynote');
});
```

The browser should of course go to /notes/view. We're checking a few parameters on the screen to verify this. We then click on the **Destroy** button:

```
casper.waitForUrl('/notes/destroy', function() {
  test.assertHttpStatus(200);
  test.assertTitle("Test Note Title");
  test.assertField("notekey", "testkey");
  this.click('input[type="submit"]');
});
```

Once the browser gets to /notes/destroy, we check that the page indeed has all the right elements. We can click on the **Submit** button to verify that the note should be deleted:

```
casper.waitForUrl(notes, function() {
  test.assertHttpStatus(200);
  test.assertTitle("Notes");
  test.assertExists('a#btnlogout', "logout button is found");
  this.click("#btnlogout");
});
```

The browser should again be on the home page. Let's now verify the ability to log out by clicking on the **Logout** button:

```
casper.waitForUrl(notes, function() {
  test.assertHttpStatus(200);
  test.assertTitle("Notes");
  test.assertExists('a#btnloginlocal', "Login button is found");
});
```

The browser should again be on the home page, but logged out. We distinguish between being logged-in and logged-out by which buttons are present.

Running the UI test with CasperJS

Now that you have the test entered, we can run it. Looking at `run.sh`, these steps will run just the UI test:

```
docker-compose stop
docker-compose up --build --force-recreate -d
docker exec -it notesapp-test npm install -g phantomjs-prebuilt@2.1.7
casperjs@1.1.0-beta5
docker exec -it userauth-test npm run setupuser
docker exec -it notesapp-test npm run test-docker-ui
```

The last step of which will look like this:

```
$ docker exec -it notesapp-test npm run test-docker-ui
npm info it worked if it ends with ok
npm info using npm@3.7.3
npm info using node@v5.9.0
npm info lifecycle notes@0.0.0~pretest-docker-ui: notes@0.0.0
npm info lifecycle notes@0.0.0~test-docker-ui: notes@0.0.0

> notes@0.0.0 test-docker-ui /usr/src/app
> cd notesui-test && casperjs test uitest.js

Test file: uitest.js
# Can login to Notes application
PASS Can login to Notes application (5 tests)
PASS Page title is: "Notes"
PASS Login button is found
PASS HTTP status code is: 200
PASS should be on /users/login
PASS HTTP status code is: 200
PASS Page title is: "Notes"
PASS logout button is found
PASS Add Note button is found
PASS HTTP status code is: 200
PASS Page title is: "Add a Note"
PASS "doCreate" input field has the value "create"
PASS HTTP status code is: 200
```

```
PASS Page title is: "Test Note Title"
PASS Find "Test Note Body with various textual delights" within the
selector "p#notebody"
PASS HTTP status code is: 200
PASS Page title is: "Test Note Title"
PASS "notekey" input field has the value "testkey"
PASS HTTP status code is: 200
PASS Page title is: "Notes"
PASS logout button is found
PASS HTTP status code is: 200
PASS Page title is: "Notes"
PASS Login button is found
PASS 23 tests executed in 88.227s, 23 passed, 0 failed, 0 dubious, 0
skipped.
```

While this report output looks cool, you probably want to generate a test results data file so your continuous build system (Jenkins, et al.) can display a green or red dot on the dashboard as appropriate. CasperJS supports outputting XUnit results as follows:

```
"test-docker-ui": "cd notesui-test && casperjs test uitest.js --xunit=/reports/notesui.xml"
```

Earlier we configured Mocha to output JSON that theoretically we can use to generate a test results report. In this case XUnit test results can be used the same way.

Summary

We've covered a lot of territory in this chapter, looking 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 which does not follow good testing practices is often bogged down with fixing regression after regression.

Testing is of course a large topic area, but with this information, you can take the steps required to improve your application quality.

We've 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 long ways 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.

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.

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 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. We can't trust that unit testing will find every bug; some bugs will only show up in the web browser.

Even so, we've only touched the beginning of what could be tested in Notes.

In this book, we've covered the gamut of Node.js development, giving you a strong foundation from which to start developing Node.js applications.

What is Node.js, and why is it an important programming platform? As professional software engineers, is it a good idea to jump onto any new programming platform just because it's new and cool? No, we should evaluate their value in implementing our applications.

Once you've decided to use Node.js, we've given you a soup-to-nuts introduction to developing software on this platform. We started with developing and using Node.js modules and using the npm package manager to manage the modules used in your application.

Express is a popular framework for developing web applications in Node.js, and we spent many chapters developing and refining an application. While Notes started as a modest note-taking application that lost everything you wrote when the server restarted, it became a full-fledged application that even included real-time commenting between multiple people. Along the way, we used several database engines (MySQL, SQLite, and MongoDB) for data storage, and the Socket.IO library for real-time communication between users. We integrated user authentication, and we looked at what's required to use OAuth2 to authenticate users from other services such as Twitter.

Distributed microservice architectures are very popular today, if only because it's a very agile approach to writing software systems. To that end, we used two different methods to develop RESTFUL services using Node.js, and we used Docker to automate not only the deployment of the application stack, but the testing infrastructure.

Index

Symbols

*BSD

Node.js, installing 18

Docker

for microservice deployment 276-278

A

absolute module identifiers 42

algorithm

application directory structure,
example 44, 45
module identifiers 42, 43
path names 42, 43
used, for resolving module 42

application directory structure

example 44, 45

assert module

used, for testing 315

asynchronous code

about 152
testing 313, 314

AuthNet

creating, for User Authentication
service 282
implementing 288-290

B

Babel

URL 33
URL, for setup 33
used, for rewriting JavaScript code 33, 34

Backbone.js

URL 234

body-parser module

URL 76

boot2docker

URL 278

Bootstrap

adding, to application templates 134-136
grid foundation, laying 136-139
URL 134

Bootstrap Jumbotron component 221

Bower

URL 132
using 132

Breadcrumb component

using 141-143

breakpoint 130

C

callback hell 100

CasperJS

setting up 338, 339
testability, improving 339
test script, writing 340-342
UI test, executing 343, 344
used, for testing frontend headless
browser 337, 338

Chai

URL 318
used, for testing 316, 317

ChakraCore JavaScript engine

URL 2

Chrome Developer Tools 128

Collection object

URL 185

Comet

about 234

reference link 234

command-line tools

- about 26, 27
- running 26
- script, executing 28, 29
- server, launching 29
- testing 26

CommonJS

- about 57
- URL 57

cookie-parser module

- URL 76

CRUD model 104

CSS-Tricks blog

- URL 131

Cursor object

- URL 187

customized Bootstrap

- building 146-148
- Cyborg theme, using 148, 149
- references 148

Cyborg theme

- using 148

D

data models

- testing 316, 323-327
- testing, with Chai 316, 317
- testing, with Mocha 316, 317
- tests, adding 321-323
- tests, configuring 320, 321
- tests, executing 320, 321
- test suite 317-320

data storage 152

Debug package

- URL 153
- using 153

default Express application 74-76

developer tools

- development instances, installing
 - with nvm 23-25
- installing, from source for all POSIX-like systems 22, 23
- installing, on Mac OS X 21

Digital Ocean

- URL 267

directories

- as modules 41

div elements

- URL 256

Docker

- about 276
- containers, creating 281, 282
- installing 278, 279
- package.json scripts, adding for Dockerized test infrastructure 331, 332
- remote access, configuring 296
- starting, for Windows/Mac 280, 281
- URL 276-278
- used, for managing test database servers 327

Docker Compose

- about 277
- cloud, deploying 302
- cloud hosting, deploying 307-312
- compose files 302-306
- Notes application, executing 306, 307
- tests, executing 332, 333
- URL 302
- used, for orchestrating test infrastructure 327-331

Docker Engine 277

Dockerfile

- using 276

Docker Machine

- about 277
- used, for starting Docker 279, 280

Docker Toolbox

- used, for starting Docker 279, 280
- VirtualBoxMachine, exploring 298, 299

Droplet 308

E

ECMAScript 6

- and Node.js 31-33
- Babel, using 33, 34
- URL 31

EJS template engine

- URL 74

ES-2015 multiline 65, 66

ES-2015 Promises

- about 99-101

asynchronous code, writing 102, 103
error handling 101, 102
references 103
tools, using 103

EventEmitters
events, receiving 60, 61
events, sending 60, 61
theory 61, 62

Express
about 70
API documentation, URL 75
error handling 79
installing 70-73
middleware function 76, 77
MVC model 104
router functions 99-101
Socket.IO, initializing with 235-239
URL 70

Express application
Fibonacci application, refactoring 95, 96
Fibonacci sequence, calculating with 79-84
REST backend service, calling 91
REST server, implementing 91-94
theming 121, 122

express-authentication
URL 190

F

Fibonacci application
refactoring 95, 96

Fibonacci sequence
algorithmic refactoring 86-88
calculating, with Express application 79-84
computationally intensive code,
executing 84-86
HTTP Client requests, creating 88-91
Node.js event loop 84-86

file module
about 39, 40
module-level encapsulation,
demonstrating 40

file-stream-rotator
URL 155

filesystem
notes, storing 157-163

FrontNet
creating 291
Dockerized Notes app, accessing 296
dockerizing 292, 293
Docker Toolbox VirtualBoxMachine,
exploring 298, 299
implementing 294, 295
location of MySQL data volumes,
controlling 299-302
MySQL, using 291, 292
remote access, configuring 296
remote access, configuring in
VirtualBox 297, 298

fs module
URL 158

functional testing 313

G

Google's Hosted Libraries
URL 122

H

Hexy 30

Homebrew
Node.js, installing with 17
URL 17

HTTP server applications 62-65

HTTP Sniffer
HTTP conversation, listening 67-69

I

inter-user chat

data model, implementing for storing
messages 249-252
implementing 249
messages, adding to Notes router 252-254
messages, deleting 256-260
messages, displaying 256-260
messages, passing 260, 261
messages, sending 256-260
Modal window, using to compose
messages 254, 256
note view template, modifying for
messages 254

J

jQuery
URL 122

K

Kitematic GUI 278

L

LevelUP data store
notes, storing with 163-167

Linux
Node.js, installing 17, 18

Linux Node.js service deployment
demonstrating 265, 266
Notes application, setting up 268-271
PM2, setting up 272-274
prerequisite 266, 267
User Authentication, setting up
on server 268-271

logbook module
about 155
URL 155

logging
implementing 152, 153
messages, debugging 155
request logging, with Morgan 153-155
stderr, capturing 155
stdout, capturing 155
uncaught exceptions, handling 156

login support, for Notes application
implementing 207
login/logout, modifying in
 routes/index.js 216
login/logout, modifying in
 routes/notes.js 216-218
login/logout, modifying to app.js 214, 215
login routing function 211-214
logout routing function 211-214
Notes application, executing 221-223
template changes, viewing 218-221
user authentication REST API,
 accessing 207-211

Long Term Support (LTS)
about 25
URL 25

Loopback

about 97
URL 97

LSB-style init script
URL 266

M

Mac OS X

developer tools, installing 21
Node.js, installing with Homebrew 17
Node.js, installing with MacPorts 16

MacPorts

Node.js, installing with 16
URL 16

MariaDB

URL 174
middleware function
about 76, 77
request paths 77, 78
URL 76

mobile-first design, for Notes application
add/edit note form, cleaning up 144, 145
Bootstrap grid foundation, laying 136-139
Breadcrumb component, using 141-143
building 136
notes list, improving 140, 141

mobile-first paradigm 129-131

Mocha
URL 316
used, for testing 316, 317

Modal windows

usage 261, 262

module
defining 37-39
directories 41
file module 39, 40
format 39
resolving, with algorithm 42

module identifiers

about 42, 43
absolute module identifiers 42
relative module identifiers 42
top-level module identifiers 42

module-level encapsulation

demonstrating 40

- MongoDB**
model, creating 183-187
Notes application, executing 187, 188
notes, storing 182, 183
URL 182
- Mongolab**
URL 182
- Mongoose**
URL 182
- Morgan**
used, for request logging 153-155
- Morgan request logger**
URL 76
- MySQL**
for User Authentication service 282-285
URL 174, 181
- ## N
- namespace** 243
- native code module** 20
- node-gyp**
about 20
URL 20
- Node.js**
about 1
and ECMAScript 6 31-33
capabilities 3
installing, on Ubuntu 268
server-side JavaScript 4
system requisites 15
URL 8, 167, 268
URL, for case study 9
URL, for downloading 22
URL, for installation 19
- Node.js, advantages**
about 4
asynchronous event-driven I/O 5-7
asynchronous event-driven model 5
event handling 9, 10
hostile fork 5-7
JavaScript support 4
microservice architecture 5, 12, 13
performance 8, 9
popularity 4
server utilization 11
testable systems 12, 13
- Twelve-Factor app model** 13
utilization 8, 9
V8, surpassing 5
web hosting 11
- Node.js installation**
from nodejs.org 18, 19
from source, on POSIX-like systems 19
on *BSD 17, 18
on Linux 17, 18
on Mac OS X, with Homebrew 17
on Mac OS X, with MacPorts 16
on Windows 17, 18
package managers, using 16
prerequisites 19
- nodejs.org**
Node.js distribution, installing 18, 19
URL 18
- notes**
storing, in filesystem 157-163
storing, in MongoDB 182, 183
storing, with LevelUP data store 163-167
storing, with Sequelize module 174, 175
storing, with SQLite3 167
- Notes application**
architecture 264, 265
Bootstrap, adding to application
templates 134-136
commenting 249
creating 104, 105
executing, with MongoDB 187, 188
executing, with Sequelize module 180-182
executing, with SQLite3 173, 174
existing note, editing 117, 118
home page, creating 108-111
inter-user chat, implementing 249
multiple Notes instances,
executing 123, 124
note, adding 112-115
notes, deleting 119, 120
Notes model, creating 106-108
notes, Viewing 116, 117
problem, quantifying 128, 129
real time updates, implementing 239
Twitter Bootstrap, using 131
Twitter login support, implementing 224
- NPM** 30, 31
- npm command** 49

npm package
about 45
dependencies, maintaining 51
format 45-47
initializing 50, 51
installing 49
installing, from npm repository 54
Node.js version compatibility, declaring 53
npm commands 49
outdated packages, updating 53
package dependencies, updating
for bugs 52
publishing 54
searching 47, 48
URL 46, 47
URL, for publishing 54
version numbers, displaying 55, 56

npm repository
URL 46

nvm
URL 24
used, for installing development
instances 23-25

O

OpenSSL
URL 15

P

package manager
URL 17
used, for installing Node.js 16

Passport
URL 190

path names 42, 43

PM2
setting up 272-274
Twitter support, for hosted Notes
app 275, 276
URL 272

POSIX-like systems
Node.js, installing from source 19

Promise class 99

Promise objects
about 101, 102
Fulfilled state 101

Pending state 101
Rejected state 101

Pyramid of Doom 99, 102

Python
URL 15

R

real time updates
EventEmitter class, implementing 239-242
home page template, modifying 244
implementing, on Notes home page 239
modifying 242, 243
Notes application, executing 245
Notes application, executing while
viewing notes 249
notes, viewing 245-247
note view template, modifying 247, 248

relative module identifiers 42

Representational State Transfer (REST) 88

REST backend services
testing 334-337

RESTful frameworks 97

RESTful modules 97

Restify
about 97
URL 97, 191

router functions
using 99-101

S

ScaleGrid.io
URL 182

Semantic Versioning model
URL 55

Sequelize module
about 174
database connection, configuring 179, 180
model, creating 175-179
Notes application, executing with 180-182
notes, storing with 174, 175
references 176
URL 174, 177

server-side JavaScript 4

Socket.IO
about 235
initializing, with Express 235-239

- URL 234
- source**
- Node.js, installing on POSIX-like systems 19
- SQLite3**
- database scheme 167-169
 - model code 169-173
 - Notes application, executing 173, 174
 - notes, storing with 167
 - URL 167
- Supervisord**
- URL 272
- ## T
- template strings** 65, 66
- top-level module identifiers** 42
- Twelve-Factor app model**
- about 13, 51
 - URL 13
- Twenty Twelve theme**
- about 130
 - URL 130
- Twitter**
- URL 224
- Twitter Bootstrap**
- setting up 132-134
 - URL 131
 - using, on Notes application 131
- Twitter login support**
- application, registering with Twitter 224, 225
 - for Notes application 224
 - TwitterStrategy, implementing 225-230
- ## U
- Ubuntu**
- Node.js, installing 268
- unit testing** 313
- User Authentication service**
- AuthNet, creating 282
 - dockerizing 286-288
 - with MySQL 282-285
- user information microservice**
- creating 190-192
 - REST server, using 197-204
- user authentication server,
administering 204-207
- user authentication server, testing 204-207
- user information model, creating 192-197
- ## V
- versions policy**
- about 25
 - considerations 25
- VirtualBox**
- remote access, configuring 297, 298
 - using 266
- Virtual Private Service (VPS)** 266
- ## W
- web application frameworks**
- using 69, 70
- Windows**
- Node.js, installing 18
- Windows/Mac**
- Docker, starting 280, 281
- ## Y
- Yahoo! scale Node.js**
- URL 9
- YAML**
- reference link 175