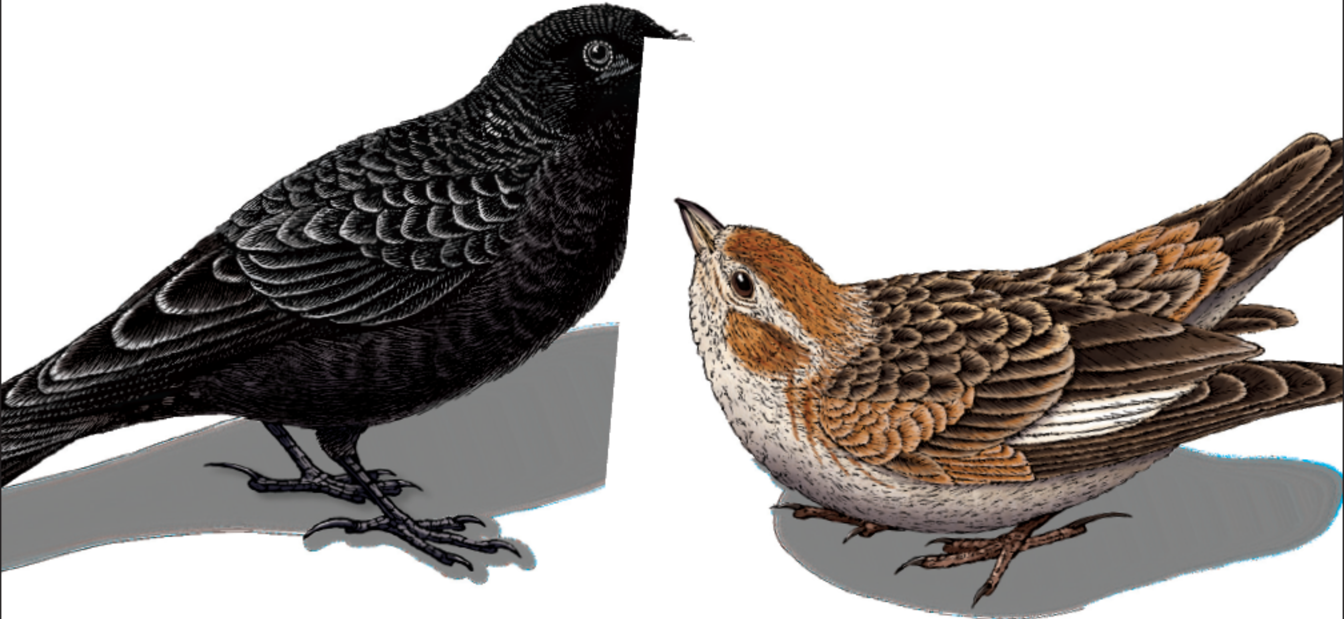
O'REILLY®

Second Edition

Web Development with Node & Express

Leveraging the JavaScript Stack



Ethan Brown

O'REILLY®

Web Development with Node & Express

Build dynamic web applications with Express, a key component of the Node/JavaScript development stack. In this updated edition, author Ethan Brown teaches you Express 5 fundamentals by walking you through the development of an example application. This hands-on guide covers everything from server-side rendering to API development suitable for use in single-page apps (SPAs).

Express strikes a balance between a robust framework and no framework at all, allowing you a free hand in your architecture choices. Frontend and backend engineers familiar with JavaScript will also learn best practices for building multipage and hybrid web apps with Express. Pick up this book and discover new ways to look at web development.

- Create a templating system for rendering dynamic data
- Dive into request and response objects, middleware, and URL routing
- Simulate a production environment for testing
- Persist data in document databases with MongoDB and relational databases with PostgreSQL
- Make your resources available to other programs with APIs
- Build secure apps with authentication, authorization, and HTTPS
- Integrate with social media, geolocation, and more
- Implement a plan for launching and maintaining your app
- Learn critical debugging skills

> "Ethan is outstanding in that he doesn't assume what his audience does or doesn't know. I was happy to see how he managed to not only do a great intro to the NodeJS/ Express ecosystem but also took the time to teach relevant web development concepts for beginners such as persistence, middleware, and Git."
>
> **—Alejandra Olvera-Novack**
> AWS Developer Relations

Ethan Brown is director of technology at VMS, where he's responsible for the architecture and implementation of VMSPro, cloud-based software for decision support, risk analysis, and creative ideation for large projects. With over 20 years of programming experience from embedded to the web, Ethan has embraced the JavaScript stack as the web platform of the future.

WEB PROGRAMMING / JAVASCRIPT

US $49.99 CAN $65.99
ISBN: 978-1-492-05351-4

5 4 9 9 9
9 781492 053514

Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

Web Development with Node and Express

Leveraging the JavaScript Stack

Ethan Brown

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

Web Development with Node and Express
by Ethan Brown

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jennifer Pollock
Developmental Editor: Angela Rufino
Production Editor: Nan Barber
Copyeditor: Kim Wimpsett
Proofreader: Sharon Wilkey
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2019: Second Edition

Revision History for the Second Edition
2019-11-12: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492053514* for release details.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-05351-4

[LSI]

This book is dedicated to my family:

My father, Tom, who gave me a love of engineering; my mother, Ann, who gave me a love of writing; and my sister, Meris, who has been a constant companion.

Table of Contents

Preface

Who This Book Is For

This book is for programmers who want to create web applications (traditional websites; single-page applications with React, Angular, or Vue; REST APIs; or anything in between) using JavaScript, Node, and Express. One of the exciting aspects of Node development is that it has attracted a whole new audience of programmers. The accessibility and flexibility of JavaScript have attracted self-taught programmers from all over the world. At no time in the history of computer science has programming been so accessible. The number and quality of online resources for learning to program (and getting help when you get stuck) is truly astonishing and inspiring. So to those new (possibly self-taught) programmers, I welcome you.

Then, of course, there are the programmers like me, who have been around for a while. Like many programmers of my era, I started off with assembler and BASIC and went through Pascal, C++, Perl, Java, PHP, Ruby, C, C#, and JavaScript. At university, I was exposed to more niche languages such as ML, LISP, and PROLOG. Many of these languages are near and dear to my heart, but in none of these languages do I see so much promise as I do in JavaScript. So I am also writing this book for programmers like myself, who have a lot of experience and perhaps a more philosophical outlook on specific technologies.

No experience with Node is necessary, but you should have some experience with JavaScript. If you're new to programming, I recommend Codecademy. If you're an intermediate or experienced programmer, I recommend my own book, *Learning JavaScript, 3rd Edition* (O'Reilly). The examples in this book can be used with any system that Node works on (which covers Windows, macOS, and Linux, among others). The examples are geared toward command-line (terminal) users, so you should have some familiarity with your system's terminal.

Most important, this book is for programmers who are excited. Excited about the future of the internet and want to be part of it. Excited about learning new things, new techniques, and new ways of looking at web development. If, dear reader, you are not excited, I hope you will be by the time you reach the end of this book….

Notes on the Second Edition

It was a joy to write the first edition of this book, and I am to this day pleased with the practical advice I was able to put into it and the warm response of my readers. The first edition was published just as Express 4.0 was released from beta, and while Express is still on version 4.x, the middleware and tools that go along with Express have undergone *massive* changes. Furthermore, JavaScript itself has evolved, and even the way web applications are designed has undergone a tectonic shift (away from pure server-side rendering and toward single-page applications [SPAs]). While many of the principles in the first edition are still useful and valid, the specific techniques and tools are almost completely different. A new edition is overdue. Because of the ascendancy of SPAs, the focus of this second edition has also shifted to place more emphasis on Express as a server for APIs and static assets, and it includes an SPA example.

How This Book Is Organized

Chapter 1 and Chapter 2 will introduce you to Node and Express and some of the tools you'll be using throughout the book. In Chapter 3 and Chapter 4, you start using Express and build the skeleton of a sample website that will be used as a running example throughout the rest of the book.

Chapter 5 discusses testing and QA, and Chapter 6 covers some of Node's more important constructs and how they are extended and used by Express. Chapter 7 covers templating (using Handlebars), which lays the foundation of building useful websites with Express. Chapter 8 and Chapter 9 cover cookies, sessions, and form handlers, rounding out the things you need to know to build basic functional websites with Express.

Chapter 10 delves into middleware, a concept central to Express. Chapter 11 explains how to use middleware to send email from the server and discusses security and layout issues inherent to email.

Chapter 12 offers a preview into production concerns. Even though at this stage in the book you don't have all the information you need to build a production-ready website, thinking about production now can save you from major headaches in the future.

Chapter 13 is about persistence, with a focus on MongoDB (one of the leading document databases) and PostgreSQL (a popular open-source relational database management system).

Chapter 14 gets into the details of routing with Express (how URLs are mapped to content), and Chapter 15 takes a diversion into writing APIs with Express. Chapter 17 covers the details of serving static content, with a focus on maximizing performance.

Chapter 18 discusses security: how to build authentication and authorization into your app (with a focus on using a third-party authentication provider), as well as how to run your site over HTTPS.

Chapter 19 explains how to integrate with third-party services. Examples used are Twitter, Google Maps, and the US National Weather Service.

Chapter 16 takes what we've learned about Express and uses it to refactor the running example as an SPA, with Express as the backend server providing the API we created in Chapter 15.

Chapter 20 and Chapter 21 get you ready for the big day: your site launch. They cover debugging, so you can root out any defects before launch, and the process of going live. Chapter 22 talks about the next important (and oft-neglected) phase: maintenance.

The book concludes with Chapter 23, which points you to additional resources, should you want to further your education about Node and Express, and where you can go to get help.

Example Website

Starting in Chapter 3, a running example will be used throughout the book: the Meadowlark Travel website. I wrote the first edition just after getting back from a trip to Lisbon, and I had travel on my mind, so the example website I chose is for a fictional travel company in my home state of Oregon (the Western Meadowlark is the state songbird of Oregon). Meadowlark Travel allows travelers to connect to local "amateur tour guides," and it partners with companies offering bike and scooter rentals and local tours, with a focus on ecotourism.

Like any pedagogical example, the Meadowlark Travel website is contrived, but it is an example that covers many of the challenges facing real-world websites: third-party component integration, geolocation, ecommerce, performance, and security.

As the focus on this book is backend infrastructure, the example website will not be complete; it merely serves as a fictional example of a real-world website to provide depth and context to the examples. Presumably, you are working on your own website, and you can use the Meadowlark Travel example as a template for it.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
Shows commands or other text that should be typed literally by the user.

`Constant width italic`
Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/EthanRBrown/web-development-with-node-and-express-2e*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Web Development with Node and Express, Second Edition* by Ethan Brown (O'Reilly). Copyright 2019 Ethan Brown, 978-1-492-05351-4."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/web_dev_node_express_2e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

So many people in my life have played a part in making this book a reality; it would not have been possible without the influence of all the people who have touched my life and made me who I am today.

I would like to start out by thanking everyone at Pop Art: not only has my time at Pop Art given me a renewed passion for engineering, but I have learned so much from everyone there, and without their support, this book would not exist. I am grateful to Steve Rosenbaum for creating an inspiring place to work, and to Del Olds for bringing me on board, making me feel welcome, and being an honorable leader. Thanks to Paul Inman for his unwavering support and inspiring attitude toward engineering, and Tony Alferez for his warm support and for helping me carve out time for writing without impacting Pop Art. Finally, thanks to all the great engineers I have worked with, who keep me on my toes: John Skelton, Dylan Hallstrom, Greg Yung, Quinn Michaels, CJ Stritzel, Colwyn Fritze-Moor, Diana Holland, Sam Wilskey, Cory Buckley, and Damion Moyer.

I owe a great debt of gratitude to my current team at Value Management Strategies, Inc. I have learned so much about the business side of software from Robert Stewart and Greg Brink, and so much about team communication, cohesion, and effectiveness from Ashley Carson (thanks for your unwavering support, Scratch Chromatic). Terry Hays, Cheryl Kramer, and Eric Trimble, thank you all for your hard work and support! And thanks to Damon Yeutter, Tyler Brenton, and Brad Wells for their critical work on requirements analysis and project management. Most importantly, thank you to the talented and dedicated developers who have worked with me—tirelessly—at VMS: Adam Smith, Shane Ryan, Jeremy Loss, Dan Mace, Michael Meow, Julianne Soifer, Matt Nakatani, and Jake Feldmann.

Thanks to all of my bandmates at School of Rock! What a crazy journey it's been and what a joyful creative outlet to have. Special thanks to the instructors who share their passion and knowledge of music: Josh Thomas, Amanda Sloane, Dave Coniglio, Dan Lee, Derek Blackstone, and Cory West. Thank you all for giving me the opportunity to be a rock star!

Zach Mason, thank you for being an inspiration to me. This book may be no *The Lost Books of the Odyssey*, but it is *mine*, and I don't know if I would have been so bold without your example.

Elizabeth and Ezra, thank you for the gifts you both gave me. I will love you both forever.

I owe everything to my family. I couldn't have wished for a better, more loving education than the one they gave me, and I see their exceptional parenting reflected in my sister too.

Many thanks to Simon St. Laurent for giving me this opportunity, and to Angela Rufino (second edition) and Brian Anderson (first edition) for their steady and encouraging editing. Thanks to everyone at O'Reilly for their dedication and passion. Thanks to Alejandra Olvera-Novack, Chetan Karande, Brian Sletten, Tamas Piros, Jennifer Pierce, Mike Wilson, Ray Villalobos, and Eric Elliot for their thorough and constructive technical reviews.

Katy Roberts and Hanna Nelson provided invaluable feedback and advice on my "over the transom" proposal that made this book possible. Thank you both so much! Thanks to Chris Cowell-Shah for his excellent feedback on the QA chapter.

Lastly, thanks to my dear friends, without whom I surely would have gone insane: Byron Clayton, Mark Booth, Katy Roberts, and Kimberly Christensen. I love you all.

CHAPTER 1

Introducing Express

The JavaScript Revolution

Before I introduce the main subject of this book, it is important to provide a little background and historical context, and that means talking about JavaScript and Node. The age of JavaScript is truly upon us. From its humble beginnings as a client-side scripting language, not only has it become completely ubiquitous on the client side, but its use as a server-side language has finally taken off too, thanks to Node.

The promise of an all-JavaScript technology stack is clear: no more context switching! No longer do you have to switch mental gears from JavaScript to PHP, C#, Ruby, or Python (or any other server-side language). Furthermore, it empowers frontend engineers to make the jump to server-side programming. This is not to say that server-side programming is strictly about the language; there's still a lot to learn. With JavaScript, though, at least the language won't be a barrier.

This book is for all those who see the promise of the JavaScript technology stack. Perhaps you are a frontend engineer looking to extend your experience into backend development. Perhaps you're an experienced backend developer like myself who is looking to JavaScript as a viable alternative to entrenched server-side languages.

If you've been a software engineer for as long as I have, you have seen many languages, frameworks, and APIs come into vogue. Some have taken off, and some have faded into obsolescence. You probably take pride in your ability to rapidly learn new languages, new systems. Every new language you come across feels a little more familiar: you recognize a bit here from a language you learned in college, a bit there from that job you had a few years ago. It feels good to have that kind of perspective, certainly, but it's also wearying. Sometimes you want to just *get something done*, without having to learn a whole new technology or dust off skills you haven't used in months or years.

JavaScript may seem, at first, an unlikely champion. I sympathize, believe me. If you told me in 2007 that I would not only come to think of JavaScript as my language of choice, but also write a book about it, I would have told you you were crazy. I had all the usual prejudices against JavaScript: I thought it was a "toy" language, something for amateurs and dilettantes to mangle and abuse. To be fair, JavaScript did lower the bar for amateurs, and there was a lot of questionable JavaScript out there, which did not help the language's reputation. To turn a popular saying on its head, "Hate the player, not the game."

It is unfortunate that people suffer this prejudice against JavaScript; it has prevented people from discovering how powerful, flexible, and elegant the language is. Many people are just now starting to take JavaScript seriously, even though the language as we know it now has been around since 1996 (although many of its more attractive features were added in 2005).

By picking up this book, you are probably free of that prejudice: either because, like me, you have gotten past it or because you never had it in the first place. In either case, you are fortunate, and I look forward to introducing you to Express, a technology made possible by a delightful and surprising language.

In 2009, years after people had started to realize the power and expressiveness of JavaScript as a browser scripting language, Ryan Dahl saw JavaScript's potential as a server-side language, and Node.js was born. This was a fertile time for internet technology. Ruby (and Ruby on Rails) took some great ideas from academic computer science, combined them with some new ideas of its own, and showed the world a quicker way to build websites and web applications. Microsoft, in a valiant effort to become relevant in the internet age, did amazing things with .NET and learned not only from Ruby and JavaScript but also from Java's mistakes, while borrowing heavily from the halls of academia.

Today, web developers have the freedom to use the very latest JavaScript language features without fear of alienating users with older browsers, thanks to transcompilation technologies like Babel. Webpack has become the ubiquitous solution for managing dependencies in web applications and ensuring performance, and frameworks such as React, Angular, and Vue are changing the way people approach web development, relegating declarative Document Object Model (DOM) manipulation libraries (such as jQuery) to yesterday's news.

It is an exciting time to be involved in internet technology. Everywhere there are amazing new ideas (or amazing old ideas revitalized). The spirit of innovation and excitement is greater now than it has been in many years.

Introducing Express

The Express website describes Express as a "minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications." What does that really mean, though? Let's break that description down:

Minimal
: This is one of the most appealing aspects of Express. Many times, framework developers forget that usually "less is more." The Express philosophy is to provide the *minimal* layer between your brain and the server. That doesn't mean that it's not robust or that it doesn't have enough useful features. It means that it gets in your way less, allowing you full expression of your ideas, while at the same time providing something useful. Express provides you a minimal framework, and you can add in different parts of Express functionality as needed, replacing whatever doesn't meet your needs. This is a breath of fresh air. So many frameworks give you *everything*, leaving you with a bloated, mysterious, and complex project before you've even written a single line of code. Often, the first task is to waste time carving off unneeded functionality or replacing the functionality that doesn't meet requirements. Express takes the opposite approach, allowing you to add what you need when you need it.

Flexible
: At the end of the day, what Express does is very simple: it accepts HTTP requests from a client (which can be a browser, a mobile device, another server, a desktop application…anything that speaks HTTP) and returns an HTTP response. This basic pattern describes almost everything connected to the internet, making Express extremely flexible in its applications.

Web application framework
: Perhaps a more accurate description would be "server-side part of a web application framework." Today, when you think of "web application framework," you generally think of a single-page application framework like React, Angular, or Vue. However, except for a handful of standalone applications, most web applications need to share data and integrate with other services. They generally do so through a web API, which can be considered the server-side component of a web application framework. Note that it's still possible (and sometimes desirable) to build an entire application with server-side rendering only, in which case Express may very well constitute the entire web application framework!

In addition to the features of Express explicitly mentioned in its own description, I would add two of my own:

Fast
: As Express became the go-to web framework for Node.js development, it attracted a lot of attention from big companies that were running high-performance,

high-traffic websites. This created pressure on the Express team to focus on performance, and Express now offers leading performance for high-traffic websites.

Unopinionated

One of the hallmarks of the JavaScript ecosystem is its size and diversity. While Express is often at the center of Node.js web development, there are hundreds (if not thousands) of community packages that go into an Express application. The Express team recognized this ecosystem diversity and responded by providing an extremely flexible middleware system that makes it easy to use the components of your choice in creating your application. Over the course of Express's development, you can see it shedding "built-in" components in favor of configurable middleware.

I mentioned that Express is the "server-side part" of a web application framework…so we should probably consider the relationship between server-side and client-side applications.

Server-Side and Client-Side Applications

A *server-side application* is one where the pages in the application are rendered on the server (as HTML, CSS, images and other multimedia assets, and JavaScript) and sent to the client. A *client-side application*, by contrast, renders most of its own user interface from an initial application bundle that is sent only once. That is, once the browser receives the initial (generally very minimal) HTML, it uses JavaScript to modify the DOM dynamically and doesn't need to rely on the server to display new pages (though raw data usually still comes from the server).

Prior to 1999, server-side applications were the standard. As a matter of fact, the term *web application* was officially introduced that year. I think of the period roughly between 1999 and 2012 as the Web 2.0 era, during which the technologies and techniques that would eventually become client-side applications were being developed. By 2012, with smartphones firmly entrenched, it was common practice to send as little information as possible over the network, a practice that favored client-side applications.

Server-side applications are often called *server-side rendered* (SSR), and client-side applications are usually called *single-page applications* (SPAs). Client-side applications are fully realized in frameworks such as React, Angular, and Vue. I've always felt that "single-page" was a bit of a misnomer because—from the user's perspective—there can indeed be many pages. The only difference is whether the page is shipped from the server or dynamically rendered in the client.

In reality, there are many blurred lines between server-side applications and client-side applications. Many client-side applications have two to three HTML bundles that can be sent to that client (for example, the public interface and the logged-in interface, or a regular interface and an admin interface). Furthermore, SPAs are often combined with SSR to increase first-page-load performance and aid in search engine optimization (SEO).

In general, if the server sends a small number of HTML files (generally one to three), and the user experiences a rich, multiview experience based on dynamic DOM manipulation, we consider that client-side rendering. The data (usually in the form of JSON) and multimedia assets for different views generally still come from the network.

Express, of course, doesn't really care much if you're making a server-side or client-side application; it is happy to fill either role. It makes no difference to Express if you are serving one HTML bundle or a hundred.

While SPAs have definitively "won" as the predominant web application architecture, this book begins with examples consistent with server-side applications. They are still relevant, and the conceptual difference between serving one HTML bundle or many is small. There is an SPA example in Chapter 16.

A Brief History of Express

The creator of Express, TJ Holowaychuk, describes Express as a web framework inspired by Sinatra, which is a web framework based on Ruby. It is no surprise that Express borrows from a framework built on Ruby: Ruby spawned a wealth of great approaches to web development, aimed at making web development faster, more efficient, and more maintainable.

As much as Express was inspired by Sinatra, it was also deeply intertwined with Connect, a "plug-in" library for Node. Connect coined the term *middleware* to describe pluggable Node modules that can handle web requests to varying degrees. In 2014, in version 4.0, Express removed its dependency on Connect, but it still owes its concept of middleware to Connect.

Express underwent a fairly substantial rewrite between 2.x and 3.0, then again between 3.x and 4.0. This book focuses on version 4.0.

Node: A New Kind of Web Server

In a way, Node has a lot in common with other popular web servers, like Microsoft's Internet Information Services (IIS) or Apache. What is more interesting, though, is how it differs, so let's start there.

Much like Express, Node's approach to web servers is very minimal. Unlike IIS or Apache, which a person can spend many years mastering, Node is easy to set up and configure. That is not to say that tuning Node servers for maximum performance in a production setting is a trivial matter; it's just that the configuration options are simpler and more straightforward.

Another major difference between Node and more traditional web servers is that Node is single threaded. At first blush, this may seem like a step backward. As it turns out, it is a stroke of genius. Single threading vastly simplifies the business of writing web apps, and if you need the performance of a multithreaded app, you can simply spin up more instances of Node, and you will effectively have the performance benefits of multithreading. The astute reader is probably thinking this sounds like smoke and mirrors. After all, isn't multithreading through server parallelism (as opposed to app parallelism) simply moving the complexity around, not eliminating it? Perhaps, but in my experience, it has moved the complexity to exactly where it should be. Furthermore, with the growing popularity of cloud computing and treating servers as generic commodities, this approach makes a lot more sense. IIS and Apache are powerful indeed, and they are designed to squeeze the very last drop of performance out of today's powerful hardware. That comes at a cost, though: they require considerable expertise to set up and tune to achieve that performance.

In terms of the way apps are written, Node apps have more in common with PHP or Ruby apps than .NET or Java apps. While the JavaScript engine that Node uses (Google's V8) does compile JavaScript to native machine code (much like C or C++), it does so transparently,[1] so from the user's perspective, it behaves like a purely interpreted language. Not having a separate compile step reduces maintenance and deployment hassles: all you have to do is update a JavaScript file, and your changes will automatically be available.

Another compelling benefit of Node apps is that Node is incredibly platform independent. It's not the first or only platform-independent server technology, but platform independence is really more of a spectrum than a binary proposition. For example, you can run .NET apps on a Linux server thanks to Mono, but it's a painful endeavor thanks to spotty documentation and system incompatibilities. Likewise, you can run PHP apps on a Windows server, but it is not generally as easy to set up as it is on a Linux machine. Node, on the other hand, is a snap to set up on all the major operating systems (Windows, macOS, and Linux) and enables easy collaboration. Among website design teams, a mix of PCs and Macs is quite common. Certain platforms, like .NET, introduce challenges for frontend developers and designers, who often use Macs, which has a huge impact on collaboration and efficiency. The idea of

[1] Often called *just in time* (JIT) compilation.

being able to spin up a functioning server on any operating system in a matter of minutes (or even seconds!) is a dream come true.

The Node Ecosystem

Node, of course, lies at the heart of the stack. It's the software that enables JavaScript to run on the server, uncoupled from a browser, which in turn allows frameworks written in JavaScript (like Express) to be used. Another important component is the database, which will be covered in more depth in Chapter 13. All but the simplest of web apps will need a database, and there are databases that are more at home in the Node ecosystem than others.

It is unsurprising that database interfaces are available for all the major relational databases (MySQL, MariaDB, PostgreSQL, Oracle, SQL Server); it would be foolish to neglect those established behemoths. However, the advent of Node development has revitalized a new approach to database storage: the so-called NoSQL databases. It's not always helpful to define something as what it's *not*, so we'll add that these NoSQL databases might be more properly called "document databases" or "key/value pair databases." They provide a conceptually simpler approach to data storage. There are many, but MongoDB is one of the front-runners, and it's the NoSQL database we will be using in this book.

Because building a functional website depends on multiple pieces of technology, acronyms have been spawned to describe the "stack" that a website is built on. For example, the combination of Linux, Apache, MySQL, and PHP is referred to as the *LAMP* stack. Valeri Karpov, an engineer at MongoDB, coined the acronym *MEAN*: Mongo, Express, Angular, and Node. While it's certainly catchy, it is limiting: there are so many choices for databases and application frameworks that "MEAN" doesn't capture the diversity of the ecosystem (it also leaves out what I believe is an important component: rendering engines).

Coining an inclusive acronym is an interesting exercise. The indispensable component, of course, is Node. While there are other server-side JavaScript containers, Node is emerging as the dominant one. Express, also, is not the only web app framework available, though it is close to Node in its dominance. The two other components that are usually essential for web app development are a database server and a rendering engine (either a templating engine like Handlebars or an SPA framework like React). For these last two components, there aren't as many clear front-runners, and this is where I believe it's a disservice to be restrictive.

What ties all these technologies together is JavaScript, so in an effort to be inclusive, I will be referring to the *JavaScript stack*. For the purposes of this book, that means Node, Express, and MongoDB (there is also a relational database example in Chapter 13).

Licensing

When developing Node applications, you may find yourself having to pay more attention to licensing than you ever have before (I certainly have). One of the beauties of the Node ecosystem is the vast array of packages available to you. However, each of those packages carries its own licensing, and worse, each package may depend on other packages, meaning that understanding the licensing of the various parts of the app you've written can be tricky.

However, there is some good news. One of the most popular licenses for Node packages is the MIT license, which is painlessly permissive, allowing you to do *almost* anything you want, including use the package in closed source software. However, you shouldn't just assume every package you use is MIT licensed.

> There are several packages available in npm that will try to figure out the licenses of each dependency in your project. Search npm for `nlf` or `license-report`.

While MIT is the most common license you will encounter, you may also see the following licenses:

GNU General Public License (GPL)
The GPL is a popular open source license that has been cleverly crafted to keep software free. That means if you use GPL-licensed code in your project, your project must *also* be GPL licensed. Naturally, this means your project can't be closed source.

Apache 2.0
This license, like MIT, allows you to use a different license for your project, including a closed source license. You must, however, include notice of components that use the Apache 2.0 license.

Berkeley Software Distribution (BSD)
Similar to Apache, this license allows you to use whatever license you wish for your project, as long as you include notice of the BSD-licensed components.

> Software is sometimes *dual licensed* (licensed under two different licenses). A common reason for doing this is to allow the software to be used in both GPL projects and projects with more permissive licensing. (For a component to be used in GPL software, the component must be GPL licensed.) This is a licensing scheme I often employ with my own projects: dual licensing with GPL and MIT.

Lastly, if you find yourself writing your own packages, you should be a good citizen and pick a license for your package, and document it correctly. There is nothing more frustrating to a developer than using someone's package and having to dig around in the source to determine the licensing or, worse, find that it isn't licensed at all.

Conclusion

I hope this chapter has given you some more insight into what Express is and how it fits into the larger Node and JavaScript ecosystem, as well some clarity on the relationship between server-side and client-side web applications.

If you're still feeling confused about what Express actually *is*, don't worry: sometimes it's much easier to just start using something to understand what it is, and this book will get you started building web applications with Express. Before we start using Express, however, we're going to take a tour of Node in the next chapter, which is important background information to understanding how Express works.

CHAPTER 2
Getting Started with Node

If you don't have any experience with Node, this chapter is for you. Understanding Express and its usefulness requires a basic understanding of Node. If you already have experience building web apps with Node, feel free to skip this chapter. In this chapter, we will be building a very minimal web server with Node; in the next chapter, we will see how to do the same thing with Express.

Getting Node

Getting Node installed on your system couldn't be easier. The Node team has gone to great lengths to make sure the installation process is simple and straightforward on all major platforms.

Go to the Node home page. Click the big green button that has a version number followed by "LTS (Recommended for Most Users)." LTS stands for *Long-Term Support*, and is somewhat more stable than the version labeled Current, which contains more recent features and performance improvements.

For Windows and macOS, an installer will be downloaded that walks you through the process. For Linux, you will probably be up and running more quickly if you use a package manager.

If you're a Linux user and you do want to use a package manager, make sure you follow the instructions in the aforementioned web page. Many Linux distributions will install an extremely old version of Node if you don't add the appropriate package repository.

You can also download a standalone installer, which can be helpful if you are distributing Node to your organization.

Using the Terminal

I'm an unrepentant fan of the power and productivity of using a terminal (also called a *console* or *command prompt*). Throughout this book, all examples will assume you're using a terminal. If you're not friends with your terminal, I highly recommend you spend some time familiarizing yourself with your terminal of choice. Many of the utilities in this book have corresponding GUI interfaces, so if you're dead set against using a terminal, you have options, but you will have to find your own way.

If you're on macOS or Linux, you have a wealth of venerable shells (the terminal command interpreter) to choose from. The most popular by far is bash, though zsh has its adherents. The main reason I gravitate toward bash (other than long familiarity) is ubiquity. Sit down in front of any Unix-based computer, and 99% of the time, the default shell will be bash.

If you're a Windows user, things aren't quite so rosy. Microsoft has never been particularly interested in providing a pleasant terminal experience, so you'll have to do a little more work. Git helpfully includes a "Git bash" shell, which provides a Unix-like terminal experience (it has only a small subset of the normally available Unix command-line utilities, but it's a useful subset). While Git bash provides you with a minimal bash shell, it's still using the built-in Windows console application, which leads to an exercise in frustration (even simple functionality like resizing a console window, selecting text, cutting, and pasting is unintuitive and awkward). For this reason, I recommend installing a more sophisticated terminal such as ConsoleZ or ConEmu. For Windows power users—especially for .NET developers or for hardcore Windows systems or network administrators—there is another option: Microsoft's own PowerShell. PowerShell lives up to its name: people do remarkable things with it, and a skilled PowerShell user could give a Unix command-line guru a run for their money. However, if you move between macOS/Linux and Windows, I still recommend sticking with Git bash for the consistency it provides.

If you're using Windows 10 or later, you can now install Ubuntu Linux directly on Windows! This is not dual-boot or virtualization but some great work on behalf of Microsoft's open source team to bring the Linux experience to Windows. You can install Ubuntu on Windows through the Microsoft App Store.

A final option for Windows users is virtualization. With the power and architecture of modern computers, the performance of virtual machines (VMs) is practically indistinguishable from actual machines. I've had great luck with Oracle's free VirtualBox.

Finally, no matter what system you're on, there are excellent cloud-based development environments, such as Cloud9 (now an AWS product). Cloud9 will spin up a new Node development environment that makes it extremely easy to get started quickly with Node.

Once you've settled on a shell that makes you happy, I recommend you spend some time getting to know the basics. There are many wonderful tutorials on the internet (The Bash Guide is a great place to start), and you'll save yourself a lot of headaches later by learning a little now. At minimum, you should know how to navigate directories; copy, move, and delete files; and break out of a command-line program (usually Ctrl-C). If you want to become a terminal ninja, I encourage you to learn how to search for text in files, search for files and directories, chain commands together (the old "Unix philosophy"), and redirect output.

> On many Unix-like systems, Ctrl-S has a special meaning: it will "freeze" the terminal (this was once used to pause output quickly scrolling past). Since this is such a common shortcut for Save, it's easy to unthinkingly press, which leads to a confusing situation for most people (this happens to me more often than I care to admit). To unfreeze the terminal, simply hit Ctrl-Q. So if you're ever confounded by a terminal that seems to have suddenly frozen, try pressing Ctrl-Q and see if that releases it.

Editors

Few topics inspire such heated debate among programmers as the choice of editors, and for good reason: the editor is your primary tool. My editor of choice is vi (or an editor that has a vi mode).[1] vi isn't for everyone (my coworkers constantly roll their eyes at me when I tell them how easy it would be to do what they're doing in vi), but finding a powerful editor and learning to use it will significantly increase your productivity and, dare I say it, enjoyment. One of the reasons I particularly like vi (though hardly the most important reason) is that, like bash, it is ubiquitous. If you have access to a Unix system, vi is there for you. Most popular editors have a "vi mode" that allows you to use vi keyboard commands. Once you get used to it, it's hard to imagine using anything else. vi is a hard road at first, but the payoff is worth it.

If, like me, you see the value in being familiar with an editor that's available anywhere, your other option is Emacs. Emacs and I have never quite gotten on (and usually you're either an Emacs person or a vi person), but I absolutely respect the power and flexibility that Emacs provides. If vi's modal editing approach isn't for you, I would encourage you to look into Emacs.

While knowing a console editor (like vi or Emacs) can come in incredibly handy, you may still want a more modern editor. A popular choice is Visual Studio Code (not to

[1] These days, vi is essentially synonymous with vim (vi improved). On most systems, vi is aliased to vim, but I usually type `vim` to make sure I'm using vim.

be confused with Visual Studio without the "Code"). I can heartily endorse Visual Studio Code; it is a well-designed, fast, efficient editor that is perfectly suited for Node and JavaScript development. Another popular choice is Atom, which is also popular in the JavaScript community. Both of these editors are available for free on Windows, macOS, and Linux (and both have vi modes!).

Now that we have a good tool to edit code, let's turn our attention to npm, which will help us get packages that other people have written so we can take advantage of the large and active JavaScript community.

npm

npm is the ubiquitous package manager for Node packages (and is how we'll get and install Express). In the wry tradition of PHP, GNU, WINE, and others, *npm* is not an acronym (which is why it isn't capitalized); rather, it is a recursive abbreviation for "npm is not an acronym."

Broadly speaking, a package manager's two primary responsibilities are installing packages and managing dependencies. npm is a fast, capable, and painless package manager, which I feel is in large part responsible for the rapid growth and diversity of the Node ecosystem.

> There is a popular competing package manager called Yarn that uses the same package database that npm does; we'll be using Yarn in Chapter 16.

npm is installed when you install Node, so if you followed the steps listed earlier, you've already got it. So let's get to work!

The primary command you'll be using with npm (unsurprisingly) is `install`. For example, to install nodemon (a popular utility to automatically restart a Node program when you make changes to the source code), you would issue the following command (on the console):

```
npm install -g nodemon
```

The `-g` flag tells npm to install the package *globally*, meaning it's available globally on the system. This distinction will become clearer when we cover the *package.json* files. For now, the rule of thumb is that JavaScript utilities (like nodemon) will generally be installed globally, whereas packages that are specific to your web app or project will not.

Unlike languages like Python—which underwent a major language change from 2.0 to 3.0, necessitating a way to easily switch between different environments—the Node platform is new enough that it is likely that you should always be running the latest version of Node. However, if you do find yourself needing to support multiple versions of Node, check out nvm or n, which allow you to switch environments. You can find out what version of Node is installed on your computer by typing `node --version`.

A Simple Web Server with Node

If you've ever built a static HTML website before or are coming from a PHP or ASP background, you're probably used to the idea of the web server (Apache or IIS, for example) serving your static files so that a browser can view them over the network. For example, if you create the file *about.html* and put it in the proper directory, you can then navigate to *http://localhost/about.html*. Depending on your web server configuration, you might even be able to omit the *.html*, but the relationship between URL and filename is clear: the web server simply knows where the file is on the computer and serves it to the browser.

localhost, as the name implies, refers to the computer you're on. This is a common alias for the IPv4 loopback address 127.0.0.1 or the IPv6 loopback address ::1. You will often see 127.0.0.1 used instead, but I will be using *localhost* in this book. If you're using a remote computer (using SSH, for example), keep in mind that browsing to *localhost* will not connect to that computer.

Node offers a different paradigm than that of a traditional web server: the app that you write *is* the web server. Node simply provides the framework for you to build a web server.

"But I don't want to write a web server," you might be saying! It's a natural response: you want to be writing an app, not a web server. However, Node makes the business of writing this web server a simple affair (just a few lines, even), and the control you gain over your application in return is more than worth it.

So let's get to it. You've installed Node, you've made friends with the terminal, and now you're ready to go.

Hello World

I've always found it unfortunate that the canonical introductory programming example is the uninspired message "Hello world." However, it seems almost sacrilegious at this point to fly in the face of such ponderous tradition, so we'll start there and then move on to something more interesting.

In your favorite editor, create a file called *helloworld.js* (*ch02/00-helloworld.js* in the companion repo):

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' })
  res.end('Hello world!')
})

server.listen(port, () => console.log(`server started on port ${port}; ` +
  'press Ctrl-C to terminate....'))
```

Depending on when and where you learned JavaScript, you may be disconcerted by the lack of semicolons in this example. I used to be a die-hard semicolon promoter, and I grudgingly stopped using them as I did more React development, where it is conventional to omit them. After a while, the fog lifted from my eyes, and I wondered why I was ever so excited about semicolons! I'm now firmly on team "no-semicolon," and the examples in this book will reflect that. It's a personal choice, and you are welcome to use semicolons if you wish.

Make sure you are in the same directory as *helloworld.js*, and type `node helloworld.js`. Then open a browser and navigate to *http://localhost:3000*, and *voilà*! Your first web server. This particular one doesn't serve HTML; rather, it just displays the message "Hello world!" in plain text to your browser. If you want, you can experiment with sending HTML instead: just change `text/plain` to `text/html` and change `'Hello world!'` to a string containing valid HTML. I didn't demonstrate that, because I try to avoid writing HTML inside JavaScript for reasons that will be discussed in more detail in Chapter 7.

Event-Driven Programming

The core philosophy behind Node is that of *event-driven programming*. What that means for you, the programmer, is that you have to understand what events are available to you and how to respond to them. Many people are introduced to event-driven programming by implementing a user interface: the user clicks something, and you handle the *click event*. It's a good metaphor, because it's understood that the programmer has no control over when, or if, the user is going to click something, so event-driven programming is really quite intuitive. It can be a little harder to make the conceptual leap to responding to events on the server, but the principle is the same.

In the previous code example, the event is implicit: the event that's being handled is an HTTP request. The `http.createServer` method takes a function as an argument; that function will be invoked every time an HTTP request is made. Our simple program just sets the content type to plain text and sends the string "Hello world!"

Once you start thinking in terms of event-driven programming, you start seeing events everywhere. One such event is when a user navigates from one page or area of your application to another. How your application responds to that navigation event is referred to as *routing*.

Routing

Routing refers to the mechanism for serving the client the content it has asked for. For web-based client/server applications, the client specifies the desired content in the URL; specifically, the path and querystring (the parts of a URL will be discussed in more detail in Chapter 6).

> Server routing traditionally hinges on the path and the querystring, but there is other information available: headers, the domain, the IP address, and more. This allows servers to take into consideration, for example, the approximate physical location of the user or the preferred language of the user.

Let's expand our "Hello world!" example to do something more interesting. Let's serve a really minimal website consisting of a home page, an About page, and a Not Found page. For now, we'll stick with our previous example and just serve plaintext instead of HTML (*ch02/01-helloworld.js* in the companion repo):

```
const http = require('http')
const port = process.env.PORT || 3000

const server = http.createServer((req,res) => {
  // normalize url by removing querystring, optional
  // trailing slash, and making it lowercase
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('Homepage')
      break
    case '/about':
      res.writeHead(200, { 'Content-Type': 'text/plain' })
      res.end('About')
      break
    default:
      res.writeHead(404, { 'Content-Type': 'text/plain' })
      res.end('Not Found')
      break
  } })

server.listen(port, () => console.log(`server started on port ${port}; ` +
  'press Ctrl-C to terminate....'))
```

If you run this, you'll find you can now browse to the home page (*http://localhost:3000*) and the About page (*http://localhost:3000/about*). Any querystrings will be ignored (so *http://localhost:3000/?foo=bar* will serve the home page), and any other URL (*http://localhost:3000/foo*) will serve the Not Found page.

Serving Static Resources

Now that we've got some simple routing working, let's serve some real HTML and a logo image. These are called *static* resources because they generally don't change (as opposed to, for example, a stock ticker: every time you reload the page, the stock prices may change).

> Serving static resources with Node is suitable for development and small projects, but for larger projects, you will probably want to use a proxy server such as NGINX or a CDN to serve static resources. See Chapter 17 for more information.

If you've worked with Apache or IIS, you're probably used to just creating an HTML file, navigating to it, and having it delivered to the browser automatically. Node doesn't work like that: we're going to have to do the work of opening the file, reading it, and then sending its contents along to the browser. So let's create a directory in our project called *public* (why we don't call it *static* will become evident in the next chapter). In that directory, we'll create *home.html*, *about.html*, *404.html*, a subdirectory called *img*, and an image called *img/logo.png*. I'll leave that up to you; if you're reading this book, you probably know how to write an HTML file and find an image. In your HTML files, reference the logo thusly: `<img src="/img/logo.png" alt="logo">`.

Now modify *helloworld.js* (*ch02/02-helloworld.js* in the companion repo):

```
const http = require('http')
const fs = require('fs')
const port = process.env.PORT || 3000

function serveStaticFile(res, path, contentType, responseCode = 200) {
  fs.readFile(__dirname + path, (err, data) => {
    if(err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' })
      return res.end('500 - Internal Error')
    }
    res.writeHead(responseCode, { 'Content-Type': contentType })
    res.end(data)
  })
}

const server = http.createServer((req,res) => {
  // normalize url by removing querystring, optional trailing slash, and
  // making lowercase
  const path = req.url.replace(/\/?(?:\?.*)?$/, '').toLowerCase()
  switch(path) {
    case '':
      serveStaticFile(res, '/public/home.html', 'text/html')
      break
    case '/about':
      serveStaticFile(res, '/public/about.html', 'text/html')
      break
    case '/img/logo.png':
      serveStaticFile(res, '/public/img/logo.png', 'image/png')
      break
    default:
      serveStaticFile(res, '/public/404.html', 'text/html', 404)
      break
  }
})

server.listen(port, () => console.log(`server started on port ${port}; ` +
  'press Ctrl-C to terminate....'))
```

In this example, we're being pretty unimaginative with our routing. If you navigate to *http://localhost:3000/about*, the *public/about.html* file is served. You could change the route to be anything you want, and change the file to be anything you want. For example, if you had a different About page for each day of the week, you could have files *public/about_mon.html*, *public/about_tue.html*, and so on, and provide logic in your routing to serve the appropriate page when the user navigates to *http://localhost:3000/about*.

Note we've created a helper function, `serveStaticFile`, that's doing the bulk of the work. `fs.readFile` is an asynchronous method for reading files. There is a synchronous version of that function, `fs.readFileSync`, but the sooner you start thinking asynchronously, the better. The `fs.readFile` function uses a pattern called *callbacks*. You provide a function called a *callback function*, and when the work has been done, that callback function is invoked ("called back," so to speak). In this case, `fs.readFile` reads the contents of the specified file and executes the callback function when the file has been read; if the file didn't exist or there were permissions issues reading the file, the `err` variable is set, and the function returns an HTTP status code of 500 indicating a server error. If the file is read successfully, the file is sent to the client with the specified response code and content type. Response codes will be discussed in more detail in Chapter 6.

`__dirname` will resolve to the directory the executing script resides in. So if your script resides in */home/sites/app.js*, `__dirname` will resolve to */home/sites*. It's a good idea to use this handy global whenever possible. Failing to do so can cause hard-to-diagnose errors if you run your app from a different directory.

Onward to Express

So far, Node probably doesn't seem that impressive to you. We've basically replicated what Apache or IIS do for you automatically, but now you have some insight into how Node does things and how much control you have. We haven't done anything particularly impressive, but you can see how we could use this as a jumping-off point to do more sophisticated things. If we continued down this road, writing more and more sophisticated Node applications, you might very well end up with something that resembles Express….

Fortunately, we don't have to: Express already exists, and it saves you from implementing a lot of time-consuming infrastructure. So now that we've gotten a little Node experience under our belt, we're ready to jump into learning Express.

CHAPTER 3

Saving Time with Express

In Chapter 2, you learned how to create a simple web server using only Node. In this chapter, we will re-create that server using Express. This will provide a jumping-off point for the rest of the content of this book and introduce you to the basics of Express.

Scaffolding

Scaffolding is not a new idea, but many people (myself included) were introduced to the concept by Ruby. The idea is simple: most projects require a certain amount of so-called *boilerplate* code, and who wants to re-create that code every time you begin a new project? A simple way is to create a rough skeleton of a project, and every time you need a new project, you just copy this skeleton, or template.

Ruby on Rails took this concept one step further by providing a program that would automatically generate scaffolding for you. The advantage of this approach is that it could generate a more sophisticated framework than just selecting from a collection of templates.

Express has taken a page from Ruby on Rails and provided a utility to generate scaffolding to start your Express project.

While the Express scaffolding utility is useful, I think it's valuable to learn how to set up Express from scratch. In addition to learning more, you have more control over what gets installed and the structure of your project. Also, the Express scaffolding utility is geared toward server-side HTML generation and is less relevant for APIs and single-page applications.

While we won't be using the scaffolding utility, I encourage you to take a look at it once you've finished the book: by then you'll be armed with everything you need to know to evaluate whether the scaffolding it generates is useful for you. For more information, see the `express-generator` documentation.

The Meadowlark Travel Website

Throughout this book, we'll be using a running example: a fictional website for Meadowlark Travel, a company offering services for people visiting the great state of Oregon. If you're more interested in creating an API, have no fear: the Meadowlark Travel website will expose an API in addition to serving a functional website.

Initial Steps

Start by creating a new directory: this will be the root directory for your project. In this book, whenever we refer to the project directory, app directory, or project root, we're referring to this directory.

> You'll probably want to keep your web app files separate from all the other files that usually accompany a project, such as meeting notes, documentation, etc. For that reason, I recommend making your project root a subdirectory of your project directory. For example, for the Meadowlark Travel website, I might keep the project in *~/projects/meadowlark*, and the project root in *~/projects/meadowlark/site*.

npm manages project dependencies—as well as metadata about the project—in a file called *package.json*. The easiest way to create this file is to run `npm init`: it will ask you a series of questions and generate a *package.json* file to get you started (for the "entry point" question, use *meadowlark.js* for the name of your project).

> Every time you run npm, you may get warnings about a missing description or repository field. It's safe to ignore these warnings, but if you want to eliminate them, edit the *package.json* file and provide values for the fields npm is complaining about. For more information about the fields in this file, see the npm *package.json* documentation.

The first step will be installing Express. Run the following npm command:

```
npm install express
```

Running `npm install` will install the named package(s) in the *node_modules* directory and update the *package.json* file. Since the *node_modules* directory can be regenerated at any time with npm, we will not save it in our repository. To ensure we don't accidentally add it to our repository, we create a file called *.gitignore*:

```
# ignore packages installed by npm
node_modules

# put any other files you don't want to check in here, such as .DS_Store
# (OSX), *.bak, etc.
```

Now create a file called *meadowlark.js*. This will be our project's entry point. Throughout the book, we will simply be referring to this file as the *app file* (*ch03/00-meadowlark.js* in the companion repo):

```
const express = require('express')

const app = express()

const port = process.env.PORT || 3000

// custom 404 page
app.use((req, res) => {
  res.type('text/plain')
  res.status(404)
  res.send('404 - Not Found')
})

// custom 500 page
app.use((err, req, res, next) => {
  console.error(err.message)
  res.type('text/plain')
  res.status(500)
  res.send('500 - Server Error')
})

app.listen(port, () => console.log(
  `Express started on http://localhost:${port}; ` +
  `press Ctrl-C to terminate.`))
```

Many tutorials, as well as the Express scaffolding generator, encourage you to name your primary file *app.js* (or sometimes *index.js* or *server.js*). Unless you're using a hosting service or deployment system that requires your main application file to have a specific name, I don't feel there's a compelling reason to do this, and I prefer to name the primary file after the project. Anyone who's ever stared at a bunch of editor tabs that all say "index.html" will immediately see the wisdom of this. `npm init` will default to *index.js*; if you use a different name for your application file, make sure to update the `main` property in *package.json*.

You now have a minimal Express server. You can start the server (`node meadowlark.js`) and navigate to *http://localhost:3000*. The result will be disappointing: you haven't provided Express with any routes, so it will simply give you a generic 404 message indicating that the page doesn't exist.

> Note how we choose the port that we want our application to run on: `const port = process.env.PORT || 3000`. This allows us to override the port by setting an environment variable before you start the server. If your app isn't running on port 3000 when you run this example, check to see whether your `PORT` environment variable is set.

Let's add some routes for the home page and an About page. Before the 404 handler, we'll add two new routes (*ch03/01-meadowlark.js* in the companion repo):

```
app.get('/', (req, res) => {
  res.type('text/plain')
  res.send('Meadowlark Travel');
})

app.get('/about', (req, res) => {
  res.type('text/plain')
  res.send('About Meadowlark Travel')
})

// custom 404 page
app.use((req, res) => {
  res.type('text/plain')
  res.status(404)
  res.send('404 - Not Found')
})
```

`app.get` is the method by which we're adding routes. In the Express documentation, you will see `app.METHOD`. This doesn't mean that there's literally a method called `METHOD`; it's just a placeholder for your (lowercased) HTTP verbs (`get` and `post` being the most common). This method takes two parameters: a path and a function.

The *path* is what defines the route. Note that `app.METHOD` does the heavy lifting for you: by default, it doesn't care about the case or trailing slash, and it doesn't consider the querystring when performing the match. So the route for the About page will work for */about*, */About*, */about/*, */about?foo=bar*, */about/?foo=bar*, etc.

The *function* you provide will get invoked when the route is matched. The parameters passed to that function are the request and response objects, which we'll learn more about in Chapter 6. For now, we're just returning plain text with a status code of 200 (Express defaults to a status code of 200—you don't have to specify it explicitly).

I highly recommend getting a browser plug-in that shows you the status code of the HTTP request as well as any redirects that took place. It will make it easier to spot redirect issues in your code or incorrect status codes, which are often overlooked. For Chrome, Ayima's Redirect Path works wonderfully. In most browsers, you can see the status code in the Network section of the developer tools.

Instead of using Node's low-level `res.end`, we're switching to using Express's extension, `res.send`. We are also replacing Node's `res.writeHead` with `res.set` and `res.status`. Express is also providing us a convenience method, `res.type`, which sets the `Content-Type` header. While it's still possible to use `res.writeHead` and `res.end`, it isn't necessary or recommended.

Note that our custom 404 and 500 pages must be handled slightly differently. Instead of using `app.get`, we are using `app.use`. `app.use` is the method by which Express adds *middleware*. We'll be covering middleware in more depth in Chapter 10, but for now, you can think of this as a catchall handler for anything that didn't get matched by a route. This brings us to an important point: *in Express, the order in which routes and middleware are added is significant*. If we put the 404 handler above the routes, the home page and About page would stop working; instead, those URLs would result in a 404. Right now, our routes are pretty simple, but they also support wildcards, which can lead to problems with ordering. For example, what if we wanted to add subpages to About, such as */about/contact* and */about/directions*? The following will not work as expected:

```
app.get('/about*', (req,res) => {
  // send content....
}) app.get('/about/contact', (req,res) => {
  // send content....
}) app.get('/about/directions', (req,res) => {
  // send content....
})
```

In this example, the `/about/contact` and `/about/directions` handlers will never be matched because the first handler uses a wildcard in its path: `/about*`.

Express can distinguish between the 404 and 500 handlers by the number of arguments their callback functions take. Error routes will be covered in depth in Chapter 10 and Chapter 12.

Now you can start the server again and see that there's a functioning home page and About page.

So far, we haven't done anything that couldn't be done just as easily without Express, but already Express is providing us some functionality that isn't immediately obvious. Remember in the previous chapter how we had to normalize `req.url` to determine what resource was being requested? We had to manually strip off the querystring and the trailing slash and convert to lowercase. Express's router is now handling those details for us automatically. While it may not seem like a large thing now, it's only scratching the surface of what Express's router is capable of.

Views and Layouts

If you're familiar with the "model-view-controller" paradigm, then the concept of a view will be no stranger to you. Essentially, a *view* is what gets delivered to the user. In the case of a website, that usually means HTML, though you could also deliver a PNG or a PDF or anything that can be rendered by the client. For our purposes, we will consider views to be HTML.

A view differs from a static resource (like an image or CSS file) in that a view doesn't necessarily have to be static: the HTML can be constructed on the fly to provide a customized page for each request.

Express supports many different view engines that provide different levels of abstraction. Express gives some preference to a view engine called *Pug* (which is no surprise, because it is also the brainchild of TJ Holowaychuk). The approach Pug takes is minimal: what you write doesn't resemble HTML at all, which certainly represents a lot less typing (no more angle brackets or closing tags). The Pug engine then takes that and converts it to HTML.

> Pug was originally called Jade, and the name changed with the release of version 2 because of a trademark issue.

Pug is appealing, but that level of abstraction comes at a cost. If you're a frontend developer, you have to understand HTML and understand it well, even if you're actually writing your views in Pug. Most frontend developers I know are uncomfortable with the idea of their primary markup language being abstracted away. For this reason, I am recommending the use of another, less abstract templating framework called *Handlebars*.

Handlebars (which is based on the popular language-independent templating language Mustache) doesn't attempt to abstract away HTML for you: you write HTML with special tags that allow Handlebars to inject content.

In the years following the original release of this book, React has taken the world by storm…which abstracts HTML away from frontend developers! Viewed through that lens, my prediction that frontend developers didn't want HTML abstracted away hasn't stood the test of time. However, JSX (the JavaScript language extension that most React developers use) is (almost) identical to writing HTML, so I wasn't entirely wrong.

To provide Handlebars support, we'll use Eric Ferraiuolo's `express-handlebars` package. In your project directory, execute the following:

```
npm install express-handlebars
```

Then in *meadowlark.js*, modify the first few lines (*ch03/02-meadowlark.js* in the companion repo):

```
const express = require('express')
const expressHandlebars = require('express-handlebars')

const app = express()

// configure Handlebars view engine
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
}))
app.set('view engine', 'handlebars')
```

This creates a view engine and configures Express to use it by default. Now create a directory called *views* that has a subdirectory called *layouts*. If you're an experienced web developer, you're probably already comfortable with the concepts of *layouts* (sometimes called *master pages*). When you build a website, there's a certain amount of HTML that's the same—or very close to the same—on every page. It not only becomes tedious to rewrite all that repetitive code for every page, but also creates a potential maintenance nightmare: if you want to change something on every page, you have to change *all* the files. Layouts free you from this, providing a common framework for all the pages on your site.

So let's create a template for our site. Create a file called *views/layouts/main.handlebars*:

```
<!doctype html>
<html>
  <head>
    <title>Meadowlark Travel</title>
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

The only thing that you probably haven't seen before is this: `{{{body}}}`. This expression will be replaced with the HTML for each view. When we created the Handlebars instance, note we specified the default layout (`defaultLayout: \'main'`). That means that unless you specify otherwise, this is the layout that will be used for any view.

Now let's create view pages for our home page, *views/home.handlebars*:

```
<h1>Welcome to Meadowlark Travel</h1>
```

Then our About page, *views/about.handlebars*:

```
<h1>About Meadowlark Travel</h1>
```

Then our Not Found page, *views/404.handlebars*:

```
<h1>404 - Not Found</h1>
```

And finally our Server Error page, *views/500.handlebars*:

```
<h1>500 - Server Error</h1>
```

> You probably want your editor to associate *.handlebars* and *.hbs* (another common extension for Handlebars files) with HTML to enable syntax highlighting and other editor features. For vim, you can add the line `au BufNewFile,BufRead *.handlebars set filetype=html` to your *~/.vimrc* file. For other editors, consult your documentation.

Now that we have some views set up, we have to replace our old routes with new ones that use these views (*ch03/02-meadowlark.js* in the companion repo):

```
app.get('/', (req, res) => res.render('home'))

app.get('/about', (req, res) => res.render('about'))

// custom 404 page
app.use((req, res) => {
  res.status(404)
  res.render('404')
})

// custom 500 page
app.use((err, req, res, next) => {
  console.error(err.message)
  res.status(500)
  res.render('500')
})
```

Note that we no longer have to specify the content type or status code: the view engine will return a content type of `text/html` and a status code of 200 by default. In the catchall handler, which provides our custom 404 page, and the 500 handler, we have to set the status code explicitly.

If you start your server and check out the home or About page, you'll see that the views have been rendered. If you examine the source, you'll see that the boilerplate HTML from *views/layouts/main.handlebars* is there.

Even though every time you visit the home page, you get the same HTML, these routes are considered *dynamic content*, because we could make a different decision each time the route gets called (which we'll see plenty of later in this book). However, content that really never changes, in other words, static content, is common and important, so we'll consider static content next.

Static Files and Views

Express relies on *middleware* to handle static files and views. Middleware is a concept that will be covered in more detail in Chapter 10. For now, it's sufficient to know that middleware provides modularization, making it easier to handle requests.

The `static` middleware allows you to designate one or more directories as containing static resources that are simply to be delivered to the client without any special handling. This is where you would put things such as images, CSS files, and client-side JavaScript files.

In your project directory, create a subdirectory called *public* (we call it *public* because anything in this directory will be served to the client without question). Then, before you declare any routes, you'll add the `static` middleware (*ch03/02-meadowlark.js* in the companion repo):

```
app.use(express.static(__dirname + '/public'))
```

The `static` middleware has the same effect as creating a route for each static file you want to deliver that renders a file and returns it to the client. So let's create an *img* subdirectory inside *public* and put our *logo.png* file in there.

Now we can simply reference */img/logo.png* (note, we do not specify `public`; that directory is invisible to the client), and the `static` middleware will serve that file, setting the content type appropriately. Now let's modify our layout so that our logo appears on every page:

```
<body>
  <header>
    <img src="/img/logo.png" alt="Meadowlark Travel Logo">
  </header>
  {{{body}}}
</body>
```

> Remember that middleware is processed in order, and static middleware—which is usually declared first or at least very early—will override other routes. For example, if you put an *index.html* file in the *public* directory (try it!), you'll find that the contents of that file get served instead of the route you configured! So if you're getting confusing results, check your static files and make sure there's nothing unexpected matching the route.

Dynamic Content in Views

Views aren't simply a complicated way to deliver static HTML (though they can certainly do that as well). The real power of views is that they can contain dynamic information.

Let's say that on the About page, we want to deliver a "virtual fortune cookie." In our *meadowlark.js* file, we define an array of fortune cookies:

```
const fortunes = [
  "Conquer your fears or they will conquer you.",
  "Rivers need springs.",
  "Do not fear what you don't know.",
  "You will have a pleasant surprise.",
  "Whenever possible, keep it simple.",
]
```

Modify the view (*/views/about.handlebars*) to display a fortune:

```
<h1>About Meadowlark Travel</h1>
{{#if fortune}}
  <p>Your fortune for the day:</p>
  <blockquote>{{fortune}}</blockquote>
{{/if}}
```

Now modify the route */about* to deliver the random fortune cookie:

```
app.get('/about', (req, res) => {
  const randomFortune = fortunes[Math.floor(Math.random()*fortunes.length)]
  res.render('about', { fortune: randomFortune })
})
```

Now if you restart the server and load the */about* page, you'll see a random fortune, and you'll get a new one every time you reload the page. Templating is incredibly useful, and we will be covering it in depth in Chapter 7.

Conclusion

We've created a basic website with Express. Even though it's simple, it contains all the seeds we need for a full-featured website. In the next chapter, we'll be crossing our *t*s and dotting our *i*s in preparation for adding more advanced functionality.

CHAPTER 4
Tidying Up

In the previous two chapters, we were just experimenting: dipping our toes into the waters, so to speak. Before we proceed to more complex functionality, we're going to do some housekeeping and build some good habits into our work.

In this chapter, we'll start our Meadowlark Travel project in earnest. Before we start building the website itself, though, we're going to make sure we have the tools we need to produce a high-quality product.

> The running example in this book is not necessarily one you have to follow. If you're anxious to build your own website, you could follow the framework of the running example but modify it accordingly so that by the time you finish this book, you could have a finished website!

File and Directory Structure

Structuring applications has spawned many a religious debate, and there's no one right way to do it. However, there are some common conventions that are helpful to know about.

It's typical to try to restrict the number of files in your project root. Typically, you'll find configuration files (like *package.json*), a *README.md* file, and a bunch of directories. Most source code goes under a directory often called *src*. For the sake of brevity, we won't be using that convention in this book (nor does the Express scaffolding application do this, surprisingly). For real-world projects, you'll probably eventually find that your project root gets cluttered if you're putting source code there, and you'll want to collect those files under a directory like *src*.

I've also mentioned that I prefer to name my main application file (sometimes called the *entry point*) after the project itself (*meadowlark.js*) as opposed to something generic like *index.js*, *app.js*, or *server.js*.

It's largely up to you how to structure your application, and I recommend providing a road map to your structure in the *README.md* file (or a readme linked from it).

At minimum, I recommend you always have the following two files in your project root: *package.json* and *README.md*. The rest is up to your imagination.

Best Practices

The phrase *best practices* is one you hear thrown around a lot these days, and it means that you should "do things right" and not cut corners (we'll talk about what this means specifically in a moment). No doubt you've heard the engineering adage that your options are "fast," "cheap," and "good," and you can pick any two. The thing that's always bothered me about this model is that it doesn't take into account the *accrual value* of doing things correctly. The first time you do something correctly, it may take five times as long to do it as it would have to do it quick and dirty. The second time, though, it's going to take only three times as long. By the time you've done it correctly a dozen times, you'll be doing it almost as fast as the quick and dirty way.

I had a fencing coach who would always remind us that practice doesn't make perfect; practice makes *permanent*. That is, if you do something over and over again, eventually it will become automatic, rote. That is true, but it says nothing about the quality of the thing you are practicing. If you practice bad habits, then bad habits become rote. Instead, you should follow the rule that *perfect* practice makes perfect. In that spirit, I encourage you to follow the rest of the examples in this book as if you were making a real-live website, as if your reputation and remuneration were depending on the quality of the outcome. Use this book to not only learn new skills but to practice building good habits.

The practices we will be focusing on are version control and QA. In this chapter, we'll be discussing version control, and we'll discuss QA in the next chapter.

Version Control

I hope I don't have to convince you of the value of version control (if I did, that might take a whole book itself). Broadly speaking, version control offers these benefits:

Documentation
: Being able to go back through the history of a project to see the decisions that were made and the order in which components were developed can be valuable documentation. Having a technical history of your project can be quite useful.

Attribution

If you work on a team, attribution can be hugely important. Whenever you find something in code that is opaque or questionable, knowing who made that change can save you many hours. It could be that the comments associated with the change are sufficient to answer your questions, and if not, you'll know who to talk to.

Experimentation

A good version control system enables experimentation. You can go off on a tangent, trying something new, without fear of affecting the stability of your project. If the experiment is successful, you can fold it back into the project, and if it is not successful, you can abandon it.

Years ago, I made the switch to distributed version control systems (DVCSs). I narrowed my choices down to Git and Mercurial and went with Git, because of its ubiquity and flexibility. Both are excellent and free version control systems, and I recommend you use one of them. In this book, we will be using Git, but you are welcome to substitute Mercurial (or another version control system altogether).

If you are unfamiliar with Git, I recommend Jon Loeliger's excellent *Version Control with Git* (O'Reilly). Also, GitHub has a good listing of Git learning resources.

How to Use Git with This Book

First, make sure you have Git. Type `git --version`. If it doesn't respond with a version number, you'll need to install Git. See the Git documentation for installation instructions.

There are two ways to follow along with the examples in this book. One is to type out the examples yourself and follow along with the Git commands. The other is to clone the companion repository I am using for all of the examples and check out the associated files for each example. Some people learn better by typing out examples, while some prefer to just see and run the changes without having to type it all in.

If You're Following Along by Doing It Yourself

We already have a very rough framework for our project: some views, a layout, a logo, a main application file, and a *package.json* file. Let's go ahead and create a Git repository and add all those files.

First, we go to the project directory and initialize a Git repository there:

```
git init
```

Now before we add all the files, we'll create a *.gitignore* file to help prevent us from accidentally adding things we don't want to add. Create a text file called *.gitignore* in your project directory in which you can add any files or directories you want Git to ignore by default (one per line). It also supports wildcards. For example, if your editor creates backup files with a tilde at the end (like *meadowlark.js~*), you might put `*~` in the *.gitignore* file. If you're on a Mac, you'll want to put `.DS_Store` in there. You'll also want to put `node_modules` in there (for reasons that will be discussed soon). So for now, the file might look like this:

```
node_modules
*~
.DS_Store
```

> Entries in the *.gitignore* file also apply to subdirectories. So if you put `*~` in the *.gitignore* in the project root, all such backup files will be ignored even if they are in subdirectories.

Now we can add all of our existing files. There are many ways to do this in Git. I generally favor `git add -A`, which is the most sweeping of all the variants. If you are new to Git, I recommend you either add files one by one (`git add meadowlark.js`, for example) if you want to commit only one or two files, or add all of your changes (including any files you might have deleted) using `git add -A`. Since we want to add all the work we've already done, we'll use the following:

```
git add -A
```

> Newcomers to Git are commonly confused by the `git add` command; it adds *changes*, not files. So if you've modified *meadowlark.js*, and then you type `git add meadowlark.js` ,what you're really doing is adding the changes you've made.

Git has a "staging area," where changes go when you run `git add`. So the changes we've added haven't actually been committed yet, but they're ready to go. To commit the changes, use `git commit`:

```
git commit -m "Initial commit."
```

The `-m "Initial commit."` allows you to write a message associated with this commit. Git won't even let you make a commit without a message, and for good reason. Always strive to make meaningful commit messages; they should briefly but concisely describe the work you've done.

If You're Following Along by Using the Official Repository

To get the official repository for this book, run `git clone`:

```
git clone https://github.com/EthanRBrown/web-development-with-node-and-express-2e
```

This repository has a directory for each chapter that contains code samples. For example, the source code for this chapter can be found in the `ch04` directory. The code samples in each chapter are generally numbered for ease of reference. Throughout the repository, I have liberally added *README.md* files containing additional notes about the samples.

> In the first version of this book, I took a different approach with the repository, with a linear history as if you were developing an increasingly sophisticated project. While this approach pleasantly mirrored the way a project in the real world might develop, it caused a lot of headache, both for me and for my readers. As npm packages changed, the code samples would change, and short of rewriting the entire history of the repo, there was no good way to update the repository or note the changes in the text. While the chapter-per-directory approach is more artificial, it allows the text to be synced more closely with the repository and also enables easier community contribution.

As this book is updated and improved, the repository will also be updated, and when it is, I will add a version tag so you can check out a version of the repository that corresponds to the version of the book you're reading now. The current version of the repository is 2.0.0. I am roughly following *semantic versioning* principles here (more on this later in this chapter); the PATCH increment (the last number) represents minor changes that shouldn't impact your ability to follow along with the book. That is, if the repo is at version 2.0.15, that should still correspond with this version of the book. However, if the MINOR increment (the second number) is different (2.1.0), that indicates that the content in the companion repo may have diverged from what you're reading, and you may want to check out a tag starting with 2.0.

The companion repo liberally makes use of *README.md* files to add additional explanation to the code samples.

> If at any point you want to experiment, keep in mind that the tag you have checked out puts you in what Git calls a "detached HEAD" state. While you are free to edit any files, it is unsafe to commit anything you do without creating a branch first. So if you do want to base an experimental branch off of a tag, simply create a new branch and check it out, which you can do with one command: `git checkout -b experiment` (where `experiment` is the name of your branch; you can use whatever you want). Then you can safely edit and commit on that branch as much as you want.

npm Packages

The npm packages that your project relies on reside in a directory called *node_modules*. (It's unfortunate that this is called *node_modules* and not *npm_packages*, as Node modules are a related but different concept.) Feel free to explore that directory to satisfy your curiosity or to debug your program, but you should never modify any code in this directory. In addition to that being bad practice, all of your changes could easily be undone by npm.

If you need to make a modification to a package your project depends on, the correct course of action would be to create your own fork of the package. If you do go this route and feel that your improvements would be useful to others, congratulations: you're now involved in an open source project! You can submit your changes, and if they meet the project standards, they'll be included in the official package. Contributing to existing packages and creating customized builds is beyond the scope of this book, but there is a vibrant community of developers out there to help you if you want to contribute to existing packages.

Two of the main purposes of the *package.json* file are to describe your project and to list its dependencies. Go ahead and look at your *package.json* file now. You should see something like this (the exact version numbers will probably be different, as these packages get updated often):

```
{
  "dependencies": {
    "express": "^4.16.4",
    "express-handlebars": "^3.0.0"
  }
}
```

Right now, our *package.json* file contains only information about dependencies. The caret (^) in front of the package versions indicates that any version that starts with the specified version number—up to the next major version number—will work. For example, this *package.json* indicates that any version of Express that starts with 4.0.0 will work, so 4.0.1 and 4.9.9 would both work, but 3.4.7 would not, nor would 5.0.0. This is the default version specificity when you use `npm install`, and is generally a pretty safe bet. The consequence of this approach is that if you want to move up to a newer version, you will have to edit the file to specify the new version. Generally, that's a good thing because it prevents changes in dependencies from breaking your project without your knowing about it. Version numbers in npm are parsed by a component called *semver* (for "semantic versioning"). If you want more information about versioning in npm, consult the Semantic Versioning Specification and this article by Tamas Piros.

The Semantic Versioning Specification states that software using semantic versioning must declare a "public API." I've always found this wording to be confusing; what they really mean is "someone must care about interfacing with your software." If you consider this in the broadest sense, it could really be construed to mean anything. So don't get hung up on that part of the specification; the important details are in the format.

Since the *package.json* file lists all the dependencies, the *node_modules* directory is really a derived artifact. That is, if you were to delete it, all you would have to do to get the project working again would be to run `npm install`, which will re-create the directory and put all the necessary dependencies in it. It is for this reason that I recommend putting `node_modules` in your *.gitignore* file and not including it in source control. However, some people feel that your repository should contain everything necessary to run the project and prefer to keep `node_modules` in source control. I find that this is "noise" in the repository, and I prefer to omit it.

As of version of 5 of npm, an additional file, *package-lock.json*, will be created. Whereas *package.json* can be "loose" in its specification of dependency versions (with the `^` and `~` version modifiers), *package-lock.json* records the *exact* versions that were installed, which can be helpful if you need to re-create the exact dependency versions in your project. I recommend you check this file into source control and don't modify it by hand. See the *package-lock.json* documentation for more information.

Project Metadata

The other purpose of the *package.json* file is to store project metadata, such as the name of the project, authors, license information, and so on. If you use `npm init` to initially create your *package.json* file, it will populate the file with the necessary fields for you, and you can update them at any time. If you intend to make your project available on npm or GitHub, this metadata becomes critical. If you would like more information about the fields in *package.json*, see the *package.json* documentation. The other important piece of metadata is the *README.md* file. This file can be a handy place to describe the overall architecture of the website, as well as any critical information that someone new to the project might need. It is in a text-based wiki format called Markdown. Refer to the Markdown documentation for more information.

Node Modules

As mentioned earlier, Node modules and npm packages are related but different concepts. *Node modules*, as the name implies, offer a mechanism for modularization and encapsulation. *npm packages* provide a standardized scheme for storing, versioning, and referencing projects (which are not restricted to modules). For example, we import Express itself as a module in our main application file:

```
const express = require('express')
```

`require` is a Node function for importing a module. By default, Node looks for modules in the directory *node_modules* (it should be no surprise, then, that there's an *express* directory inside of *node_modules*). However, Node also provides a mechanism for creating your own modules (you should never create your own modules in the *node_modules* directory). In addition to modules installed into *node_modules* via a package manager, there are more than 30 "core modules" provided by Node, such as `fs`, `http`, `os`, and `path`. To see the whole list, see this illuminating Stack Overflow question and refer to the official Node documentation.

Let's see how we can modularize the fortune cookie functionality we implemented in the previous chapter.

First let's create a directory to store our modules. You can call it whatever you want, but *lib* (short for "library") is a common choice. In that folder, create a file called *fortune.js* (*ch04/lib/fortune.js* in the companion repo):

```
const fortuneCookies = [
  "Conquer your fears or they will conquer you.",
  "Rivers need springs.",
  "Do not fear what you don't know.",
  "You will have a pleasant surprise.",
  "Whenever possible, keep it simple.",
]

exports.getFortune = () => {
  const idx = Math.floor(Math.random()*fortuneCookies.length)
  return fortuneCookies[idx]
}
```

The important thing to note here is the use of the global variable `exports`. If you want something to be visible outside of the module, you have to add it to `exports`. In this example, the function `getFortune` will be available from outside this module, but our array `fortuneCookies` will be *completely hidden*. This is a good thing: encapsulation allows for less error-prone and fragile code.

There are several ways to export functionality from a module. We will be covering different methods throughout the book and summarizing them in Chapter 22.

Now in *meadowlark.js*, we can remove the `fortuneCookies` array (though there would be no harm in leaving it; it can't conflict in any way with the array of the same name defined in *lib/fortune.js*). It is traditional (but not required) to specify imports at the top of the file, so at the top of the *meadowlark.js* file, add the following line (*ch04/meadowlark.js* in the companion repo):

```
const fortune = require('./lib/fortune')
```

Note that we prefix our module name with `./`. This signals to Node that it should not look for the module in the *node_modules* directory; if we omitted that prefix, this would fail.

Now in our route for the About page, we can utilize the `getFortune` method from our module:

```
app.get('/about', (req, res) => {
  res.render('about', { fortune: fortune.getFortune() } )
})
```

If you're following along, let's commit those changes:

```
git add -A git commit -m "Moved 'fortune cookie' into module."
```

You will find modules to be a powerful and easy way to encapsulate functionality, which will improve the overall design and maintainability of your project, as well as make testing easier. Refer to the official Node module documentation for more information.

> Node modules are sometimes called *CommonJS (CJS) modules*, in reference to an older specification that Node took inspiration from. The JavaScript language is adopting an official packaging mechanism, called ECMAScript Modules (ESM). If you've been writing JavaScript in React or another progressive frontend language, you may already be familiar with ESM, which uses `import` and `export` (instead of `exports`, `module.exports`, and `require`). For more information, see Dr. Axel Rauschmayer's blog post "ECMAScript 6 modules: the final syntax".

Conclusion

Now that we're armed with some more information about Git, npm, and modules, we're ready to discuss how we can produce a better product by employing good quality assurance (QA) practices in our coding.

I encourage you to keep in mind the following lessons from this chapter:

- Version control makes the software development process safer and more predictable, and I encourage you to use it even for small projects; it builds good habits!

- Modularization is an important technique for managing the complexity of software. In addition to providing a rich ecosystem of modules others have developed through npm, you can package your own code in modules to better organize your project.
- Node modules (also called CJS) use a different syntax than ECMAScript modules (ESM), and you may have to switch between the two syntaxes when you go between frontend and backend code. It's a good idea to be familiar with both.

CHAPTER 5
Quality Assurance

Quality assurance is a phrase that is prone to send shivers down the spines of developers—which is unfortunate. After all, don't you want to make quality software? Of course you do. So it's not the end goal that's the sticking point; it's the politics of the matter. I've found that two common situations arise in web development:

Large or well-funded organizations
There's usually a QA department, and, unfortunately, an adversarial relationship springs up between QA and development. This is the worst thing that can happen. Both departments are playing on the same team, for the same goal, but QA often defines success as finding more bugs, while development defines success as generating fewer bugs, and that serves as the basis for conflict and competition.

Small organizations and organizations on a budget
Often, there is no QA department; the development staff is expected to serve the dual role of establishing QA and developing software. This is not a ridiculous stretch of the imagination or a conflict of interest. However, QA is a very different discipline than development, and it attracts different personalities and talents. This is not an impossible situation, and certainly there are developers out there who have the QA mind-set, but when deadlines loom, it's usually QA that gets the short shrift, to the project's detriment.

With most real-world endeavors, multiple skills are required, and increasingly, it's harder to be an expert in all of those skills. However, some competency in the areas for which you are not directly responsible will make you more valuable to the team and make the team function more effectively. A developer acquiring QA skills offers a great example: these two disciplines are so tightly intertwined that cross-disciplinary understanding is extremely valuable.

It is also common to shift activities traditionally done by QA to development, making developers responsible for QA. In this paradigm, software engineers who specialize in QA act almost as consultants to developers, helping them build QA into their development workflow. Whether QA roles are divided or integrated, it is clear that understanding QA is beneficial to developers.

This book is not for QA professionals; it is aimed at developers. So my goal is not to make you a QA expert but to give you some experience in that area. If your organization has a dedicated QA staff, it will make it easier for you to communicate and collaborate with them. If you do not, it will give you a starting point to establishing a comprehensive QA plan for your project.

In this chapter, you'll learn the following:

- Quality fundamentals and effective habits
- The types of tests (unit and integration)
- How to write unit tests with Jest
- How to write integration tests with Puppeteer
- How to configure ESLint to help prevent common errors
- What continuous integration is and where to start learning about it

The QA Plan

Development is, by and large, a creative process: envisioning something and then translating it into reality. QA, in contrast, lives more in the realm of validation and order. As such, a large part of QA is simply a matter of *knowing what needs to be done* and *making sure it gets done*. It is a discipline well-suited for checklists, procedures, and documentation. I would go so far as to say the primary activity of QA is not the testing of software itself but the *creation of a comprehensive, repeatable QA plan*.

I recommend the creation of a QA plan for every project, no matter how big or small (yes, even your weekend "fun" project!). The QA plan doesn't have to be big or elaborate; you can put it in a text file or a word processing document or a wiki. The objective of the QA plan is to record all of the steps you'll take to ensure that your product is functioning as intended.

In whatever form it takes, the QA plan is a living document. You will update it in response to the following:

- New features
- Changes in existing features
- Removed features

- Changes in testing technologies or techniques
- Defects that were missed by the QA plan

That last point deserves special mention. No matter how robust your QA is, defects will happen. And when they do, you should ask yourself, “How could we have prevented this?” When you answer that question, you can modify your QA plan accordingly to prevent future instances of this type of defect.

By now you might be getting a feel for the not insignificant effort involved in QA, and you might be reasonably wondering how much effort you want to put into it.

QA: Is It Worth It?

QA can be expensive—sometimes *very* expensive. So is it worth it? It’s a complicated formula with complicated inputs. Most organizations operate on some kind of “return on investment” model. If you spend money, you must expect to receive at least as much money in return (preferably more). With QA, though, the relationship can be muddy. A well-established and well-regarded product, for example, may be able to get by with quality issues for longer than a new and unknown project. Obviously, no one *wants* to produce a low-quality product, but the pressures in technology are high. Time-to-market can be critical, and sometimes it’s better to come to market with something that’s less than perfect than to come to market with the perfect product months later.

In web development, quality can be broken down into four dimensions:

Reach

Reach refers to the market penetration of your product: the number of people viewing your website or using your service. There’s a direct correlation between reach and profitability: the more people who visit the website, the more people who buy the product or service. From a development perspective, search engine optimization (SEO) will have the biggest impact on reach, which is why we will be including SEO in our QA plan.

Functionality

Once people are visiting your site or using your service, the quality of your site’s functionality will have a large impact on user retention; a site that works as advertised is more likely to drive return visits than one that isn’t. Functionality offers the most opportunity for test automation.

Usability

Where functionality is concerned with functional correctness, usability evaluates human-computer interaction (HCI). The fundamental question is, “Is the functionality delivered in a way that is useful to the target audience?” This often translates to “Is it easy to use?” though the pursuit of ease can often oppose flexibility or power; what seems easy to a programmer might be different from what seems easy to a nontechnical consumer. In other words, you must consider your target audience when assessing usability. Since a fundamental input to a usability measurement is a user, usability is not usually something that can be automated. However, user testing should be included in your QA plan.

Aesthetics

Aesthetics is the most subjective of the four dimensions and is therefore the least relevant to development. While there are few development concerns when it comes to your site's aesthetics, routine reviews of your site's aesthetics should be part of your QA plan. Show your site to a representative sample audience, and find out if it feels dated or does not invoke the desired response. Keep in mind that aesthetics is time sensitive (aesthetic standards shift over time) and audience specific (what appeals to one audience may be completely uninteresting to another).

While all four dimensions should be addressed in your QA plan, functionality testing and SEO can be tested automatically during development, so that will be the focus of this chapter.

Logic Versus Presentation

Broadly speaking, in your website, there are two "realms": *logic* (often called *business logic*, a term I eschew because of its bias toward commercial endeavor) and *presentation*. You can think of your website's logic existing in kind of a pure intellectual domain. For example, in our Meadowlark Travel scenario, there might be a rule that a customer must possess a valid driver's license before renting a scooter. This is a simple data-based rule: for every scooter reservation, the user needs a valid driver's license. The *presentation* of this is disconnected. Perhaps it's just a checkbox on the final form of the order page, or perhaps the customer has to provide a valid driver's license number, which is validated by Meadowlark Travel. It's an important distinction, because things should be as clear and simple as possible in the logic domain, whereas the presentation can be as complicated or as simple as it needs to be. The presentation is also subject to usability and aesthetic concerns, whereas the business domain is not.

Whenever possible, you should seek a clear delineation between your logic and presentation. There are many ways to do that, and in this book, we will be focusing on encapsulating logic in JavaScript modules. Presentation, on the other hand, will be a combination of HTML, CSS, multimedia, JavaScript, and frontend frameworks like React, Vue, or Angular.

The Types of Tests

The type of testing we will be considering in this book falls into two broad categories: unit testing and integration testing (I am considering system testing to be a type of integration testing). Unit testing is very fine-grained, testing single components to make sure they function properly, whereas integration testing tests the interaction between multiple components or even the whole system.

In general, unit testing is more useful and appropriate for logic testing. Integration testing is useful in both realms.

Overview of QA Techniques

In this book, we will be using the following techniques and software to accomplish thorough testing:

Unit tests
: Unit tests cover the smallest units of functionality in your application, usually a single function. They are almost always written by developers, not QA (though QA should be empowered to assess the quality and coverage of unit tests). In this book, we'll be using Jest for unit tests.

Integration tests
: Integration tests cover larger units of functionality, usually involving multiple parts of your application (functions, modules, subsystems, etc.). Since we are building web applications, the "ultimate" integration test is to render the application in a browser, manipulate that browser, and verify that the application behaves as expected. These tests are typically more complicated to set up and maintain, and since the focus of this book isn't QA, we'll have only one simple example of this, using Puppeteer and Jest.

Linting
: Linting isn't about finding errors but *potential* errors. The general concept of linting is that it identifies areas that could represent possible errors, or fragile constructs that could lead to errors in the future. We will be using ESLint for linting.

Let's start with Jest, our test framework (which will run both unit and integration tests).

Installing and Configuring Jest

I struggled somewhat to decide which testing framework to use in this book. Jest began its life as a framework to test React applications (and it is still the obvious choice for that), but Jest is not React-specific and is an excellent general-purpose testing framework. It's certainly not the only one: Mocha, Jasmine, Ava, and Tape are also excellent choices.

In the end, I chose Jest because I feel it offers the best overall experience (an opinion backed by Jest's excellent scores in the State of JavaScript 2018 survey). That said, there are a lot of similarities among the testing frameworks mentioned here, so you should be able to take what you learn and apply it to your favorite test framework.

To install Jest, run the following from your project root:

```
npm install --save-dev jest
```

(Note that we use `--save-dev` here; this tells npm that this is a development dependency and is not needed for the application itself to function; it will be listed in the `devDependencies` section of the *package.json* file instead of the `dependencies` section.)

Before we move on, we need a way to run Jest (which will run any tests in our project). The conventional way to do that is to add a script to *package.json*. Edit *package.json* (*ch05/package.json* in the companion repo), and modify the `scripts` property (or add it if it doesn't exist):

```
  "scripts": {
    "test": "jest"
  },
```

Now you can run all the tests in your project simply by typing that:

```
npm test
```

If you try that now, you'll probably get an error that there aren't any tests configured…because we haven't added any yet. So let's write some unit tests!

> Normally, if you add a script to your *package.json* file, you would run it with `npm run`. For example, if you added a script `foo`, you would type `npm run foo` to run it. The `test` script is so common, however, that npm knows to run it if you simply type `npm test`.

Unit Testing

Now we'll turn our attention to unit testing. Since the focus of unit testing is on isolating a single function or component, we'll first need to learn about mocking, an important technique for achieving that isolation.

Mocking

One of the challenges you'll frequently face is how to write code that is "testable." In general, code that tries to do too much or assumes a lot of dependencies is harder to test than focused code that assumes few or no dependencies.

Whenever you have a dependency, you have something that needs to be *mocked* (simulated) for effective testing. For example, our primary dependency is Express, which is already thoroughly tested, so we don't need or want to test Express itself, just *how we use it*. The only way we can determine if we're using Express correctly is to simulate Express itself.

The routes we currently have (the home page, About page, 404 page, and 500 page) are pretty difficult to test because they assume three dependencies on Express: they assume we have an Express app (so we can have `app.get`), as well as request and response objects. Fortunately, it's pretty easy to eliminate the dependence on the Express app itself (the request and response objects are harder…more on that later). Fortunately, we're not using very much functionality from the response object (we're using only the `render` method), so it will be easy to mock it, which we will see shortly.

Refactoring the Application for Testability

We don't really have a lot of code in our application to test yet. To date, we've currently added only a handful of route handlers and the `getFortune` function.

To make our app more testable, we're going to *extract* the actual route handlers to their own library. Create a file *lib/handlers.js* (*ch05/lib/handlers.js* in the companion repo):

```
const fortune = require('./fortune')

exports.home = (req, res) => res.render('home')

exports.about = (req, res) =>
  res.render('about', { fortune: fortune.getFortune() })

exports.notFound = (req, res) => res.render('404')

exports.serverError = (err, req, res, next) => res.render('500')
```

Now we can rewrite our *meadowloark.js* application file to use these handlers (*ch05/meadowlark.js* in the companion repo):

```
// typically at the top of the file
const handlers = require('./lib/handlers')

app.get('/', handlers.home)

app.get('/about', handlers.about)
```

```
// custom 404 page
app.use(handlers.notFound)

// custom 500 page
app.use(handlers.serverError)
```

It's easier now to test those handlers: they are just functions that take request and response objects, and we need to verify that we're using those objects correctly.

Writing Our First Test

There are multiple ways to identify tests to Jest. The two most common are to put tests in subdirectories named *__test__* (two underscores before and after *test*) and to name files with the extension *.test.js*. I personally like to combine the two techniques because they both serve a purpose in my mind. Putting tests in *__test__* directories keeps my test from cluttering up my source directories (otherwise, everything will look doubled in your source directory…you'll have a *foo.test.js* for every file *foo.js*), and having the *.test.js* extension means that if I'm looking at a bunch of tabs in my editor, I can see at a glance what is a test and what is source code.

So let's create a file called *lib/__tests__/handlers.test.js* (*ch05/lib/__tests__/handlers.test.js* in the companion repo):

```
const handlers = require('../handlers')

test('home page renders', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.home(req, res)
  expect(res.render.mock.calls[0][0]).toBe('home')
})
```

If you're new to testing, this will probably look pretty weird, so let's break it down.

First, we import the code we're trying to test (in this case, the route handlers). Then each test has a description; we're trying to describe what's being tested. In this case, we want to make sure that the home page gets rendered.

To invoke our render, we need request and response objects. We'd be writing code all week if we wanted to simulate the whole request and response objects, but fortunately we don't actually need much from them. We know that we don't need anything at all from the request object in this case (so we just use an empty object), and the only thing we need from the response object is a render method. Note how we construct the render function: we just call a Jest method called *jest.fn()*. This creates a generic mock function that keeps track of how it's called.

Finally, we get to the important part of the test: assertions. We've gone to all the trouble to invoke the code we're testing, but how do we assert that it did what it should?

In this case, what the code should do is call the `render` method of the response object with the string `home`. Jest's mock function keeps track of all the times it got called, so all we have to do is verify it got called exactly once (it would probably be a problem if it got called twice), which is what the first `expect` does, and that it gets called with `home` as its first argument (the first array index specifies which invocation, and the second one specifies which argument).

> It can get tedious to constantly be rerunning your tests every time you make a change to your code. Fortunately, most test frameworks have a "watch" mode that constantly monitors your code and tests for changes and reruns them automatically. To run your tests in watch mode, type `npm test -- --watch` (the extra double-dash is necessary to let npm know to pass the `--watch` argument to Jest).

Go ahead and change your `home` handler to render something other than the home view; you'll notice that your test has now failed, and you caught a bug!

We can now add tests for our other routes:

```
test('about page renders with fortune', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.about(req, res)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('about')
  expect(res.render.mock.calls[0][1])
    .toEqual(expect.objectContaining({
      fortune: expect.stringMatching(/\W/),
    }))
})

test('404 handler renders', () => {
  const req = {}
  const res = { render: jest.fn() }
  handlers.notFound(req, res)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('404')
})

test('500 handler renders', () => {
  const err = new Error('some error')
  const req = {}
  const res = { render: jest.fn() }
  const next = jest.fn()
  handlers.serverError(err, req, res, next)
  expect(res.render.mock.calls.length).toBe(1)
  expect(res.render.mock.calls[0][0]).toBe('500')
})
```

Note some extra functionality in the "about" and server error tests. The "about" render function gets called with a fortune, so we've added an expectation that it will get a fortune that is a string that contains at least one character. It's beyond the scope of this book to describe all of the functionality that is available to you through Jest and its `expect` method, but you can find comprehensive documentation on the Jest home page. Note that the server error handler takes four arguments, not two, so we have to provide additional mocks.

Test Maintenance

You might be realizing that tests are not a "set it and forget it" affair. For example, if we renamed our "home" view for legitimate reasons, our test would fail, and then we would have to fix the test in addition to fixing the code.

For this reason, teams put a lot of effort into setting realistic expectations about what should be tests and how specific the tests should be. For example, we didn't have to check to see if the "about" handler was being called with a fortune…which would save us from having to fix the test if we ditch that feature.

Furthermore, I can't offer much advice about how thoroughly you should test your code. I would expect you to have very different standards for testing code for avionics or medical equipment than for testing the code behind a marketing website.

What I can offer you is a way to answer the question, "How much of my code is tested?" The answer to that is called *code coverage*, which we'll discuss next.

Code Coverage

Code coverage offers a quantitative answer to how much of your code is tested, but like most topics in programming, there are no simple answers.

Jest helpfully provides some automated code coverage analysis. To see how much of your code is tested, run the following:

```
npm test -- --coverage
```

If you've been following along, you should see a bunch of reassuringly green "100%" coverage numbers for the files in *lib*. Jest will report on the coverage percentage of statements (Stmts), branches, functions (Funcs), and lines.

Statements are referring to JavaScript statements, such as every expression, control flow statement, etc. Note that you could have 100% line coverage but not 100% statement coverage because you can put multiple statements on a single line in JavaScript. Branch coverage refers to control flow statements, such as `if-else`. If you have an `if-else` statement and your test exercises only the `if` part, you will have 50% branch coverage for that statement.

You may note that *meadowlark.js* does not have 100% coverage. This is not necessarily a problem; if you look at our refactored *meadowlark.js* file, you'll see that most of what's in there now is simply configuration…we're just gluing things together. We're configuring Express with the relevant middleware and starting the server. Not only would it be hard to meaningfully test this code, but it's a reasonable argument that you shouldn't have to since it's merely assembling well-tested code.

You could even make the argument that the tests we've written so far are not particularly useful; they're also just verifying that we're configuring Express correctly.

Once again, I have no easy answers. At the end of the day, the type of application you're building, your level of experience, and the size and configuration of your team will have a large impact on how far down the test rabbit hole you go. I encourage you to err on the side of *too much* testing than *not enough*, but with experience, you'll find the "just right" sweet spot.

Testing Entropic Functionality

Testing *entropic* functionality (functionality that is random) comes with its own challenges. Another test we could add for our fortune cookie generator would be a test to make sure that it returns a *random* fortune cookie. But how do you know if something is random? One approach is to get a large number of fortunes—a thousand, for example—and then measure the distribution of the responses. If the function is properly random, no one response will stand out. The downside of this approach is that it's nondeterministic: it's possible (but unlikely) to get one fortune 10 times more frequently than any other fortune. If that happened, the test could fail (depending on how aggressive you set the threshold of what is "random"), but that might not actually indicate that the system being tested is failing; it's just a consequence of testing entropic systems. In the case of our fortune generator, it would be reasonable to generate 50 fortunes and expect at least three different ones. On the other hand, if we were developing a random source for a scientific simulation or security component, we would probably want to have much more detailed tests. The point is that testing entropic functionality is difficult and requires more thought.

Integration Testing

There's currently nothing interesting to test in our application; we just have a couple of pages and there's no interaction. So before we write an integration test, let's add some functionality that we can test. In the interest of keeping things simple, we'll let that functionality be a link that allows you to get from the home page to the About page. It doesn't get much simpler than that! And yet, as simple as that would appear to a user, it is a true integration test because it's exercising not only two Express route handlers, but also the HTML and the DOM interaction (the user clicking the link and the resulting page navigation). Let's add a link to *views/home.handlebars*:

```
<p>Questions?  Checkout out our
<a href="/about" data-test-id="about">About Us</a> page!</p>
```

You might be wondering about the `data-test-id` attribute. To make testing, we need some way to identify the link so we can (virtually) click it. We could have used a CSS class for this, but I prefer to reserve classes for styling and use data attributes for automation. We also could have searched for the text *About Us*, but that would be a fragile and expensive DOM search. We also could have queried against the `href` parameter, which would make sense (but then it would be harder to make this test fail, which we want to do for educational purposes).

We can go ahead and run our application and verify with our clumsy human hands that the functionality works as intended before we move on to something more automated.

Before we jump into installing Puppeteer and writing an integration test, we need to modify our application so that it can be required as a module (right now it is designed only to be run directly). The way to do that in Node is a little opaque: at the bottom of *meadowlark.js*, replace the call to `app.listen` with the following:

```
if(require.main === module) {
  app.listen(port, () => {
    console.log( `Express started on http://localhost:${port}` +
      '; press Ctrl-C to terminate.' )
  })
} else {
  module.exports = app
}
```

I'll skip the technical explanation for this as it's rather tedious, but if you're curious, a careful reading of Node's module documentation will make it clear. What's important to know is that if you run a JavaScript file directly with node, `require.main` will equal the global `module`; otherwise, it's being imported from another module.

Now that we've got that out of the way, we can install Puppeteer. Puppeteer is essentially a controllable, headless version of Chrome. (*Headless* simply means that the browser is capable of running without actually rendering a UI on-screen.) To install Puppeteer:

```
npm install --save-dev puppeteer
```

We'll also install a small utility to find an open port so that we don't get a lot of test errors because our app can't start on the port we requested:

```
npm install --save-dev portfinder
```

Now we can write an integration that does the following:

1. Starts our application server on an unoccupied port
2. Launches a headless Chrome browser and opens a page
3. Navigates to our application's home page
4. Finds a link with `data-test-id="about"` and clicks it
5. Waits for the navigation to happen
6. Verifies that we are on the */about* page

Create a directory called *integration-tests* (you're welcome to call it something else if you like) and a file in that directory called *basic-navigation.test.js* (*ch05/integration-tests/basic-navigation.test.js* in the companion repo):

```
const portfinder = require('portfinder')
const puppeteer = require('puppeteer')

const app = require('../meadowlark.js')

let server = null
let port = null

beforeEach(async () => {
  port = await portfinder.getPortPromise()
  server = app.listen(port)
})

afterEach(() => {
  server.close()
})

test('home page links to about page', async () => {
  const browser = await puppeteer.launch()
  const page = await browser.newPage()
  await page.goto(`http://localhost:${port}`)
  await Promise.all([
    page.waitForNavigation(),
    page.click('[data-test-id="about"]'),
  ])
  expect(page.url()).toBe(`http://localhost:${port}/about`)
  await browser.close()
})
```

We are using Jest's `beforeEach` and `afterEach` hooks to start our server before each test and stop it after each test (right now we have only one test, so this will really be meaningful when we add more tests). We could instead use `beforeAll` and `afterAll` so we're not starting and tearing down our server for every test, which may speed up your tests, but at the cost of not having a "clean" environment for each test. That is, if one of your tests makes changes that affect the outcome of future tests, you're introducing hard-to-maintain dependencies.

Our actual test uses Puppeteer's API, which gives us a lot of DOM query functionality. Note that almost everything here is asynchronous, and we're using `await` liberally to make the test easier to read and write (almost all of the Puppeteer API calls return a promise).[1] We wrap the navigation and the click together in a call to `Promise.all` to prevent race conditions per the Puppeteer documentation.

There's far more functionality in the Puppeteer API than I could hope to cover in this book. Fortunately, it has excellent documentation.

Testing is a vital backstop in ensuring the quality of your product, but it's not the only tool at your disposal. Linting helps you prevent common errors in the first place.

Linting

A good linter is like having a second set of eyes: it will spot things that will slide right past our human brains. The original JavaScript linter is Douglas Crockford's JSLint. In 2011, Anton Kovalyov forked JSLint, and JSHint was born. Kovalyov found that JSLint was becoming too opinionated, and he wanted to create a more customizable, community-developed JavaScript linter. After JSHint came Nicholas Zakas' ESLint, which has become the most popular choice (it won by a landslide in the 2017 State of JavaScript survey). In addition to its ubiquity, ESLint appears to be the most actively maintained linter, and I prefer its flexible configuration over JSHint, and it is what I am recommending.

ESLint can be installed on a per project basis or globally. To avoid inadvertently breaking things, I try to avoid global installations (for example, if I install ESLint globally and update it frequently, old projects may no longer lint successfully because of breaking changes, and now I have to do the extra work of updating my project).

To install ESLint in your project:

```
npm install --save-dev eslint
```

ESLint requires a configuration file to tell it which rules to apply. Doing this from scratch would be a time-consuming task, so fortunately ESLint provides a utility for creating one for you. From your project root, run the following:

```
./node_modules/.bin/eslint --init
```

[1] If you are unfamiliar with `await`, I recommend this article by Tamas Piros.

If we installed ESLint globally, we could just use `eslint --init`. The awkward `./node_modules/.bin` path is required to directly run locally installed utilities. We'll see soon that we don't have to do that if we add utilities to the `scripts` section of our *package.json* file, which is recommended for things we do frequently. However, creating an ESLint configuration is something we have to do only once per project.

ESLint will ask you some questions. For most of them, it's safe to choose the defaults, but a couple deserve note:

What type of modules does your project use?

Since we're using Node (as opposed to code that will run in the browser), you'll want to choose "CommonJS (require/exports)." You may have client-side JavaScript in your project too, in which case you may want a separate lint configuration. The easiest way to do this is to have two separate projects, but it is possible to have multiple ESLint configurations in the same project. Consult the ESLint documentation for more information.

Which framework does your project use?

Unless you see Express on there (I don't at the time of this writing), choose "None of these."

Where does your code run?

Choose Node.

Now that ESLint is set up, we need a convenient way of running it. Add the following to the `scripts` section of your *package.json*:

```
"lint": "eslint meadowlark.js lib"
```

Note that we have to explicitly tell ESLint what files and directories we want to lint. This is an argument for collecting all of your source under one directory (usually *src*).

Now brace yourself and run the following:

```
npm run lint
```

You'll probably see a lot of unpleasant-looking errors—that's usually what happens when you first run ESLint. However, if you've been following along with the Jest test, there will be some spurious errors related to Jest, which look like this:

```
 3:1   error  'test' is not defined    no-undef
 5:25  error  'jest' is not defined    no-undef
 7:3   error  'expect' is not defined  no-undef
 8:3   error  'expect' is not defined  no-undef
11:1   error  'test' is not defined    no-undef
13:25  error  'jest' is not defined    no-undef
15:3   error  'expect' is not defined  no-undef
```

ESLint (quite sensibly) doesn't appreciate unrecognized global variables. Jest injects global variables (notably `test`, `describe`, `jest`, and `expect`). Fortunately, this is an easy problem to fix. In your project root, open the *.eslintrc.js* file (this is the ESLint configuration). In the `env` section, add the following:

```
"jest": true,
```

Now if you run `npm run lint` again, you should see a lot fewer errors.

So what to do about the remaining errors? Here's where I can offer wisdom but no specific guidance. Broadly speaking, a linting error has one of three causes:

- It's a legitimate problem, and you should fix it. It may not always be obvious, in which case you may need to refer to the ESLint documentation for the particular error.
- It's a rule you don't agree with, and you can simply disable it. Many of the rules in ESLint are a matter of opinion. I'll demonstrate disabling a rule in a moment.
- You agree with the rule, but there's an instance where it's infeasible or costly to fix in some specific circumstance. For those situations, you can disable rules for only specific lines in a file, which we'll also see an example of.

If you've been following along, you should currently see the following errors:

```
/Users/ethan/wdne2e-companion/ch05/meadowlark.js
  27:5  error  Unexpected console statement  no-console

/Users/ethan/wdne2e-companion/ch05/lib/handlers.js
  10:39  error  'next' is defined but never used  no-unused-vars
```

ESLint complains about console logging because it's not necessarily a good way to provide output for your application; it can be noisy and inconsistent, and, depending on how you run it, the output can get swept under the rug. However, for our use, let's say it doesn't bother us and we want to disable that rule. Open your *.eslintrc* file, find the `rules` section (if there isn't a `rules` section, create one at the top level of the exported object), and add the following rule:

```
  "rules": {
    "no-console": "off",
  },
```

Now if we run `npm run lint` again, we'll see that error is no more! The next one is a little trickier….

Open *lib/handlers.js* and consider the line in question:

```
exports.serverError = (err, req, res, next) => res.render('500')
```

ESLint is correct; we're providing `next` as an argument but not doing anything with it (we're also not doing anything with `err` and `req`, but because of the way JavaScript treats function arguments, we have to put *something* there so we can get at `res`, which we *are* using).

You may be tempted to just remove the `next` argument. “What’s the harm?” you may think. And indeed, there would be no runtime errors, and your linter would be happy…but a hard-to-see harm would be done: your custom error handler would stop working! (If you want to see for yourself, throw an exception from one of your routes and try visiting it, and then remove the `next` argument from the `serverError` handler.)

Express is doing something subtle here: it’s using the number of actual arguments you pass to it to recognize that it’s supposed to be an error handler. Without that `next` argument—whether you use it or not—Express no longer recognizes it as an error handler.

> What the Express team has done with the error handler is undeniably “clever,” but clever code can often be confusing, easy to break, or inscrutable. As much as I love Express, this is one choice I think the team got wrong: I think it should have found a less idiosyncratic and more explicit way to specify an error handler.

We can’t change our handler code, and we need our error handler, but we like this rule and don’t want to disable it. We could just live with the error, but the errors will accumulate and be a constant irritation, and they will eventually corrode the very point of having a linter. Fortunately, we can fix it by disabling that rule for that single line. Edit *lib/handlers.js* and add the following around your error handler:

```
// Express recognizes the error handler by way of its four
// arguments, so we have to disable ESLint's no-unused-vars rule
/* eslint-disable no-unused-vars */
exports.serverError = (err, req, res, next) => res.render('500')
/* eslint-enable no-unused-vars */
```

Linting can be a little frustrating at first—it may feel like it’s constantly tripping you up. And certainly you should feel free to disable rules that don’t suit you. Eventually, you will find it less and less frustrating as you learn to avoid the common mistakes that linting is designed to catch.

Testing and linting are undeniably useful, but any tool is worthless if you never use it! It may seem crazy that you would go to the time and trouble to write unit tests and set up linting, but I’ve seen it happen, especially when the pressure is on. Fortunately, there is a way to ensure that these helpful tools don’t get forgotten: continuous integration.

Continuous Integration

I'll leave you with another extremely useful QA concept: continuous integration (CI). It's especially important if you're working on a team, but even if you're working on your own, it can provide some helpful discipline.

Basically, CI runs some or all of your tests every time you contribute code to a source code repository (you can control which branches this applies to). If all of the tests pass, nothing usually happens (you may get an email saying "good job," depending on how your CI is configured).

If, on the other hand, there are failures, the consequences are usually more…public. Again, it depends on how you configure your CI, but usually the entire team gets an email saying that you "broke the build." If your integration master is really sadistic, sometimes your boss is also on that email list! I've even known teams that set up lights and sirens when someone broke the build, and in one particularly creative office, a tiny robotic foam missile launcher fired soft projectiles at the offending developer! It's a powerful incentive to run your QA toolchain before committing.

It's beyond the scope of this book to cover installing and configuring a CI server, but a chapter on QA wouldn't be complete without mentioning it.

Currently, the most popular CI server for Node projects is Travis CI. Travis CI is a hosted solution, which can be appealing (it saves you from having to set up your own CI server). If you're using GitHub, it offers excellent integration support. CircleCI is another option.

If you're working on a project on your own, you may not get much benefit from a CI server, but if you're working on a team or an open source project, I highly recommend looking into setting up CI for your project.

Conclusion

This chapter covered a lot of ground, but I consider these essential real-world skills in any development framework. The JavaScript ecosystem is dizzyingly large, and if you're new to it, it can be hard to know where to start. I hope this chapter pointed you in the right direction.

Now that we have some experience with these tools, we'll turn our attention to some fundamentals of the Node and Express objects that bracket everything that happens in an Express application: the request and response objects.

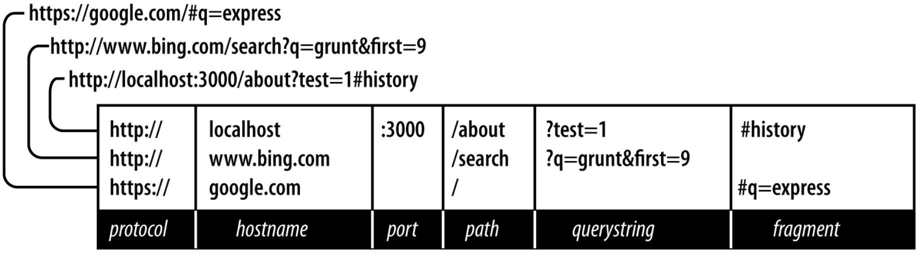
CHAPTER 6
The Request and Response Objects

In this chapter, we'll learn the important details of the request and response objects—which are very much the beginning and end of everything that happens in an Express application. When you're building a web server with Express, most of what you'll be doing starts with a request object and ends with a response object.

These two objects originate in Node and are extended by Express. Before we delve into what these objects offer us, let's establish a little background on how a client (a browser, usually) requests a page from a server and how that page is returned.

The Parts of a URL

We see URLs all the time, but we don't often stop to think about their component parts. Let's consider three URLs and examine their component parts:



https://google.com/#q=express
http://www.bing.com/search?q=grunt&first=9
http://localhost:3000/about?test=1#history

protocol	hostname	port	path	querystring	fragment
http://	localhost	:3000	/about	?test=1	#history
http://	www.bing.com		/search	?q=grunt&first=9	
https://	google.com		/		#q=express

Protocol
: The protocol determines how the request will be transmitted. We will be dealing exclusively with *http* and *https*. Other common protocols include *file* and *ftp*.

Host

The host identifies the server. Servers on your computer (localhost) or a local network may be identified simply be one word or by a numeric IP address. On the internet, the host will end in a top-level domain (TLD) like *.com* or *.net*. Additionally, there may be *subdomains*, which prefix the hostname. *www* is a common subdomain, though it can be anything. Subdomains are optional.

Port

Each server has a collection of numbered ports. Some port numbers are special, like 80 and 443. If you omit the port, port 80 is assumed for HTTP and 443 for HTTPS. In general, if you aren't using port 80 or 443, you should use a port number greater than 1023.[1] It's common to use easy-to-remember port numbers like 3000, 8080, and 8088. Only one server can be associated with a given port, and even though there are plenty of numbers to choose from, you may have to change the port number if you're using a commonly used port number.

Path

The path is generally the first part of the URL that your app cares about (it is possible to make decisions based on protocol, host, and port, but it's not good practice). The path should be used to uniquely identify pages or other resources in your app.

Querystring

The querystring is an optional collection of name/value pairs. The querystring starts with a question mark (*?*), and name/value pairs are separated by ampersands (*&*). Both names and values should be *URL encoded*. JavaScript provides a built-in function to do that: `encodeURIComponent`. For example, spaces will be replaced with plus signs (+). Other special characters will be replaced with numeric character references. Sometimes the querystring will be referred to as the *search string* or simply the *search*.

Fragment

The fragment (or *hash*) is not passed to the server at all; it is strictly for use by the browser. Some single-page applications use the fragment to control application navigation. Originally, the fragment's sole purpose was to cause the browser to display a specific part of the document, marked by an anchor tag (for example: `<a id="chapter06">`).

[1] Ports 0–1023 are "well-known ports" reserved for common services.

HTTP Request Methods

The HTTP protocol defines a collection of *request methods* (often referred to as *HTTP verbs*) that a client uses to communicate with a server. Far and away, the most common methods are GET and POST.

When you type a URL into a browser (or click a link), the browser issues an HTTP GET request to the server. The important information passed to the server is the URL path and querystring. The combination of method, path, and querystring is what your app uses to determine how to respond.

For a website, most of your pages will respond to GET requests. POST requests are usually reserved for sending information back to the server (form processing, for example). It's quite common for POST requests to respond with the same HTML as the corresponding GET request after the server has processed any information included in the request (like a form). Browsers will primarily use the GET and POST methods when communicating with your server. The Ajax requests your application makes, however, may use any HTTP verb. For example, there's an HTTP method called DELETE that is useful for, well, an API call that deletes things.

With Node and Express, you are fully in charge of what methods you respond to. In Express, you'll usually be writing handlers for specific methods.

Request Headers

The URL isn't the only thing that's passed to the server when you navigate to a page. Your browser is sending a lot of "invisible" information every time you visit a website. I'm not talking about spooky personal information (though if your browser is infected by malware, that can happen). The browser will tell the server what language it prefers to receive the page in (for example, if you download Chrome in Spain, it will request the Spanish version of pages you visit, if they exist). It will also send information about the *user agent* (the browser, operating system, and hardware) and other bits of information. All this information is sent as a request header, which is made available to you through the request object's headers property. If you're curious to see the information your browser is sending, you can create a simple Express route to display that information (*ch06/00-echo-headers.js* in the companion repo):

```
app.get('/headers', (req, res) => {
  res.type('text/plain')
  const headers = Object.entries(req.headers)
    .map(([key, value]) => `${key}: ${value}`)
  res.send(headers.join('\n'))
})
```

Response Headers

Just as your browser sends hidden information to the server in the form of request headers, when the server responds, it also sends information back that is not necessarily rendered or displayed by the browser. The information typically included in response headers is metadata and server information. We've already seen the `Content-Type` header, which tells the browser what kind of content is being transmitted (HTML, an image, CSS, JavaScript, etc.). Note that the browser will respect the `Content-Type` header regardless of what the URL path is. So you could serve HTML from a path of */image.jpg* or an image from a path of */text.html*. (There's no legitimate reason to do this; it's just important to understand that paths are abstract, and the browser uses `Content-Type` to determine how to render content.) In addition to `Content-Type`, headers can indicate whether the response is compressed and what kind of encoding it's using. Response headers can also contain hints for the browser about how long it can cache the resource. This is an important consideration for optimizing your website, and we'll be discussing that in detail in Chapter 17.

It is also common for response headers to contain some information about the server, indicating what type of server it is and sometimes even details about the operating system. The downside about returning server information is that it gives hackers a starting point to compromise your site. Extremely security-conscious servers often omit this information or even provide false information. Disabling Express's default `X-Powered-By` header is easy (*ch06/01-disable-x-powered-by.js* in the companion repo):

```
app.disable('x-powered-by')
```

If you want to see the response headers, they can be found in your browser's developer tools. To see the response headers in Chrome, for example:

1. Open the JavaScript console.
2. Click the Network tab.
3. Reload the page.
4. Pick the HTML from the list of requests (it will be the first one).
5. Click the Headers tab; you will see all response headers.

Internet Media Types

The `Content-Type` header is critically important; without it, the client would have to painfully guess how to render the content. The format of the `Content-Type` header is an *internet media type*, which consists of a type, subtype, and optional parameters. For example, `text/html; charset=UTF-8` specifies a type of "text," a subtype of "html," and a character encoding of UTF-8. The Internet Assigned Numbers Authority maintains an official list of internet media types. Often, you will hear "content type," "Internet media type," and "MIME type" used interchangeably. MIME (Multipurpose Internet Mail Extensions) was a precursor of internet media types and, for the most part, is equivalent.

Request Body

In addition to the request headers, a request can have a *body* (just like the body of a response is the actual content that's being returned). Normal `GET` requests don't have bodies, but `POST` requests usually do. The most common media type for `POST` bodies is `application/x-www-form-urlencoded`, which is simply encoded name/value pairs separated by ampersands (essentially the same format as a querystring). If the `POST` needs to support file uploads, the media type is `multipart/form-data`, which is a more complicated format. Lastly, Ajax requests can use `application/json` for the body. We'll learn more about request bodies in Chapter 8.

The Request Object

The *request object* (which is passed as the first parameter of a request handler, meaning you can name it whatever you want; it is common to name it `req` or `request`) starts its life as an instance of `http.IncomingMessage`, a core Node object. Express adds further functionality. Let's look at the most useful properties and methods of the request object (all of these methods are added by Express, except for `req.headers` and `req.url`, which originate in Node):

`req.params`
: An array containing the *named route parameters*. We'll learn more about this in Chapter 14.

`req.query`
: An object containing querystring parameters (sometimes called `GET` parameters) as name/value pairs.

`req.body`
: An object containing `POST` parameters. It is so named because `POST` parameters are passed in the body of the request, not in the URL as querystring parameters are. To make `req.body` available, you'll need middleware that can parse the body content type, which we will learn about in Chapter 10.

`req.route`
: Information about the currently matched route. This is primarily useful for route debugging.

`req.cookies`/`req.signedCookies`
: Objects containing cookie values passed from the client. See Chapter 9.

`req.headers`
: The request headers received from the client. This is an object whose keys are the header names and whose values are the header values. Note that this comes from the underlying `http.IncomingMessage` object, so you won't find it listed in the Express documentation.

`req.accepts(types)`
: A convenience method to determine whether the client accepts a given type or types (optional `types` can be a single MIME type, such as `application/json`, a comma-delimited list, or an array). This method is of primary interest to those writing public APIs; it is assumed that browsers will always accept HTML by default.

`req.ip`
: The IP address of the client.

`req.path`
: The request path (without protocol, host, port, or querystring).

`req.hostname`
: A convenience method that returns the hostname reported by the client. This information can be spoofed and should not be used for security purposes.

`req.xhr`
: A convenience property that returns `true` if the request originated from an Ajax call.

`req.protocol`
: The protocol used in making this request (for our purposes, it will be either `http` or `https`).

`req.secure`
: A convenience property that returns `true` if the connection is secure. This is equivalent to `req.protocol === 'https'`.

`req.url`/`req.originalUrl`
: A bit of a misnomer, these properties return the path and querystring (they do not include protocol, host, or port). `req.url` can be rewritten for internal routing purposes, but `req.originalUrl` is designed to remain the original request and querystring.

The Response Object

The *response object* (which is passed as the second parameter of a request handler, meaning you can name it whatever you want; it is common to name it `res`, `resp`, or `response`) starts its life as an instance of `http.ServerResponse`, a core Node object. Express adds further functionality. Let's look at the most useful properties and methods of the response object (all of these are added by Express):

`res.status(code)`
: Sets the HTTP status code. Express defaults to 200 (OK), so you will have to use this method to return a status of 404 (Not Found) or 500 (Server Error), or any other status code you want to use. For redirects (status codes 301, 302, 303, and 307), there is a method `redirect`, which is preferable. Note that `res.status` returns the response object, meaning you can chain calls: `res.status(404).send('Not found')`.

`res.set(name, value)`
: Sets a response header. This is not something you will normally be doing manually. You can also set multiple headers at once by passing a single object argument whose keys are the header names and whose values are the header values.

`res.cookie(name, value, [options])`, `res.clearCookie(name, [options])`
: Sets or clears cookies that will be stored on the client. This requires some middleware support; see Chapter 9.

`res.redirect([status], url)`
: Redirects the browser. The default redirect code is 302 (Found). In general, you should minimize redirection unless you are permanently moving a page, in which case you should use the code 301 (Moved Permanently).

`res.send(body)`
: Sends a response to the client. Express defaults to a content type of `text/html`, so if you want to change it to `text/plain` (for example), you'll have to call `res.type('text/plain')` before calling `res.send`. If `body` is an object or an array, the response is sent as JSON (with the content type being set appropriately), though if you want to send JSON, I recommend doing so explicitly by calling `res.json` instead.

`res.json(json)`
: Sends JSON to the client.

`res.jsonp(json)`
: Sends JSONP to the client.

`res.end()`
: Ends the connection without sending a response. To learn more about the differences between `res.send`, `res.json`, and `res.end`, see this article by Tamas Piros.

`res.type(type)`
: A convenience method to set the `Content-Type` header. This is essentially equivalent to `res.set(\'Content-Type ', type)`, except that it will also attempt to map file extensions to an internet media type if you provide a string without a slash in it. For example, `res.type(\'txt ')` will result in a `Content-Type` of `text/plain`. There are areas where this functionality could be useful (for example, automatically serving disparate multimedia files), but in general, you should avoid it in favor of explicitly setting the correct internet media type.

`res.format(object)`
: This method allows you to send different content depending on the `Accept` request header. This is of primary use in APIs, and we will discuss this more in Chapter 15. Here's a simple example: `res.format({'text/plain': 'hi there', 'text/html': '<b>hi there</b>'})`.

`res.attachment([filename])`, `res.download(path, [filename], [callback])`
: Both of these methods set a response header called `Content-Disposition` to `attachment`; this will prompt the browser to download the content instead of displaying it in a browser. You may specify `filename` as a hint to the browser. With `res.download`, you can specify the file to download, whereas `res.attachment` just sets the header; you still have to send content to the client.

`res.sendFile(path, [options], [callback])`
: This method will read a file specified by `path` and send its contents to the client. There should be little need for this method; it's easier to use the `static` middleware and put files you want available to the client in the *public* directory. However, if you want to have a different resource served from the same URL depending on some condition, this method could come in handy.

`res.links(links)`
: Sets the `Links` response header. This is a specialized header that has little use in most applications.

`res.locals`, `res.render(view, [locals], callback)`
: `res.locals` is an object containing *default* context for rendering views. `res.render` will render a view using the configured templating engine (the `locals` parameter to `res.render` shouldn't be confused with `res.locals`: it will override the context in `res.locals`, but context not overridden will still be available). Note that `res.render` will default to a response code of 200; use `res.status` to

specify a different response code. Rendering views will be covered in depth in Chapter 7.

Getting More Information

Because of JavaScript's prototypal inheritance, knowing exactly what you're dealing with can sometimes be challenging. Node provides you with objects that Express extends, and packages that you add may also extend those. Figuring out exactly what's available to you can be challenging sometimes. In general, I recommend working backward: if you're looking for some functionality, first check the Express API documentation. The Express API is pretty complete, and chances are, you'll find what you're looking for there.

If you need information that isn't documented, sometimes you have to dive into the Express source. I encourage you to do this! You'll probably find that it's a lot less intimidating than you might think. Here's a quick roadmap to where you'll find things in the Express source:

lib/application.js
: The main Express interface. If you want to understand how middleware is linked in or how views are rendered, this is the place to look.

lib/express.js
: A relatively short file that primarily provides the `createApplication` function (the default export of this file), which creates an Express application instance.

lib/request.js
: Extends Node's `http.IncomingMessage` object to provide a robust request object. For information about all the request object properties and methods, this is where to look.

lib/response.js
: Extends Node's `http.ServerResponse` object to provide the response object. For information about response object properties and methods, this is where to look.

lib/router/route.js
: Provides basic routing support. While routing is central to your app, this file is less than 230 lines long; you'll find that it's quite simple and elegant.

As you dig into the Express source code, you'll probably want to refer to the Node documentation, especially the section on the `HTTP` module.

Boiling It Down

This chapter has provided an overview of the request and response objects, which are the meat and potatoes of an Express application. However, the chances are that you will be using a small subset of this functionality most of the time. So let's break it down by functionality you'll be using most frequently.

Rendering Content

When you're rendering content, you'll be using `res.render` most often, which renders views within layouts, providing maximum value. Occasionally, you may want to write a quick test page, so you might use `res.send` if you just want a test page. You may use `req.query` to get querystring values, `req.session` to get session values, or `req.cookie`/`req.signedCookies` to get cookies. Example 6-1 to Example 6-8 demonstrate common content rendering tasks.

Example 6-1. Basic usage (ch06/02-basic-rendering.js)

```
// basic usage
app.get('/about', (req, res) => {
  res.render('about')
})
```

Example 6-2. Response codes other than 200 (ch06/03-different-response-codes.js)

```
app.get('/error', (req, res) => {
  res.status(500)
  res.render('error')
})

// or on one line...

app.get('/error', (req, res) => res.status(500).render('error'))
```

Example 6-3. Passing a context to a view, including querystring, cookie, and session values (ch06/04-view-with-content.js)

```
app.get('/greeting', (req, res) => {
  res.render('greeting', {
    message: 'Hello esteemed programmer!',
    style: req.query.style,
    userid: req.cookies.userid,
    username: req.session.username
  })
})
```

Example 6-4. Rendering a view without a layout (ch06/05-view-without-layout.js)

```
// the following layout doesn't have a layout file, so
// views/no-layout.handlebars must include all necessary HTML
app.get('/no-layout', (req, res) =>
  res.render('no-layout', { layout: null })
)
```

Example 6-5. Rendering a view with a custom layout (ch06/06-custom-layout.js)

```
// the layout file views/layouts/custom.handlebars will be used
app.get('/custom-layout', (req, res) =>
  res.render('custom-layout', { layout: 'custom' })
)
```

Example 6-6. Rendering plain text output (ch06/07-plaintext-output.js)

```
app.get('/text', (req, res) => {
  res.type('text/plain')
  res.send('this is a test')
})
```

Example 6-7. Adding an error handler (ch06/08-error-handler.js)

```
// this should appear AFTER all of your routes
// note that even if you don't need the "next" function, it must be
// included for Express to recognize this as an error handler
app.use((err, req, res, next) => {
  console.error('** SERVER ERROR: ' + err.message)
  res.status(500).render('08-error',
    { message: "you shouldn't have clicked that!" })
})
```

Example 6-8. Adding a 404 handler (ch06/09-custom-404.js)

```
// this should appear AFTER all of your routes
app.use((req, res) =>
  res.status(404).render('404')
)
```

Processing Forms

When you're processing forms, the information from the forms will usually be in `req.body` (or occasionally in `req.query`). You may use `req.xhr` to determine whether the request was an Ajax request or a browser request (this will be covered in depth in Chapter 8). See Example 6-9 and Example 6-10. For the following examples, you'll need to have body parser middleware linked in:

```
const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: false }))
```

We'll learn more about body parser middleware in Chapter 8.

Example 6-9. Basic form processing (ch06/10-basic-form-processing.js)

```
app.post('/process-contact', (req, res) => {
  console.log(`received contact from ${req.body.name} <${req.body.email}>`)
  res.redirect(303, '10-thank-you')
})
```

Example 6-10. More robust form processing (ch06/11-more-robust-form-processing.js)

```
app.post('/process-contact', (req, res) => {
  try {
    // here's where we would try to save contact to database or other
    // persistence mechanism...for now, we'll just simulate an error
    if(req.body.simulateError) throw new Error("error saving contact!")
    console.log(`contact from ${req.body.name} <${req.body.email}>`)
    res.format({
      'text/html': () => res.redirect(303, '/thank-you'),
      'application/json': () => res.json({ success: true }),
    })
  } catch(err) {
    // here's where we would handle any persistence failures
    console.error(`error processing contact from ${req.body.name} ` +
      `<${req.body.email}>`)
    res.format({
      'text/html': () =>  res.redirect(303, '/contact-error'),
      'application/json': () => res.status(500).json({
        error: 'error saving contact information' }),
    })
  }
})
```

Providing an API

When you're providing an API, much like processing forms, the parameters will usually be in `req.query`, though you can also use `req.body`. What's different about APIs is that you'll usually be returning JSON, XML, or even plain text, instead of HTML, and you'll often be using less common HTTP methods like `PUT`, `POST`, and `DELETE`. Providing an API will be covered in Chapter 15. Example 6-11 and Example 6-12 use the following "products" array (which would normally be retrieved from a database):

```
const tours = [
  { id: 0, name: 'Hood River', price: 99.99 },
  { id: 1, name: 'Oregon Coast', price: 149.95 },
]
```

The term *endpoint* is often used to describe a single function in an API.

Example 6-11. Simple GET endpoint returning only JSON (ch06/12-api.get.js)

```
app.get('/api/tours', (req, res) => res.json(tours))
```

Example 6-12 uses the `res.format` method in Express to respond according to the preferences of the client.

Example 6-12. GET endpoint that returns JSON, XML, or text (ch06/13-api-json-xml-text.js)

```
app.get('/api/tours', (req, res) => {
  const toursXml = '<?xml version="1.0"?><tours>' +
    tours.map(p =>
      `<tour price="${p.price}" id="${p.id}">${p.name}</tour>`
    ).join('') + '</tours>'
  const toursText = tours.map(p =>
      `${p.id}: ${p.name} (${p.price})`
    ).join('\n')
  res.format({
    'application/json': () => res.json(tours),
    'application/xml': () => res.type('application/xml').send(toursXml),
    'text/xml': () => res.type('text/xml').send(toursXml),
    'text/plain': () => res.type('text/plain').send(toursXml),
  })
})
```

In Example 6-13, the PUT endpoint updates a product and returns JSON. Parameters are passed in the request body (the `:id` in the route string tells Express to add an `id` property to `req.params`).

Example 6-13. PUT endpoint for updating (ch06/14-api-put.js)

```
app.put('/api/tour/:id', (req, res) => {
  const p = tours.find(p => p.id === parseInt(req.params.id))
  if(!p) return res.status(404).json({ error: 'No such tour exists' })
  if(req.body.name) p.name = req.body.name
  if(req.body.price) p.price = req.body.price
  res.json({ success: true })
})
```

Finally, Example 6-14 shows a DELETE endpoint.

Example 6-14. DELETE endpoint for deleting (ch06/15-api-del.js)

```
app.delete('/api/tour/:id', (req, res) => {
  const idx = tours.findIndex(tour => tour.id === parseInt(req.params.id))
  if(idx < 0) return res.json({ error: 'No such tour exists.' })
  tours.splice(idx, 1)
  res.json({ success: true })
})
```

Conclusion

I hope the micro-examples in this chapter gave you a feel for the kind of functionality that is common in an Express application. These examples are intended to be a quick reference you can revisit in the future.

In the next chapter, we'll dig deeper into templating, which we touched on in the rendering examples in this chapter.

CHAPTER 7

Templating with Handlebars

In this chapter, we'll cover *templating*, which is a technique for constructing and formating your content to display to the user. You can think of templating as an evolution of the form letter: "Dear [Name]: we regret to inform you nobody uses [Outdated Technology] anymore, but templating is alive and well!" To send that letter to a bunch of people, you just have to replace [Name] and [Outdated Technology].

> This process of replacing fields is sometimes called *interpolation*, which is just a fancy word for "supplying missing information" in this context.

While server-side templating is being fast supplanted by frontend frameworks like React, Angular, and Vue, it still has applications, like creating HTML email. Also, Angular and Vue both use a template-like approach to writing HTML, so what you learn about server-side templating will transfer to those frontend frameworks.

If you're coming from a PHP background, you may wonder what the fuss is all about: PHP is one of the first languages that could really be called a templating language. Almost all major languages that have been adapted for the web have included some kind of templating support. What is different today is that the *templating engine* is being decoupled from the language.

So what does templating look like? Let's start with what templating is replacing by considering the most obvious and straightforward way to generate one language from another (specifically, we'll generate some HTML with JavaScript):

```
document.write('<h1>Please Don\'t Do This</h1>')
document.write('<p><span class="code">document.write</span> is naughty,\n')
document.write('and should be avoided at all costs.</p>')
document.write('<p>Today\'s date is ' + new Date() + '.</p>')
```

Perhaps the only reason this seems "obvious" is that it's the way programming has always been taught:

```
10 PRINT "Hello world!"
```

In imperative languages, we're used to saying, "Do this, then do that, then do something else." For some things, this approach works fine. If you have 500 lines of JavaScript to perform a complicated calculation that results in a single number, and every step is dependent on the previous step, there's no harm in it. What if it's the other way around, though? You have 500 lines of HTML and 3 lines of JavaScript. Does it make sense to write `document.write` 500 times? Not at all.

Really, the problem boils down to this: switching context is problematic. If you're writing lots of JavaScript, it's inconvenient and confusing to be mixing in HTML. The other way isn't so bad. We're quite used to writing JavaScript in `<script>` blocks, but I hope you see the difference: there's still a context switch, and either you're writing HTML or you're in a `<script>` block writing JavaScript. Having JavaScript emit HTML is fraught with problems:

- You have to constantly worry about what characters need to be escaped and how to do that.
- Using JavaScript to generate HTML that itself includes JavaScript quickly leads to madness.
- You usually lose the nice syntax highlighting and other handy language-specific features your editor has.
- It can be much harder to spot malformed HTML.
- Your code is hard to visually parse.
- It can make it harder for other people to understand your code.

Templating solves the problem by allowing you to write in the target language, while at the same time providing the ability to insert dynamic data. Consider the previous example rewritten as a Mustache template:

```
<h1>Much Better</h1>
<p>No <span class="code">document.write</span> here!</p>
<p>Today's date is {{today}}.</p>
```

Now all we have to do is provide a value for `{{today}}`, and that's at the heart of templating languages.

There Are No Absolute Rules Except This One

I'm not suggesting that you should *never* write HTML in JavaScript, only that you should avoid it whenever possible. In particular, it's slightly more palatable in frontend code, especially if you're using a robust frontend framework. For example, this would pass with little comment from me:

```
document.querySelector('#error').innerHTML =
  'Something <b>very bad</b> happened!'
```

However, say that eventually mutated into this:

```
document.querySelector('#error').innerHTML =
  '<div class="error"><h3>Error</h3>' +
  '<p>Something <b><a href="/error-detail/' + errorNumber +
  '">very bad</a></b> ' +
  'happened.  <a href="/try-again">Try again<a>, or ' +
  '<a href="/contact">contact support</a>.</p></div>'
```

Then I might suggest it's time to employ a template. The point is, I suggest you develop good judgment when deciding where to draw the line between HTML in strings and using templates. I would err on the side of templates, however, and avoid generating HTML with JavaScript except for the simplest cases.

Choosing a Template Engine

In the Node world, you have many templating engines to choose from, so how to pick? It's a complicated question, and very much depends on your needs. Here are some criteria to consider, though:

Performance
: Clearly, you want your templating engine to be as fast as possible. It's not something you want slowing down your website.

Client, server, or both?
: Most, but not all, templating engines are available on both the server and client sides. If you need to use templates in both realms (and you will), I recommend you pick something that is equally capable in either capacity.

Abstraction
: Do you want something familiar (like normal HTML with curly brackets thrown in, for example), or do you secretly hate HTML and would love something that saves you from all those angle brackets? Templating (especially server-side templating) gives you some choices here.

These are just some of the more prominent criteria in selecting a templating language. Templating options are pretty mature at this point, so you probably can't go too wrong with whatever you pick.

Express allows you to use any templating engine you wish, so if Handlebars is not to your liking, you'll find it's easy to switch it out. If you want to explore your options, you can use the fun and useful Template-Engine-Chooser (it's still useful even though it's no longer being updated).

Let's take a look at a particularly abstract templating engine before we get to our discussion of Handlebars.

Pug: A Different Approach

Whereas most templating engines take an HTML-centric approach, Pug stands out by abstracting the details of HTML away from you. It is also worth noting that Pug is the brainchild of TJ Holowaychuk, the same person who brought us Express. It should come as no surprise, then, that Pug integration with Express is very good. The approach that Pug takes is noble: at its core is the assertion that HTML is a fussy and tedious language to write by hand. Let's take a look at what a Pug template looks like, along with the HTML it will output (originally taken from the Pug home page and modified slightly to fit the book format):

```
doctype html                        <!DOCTYPE html>
html(lang="en")                     <html lang="en">
  head                              <head>
    title= pageTitle                <title>Pug Demo</title>
    script.                         <script>
      if (foo) {                        if (foo) {
         bar(1 + 5)                         bar(1 + 5)
      }                                 }
  body                              </script>
                                    <body>
    h1 Pug                         <h1>Pug</h1>
    #container                      <div id="container">
      if youAreUsingPug
        p You are amazing           <p>You are amazing</p>
      else
        p Get on it!
      p.                            <p>
        Pug is a terse and           Pug is a terse and
        simple templating             simple templating
        language with a               language with a
        strong focus on               strong focus on
        performance and               performance and
        powerful features.            powerful features.
                                    </p>
                                    </body>
                                    </html>
```

Pug certainly represents a lot less typing (no more angle brackets or closing tags). Instead, it relies on indentation and some commonsense rules, making it easier to say what you mean. Pug has an additional advantage: theoretically, when HTML itself changes, you can simply get Pug to retarget the newest version of HTML, allowing you to "future proof" your content.

As much as I admire the Pug philosophy and the elegance of its execution, I've found that I don't want the details of HTML abstracted away from me. As a web developer, HTML is at the heart of everything I do, and if the price is wearing out the angle bracket keys on my keyboard, then so be it. A lot of frontend developers I talk to feel the same, so maybe the world just isn't ready for Pug.

Here's where we'll part ways with Pug; you won't be seeing it in this book. However, if the abstraction appeals to you, you will certainly have no problems using Pug with Express, and there are plenty of resources to help you do so.

Handlebars Basics

Handlebars is an extension of Mustache, another popular templating engine. I recommend Handlebars for its easy JavaScript integration (both frontend and backend) and familiar syntax. For me, it strikes all the right balances and is what we'll be focusing on in this book. The concepts we're discussing are broadly applicable to other templating engines, though, so you will be well prepared to try different templating engines if Handlebars doesn't strike your fancy.

The key to understanding templating is understanding the concept of *context*. When you render a template, you pass the templating engine an object called the *context object*, and this is what allows replacements to work.

For example, if my context object is

```
{ name: 'Buttercup' }
```

and my template is

```
<p>Hello, {{name}}!</p>
```

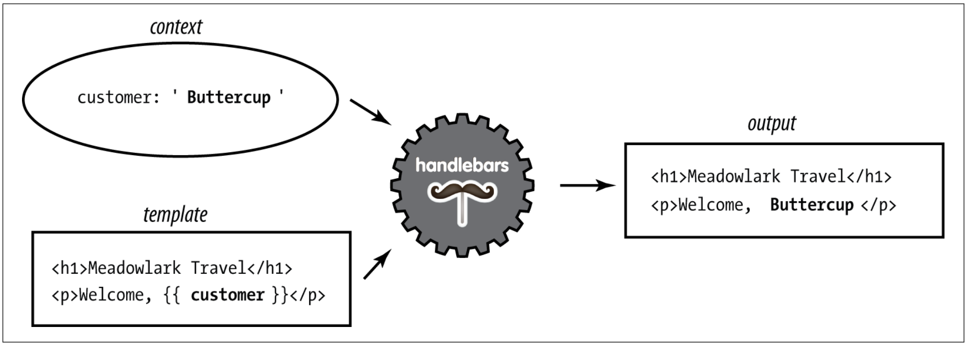
then `{{name}}` will be replaced with `Buttercup`. What if you want to pass HTML to the template? For example, if our context was instead

```
{ name: '<b>Buttercup</b>' }
```

then using the previous template will result in `<p>Hello, &lt;b&gt;Buttercup&lt;b&gt;</p>`, which is probably not what you're looking for. To solve this problem, simply use three curly brackets instead of two: `{{{name}}}`.

While we've already established that we should avoid writing HTML in JavaScript, the ability to turn off HTML escaping with triple curly brackets has some important uses. For example, if you were building a content management system (CMS) with what you see is what you get (WYSIWYG) editors, you would probably want to be able to pass HTML to your views. Also, the ability to render properties from the context without HTML escaping is important for *layouts* and *sections*, which we'll learn about shortly.

In Figure 7-1, we see how the Handlebars engine uses the context (represented by an oval) combined with the template to render HTML.



```
customer: ' Buttercup '
```

```
<h1>Meadowlark Travel</h1>
<p>Welcome, {{ customer }}</p>
```

```
<h1>Meadowlark Travel</h1>
<p>Welcome,  Buttercup </p>
```

Figure 7-1. Rendering HTML with Handlebars

Comments

Comments in Handlebars look like `{{! comment goes here }}`. It's important to understand the distinction between Handlebars comments and HTML comments. Consider the following template:

```
{{! super-secret comment }}
<!-- not-so-secret comment -->
```

Assuming this is a server-side template, the super-secret comment will never be sent to the browser, whereas the not-so-secret comment will be visible if the user inspects the HTML source. You should prefer Handlebars comments for anything that exposes implementation details, or anything else you don't want exposed.

Blocks

Things start to get more complicated when you consider *blocks*. Blocks provide flow control, conditional execution, and extensibility. Consider the following context object:

```
{
  currency: {
    name: 'United States dollars',
    abbrev: 'USD',
  },
  tours: [
    { name: 'Hood River', price: '$99.95' },
    { name: 'Oregon Coast', price: '$159.95' },
  ],
  specialsUrl: '/january-specials',
  currencies: [ 'USD', 'GBP', 'BTC' ],
}
```

Now let's examine a template we can pass that context to:

```
<ul>
  {{#each tours}}
    {{! I'm in a new block...and the context has changed }}
    <li>
      {{name}} - {{price}}
      {{#if ../currencies}}
        ({{../currency.abbrev}})
      {{/if}}
    </li>
  {{/each}}
</ul>
{{#unless currencies}}
  <p>All prices in {{currency.name}}.</p>
{{/unless}}
{{#if specialsUrl}}
  {{! I'm in a new block...but the context hasn't changed (sortof) }}
  <p>Check out our <a href="{{specialsUrl}}">specials!</p>
{{else}}
  <p>Please check back often for specials.</p>
{{/if}}
<p>
  {{#each currencies}}
    <a href="#" class="currency">{{.}}</a>
  {{else}}
    Unfortunately, we currently only accept {{currency.name}}.
  {{/each}}
</p>
```

A lot is going on in this template, so let's break it down. It starts off with the `each` helper, which allows us to iterate over an array. What's important to understand is that between `{{#each tours}}` and `{{/each tours}}`, the context changes. On the first pass, it changes to `{ name: 'Hood River', price: '$99.95' }`, and on the second pass, the context is `{ name: 'Oregon Coast', price: '$159.95' }`. So within that block, we can refer to `{{name}}` and `{{price}}`. However, if we want to access the `currency` object, we have to use `../` to access the *parent* context.

If a property of the context is itself an object, we can access its properties as normal with a period, such as `{{currency.name}}`.

Both `if` and `each` have an optional `else` block (with `each`, if there are no elements in the array, the `else` block will execute). We've also used the `unless` helper, which is essentially the opposite of the `if` helper: it executes only if the argument is false.

The last thing to note about this template is the use of `{{.}}` in the `{{#each currencies}}` block. `{{.}}` refers to the current context; in this case, the current context is simply a string in an array that we want to print out.

> Accessing the current context with a lone period has another use: it can distinguish helpers (which we'll learn about soon) from properties of the current context. For example, if you have a helper called `foo` and a property in the current context called `foo`, `{{foo}}` refers to the helper, and `{{./foo}}` refers to the property.

Server-Side Templates

Server-side templates allow you to render HTML *before* it's sent to the client. Unlike client-side templating, where the templates are available for the curious user who knows how to view the HTML source, your users will never see your server-side template or the context objects used to generate the final HTML.

Server-side templates, in addition to hiding your implementation details, support template *caching*, which is important for performance. The templating engine will cache compiled templates (recompiling and recaching only when the template itself changes), which will improve the performance of templated views. By default, view caching is disabled in development mode and enabled in production mode. If you want to explicitly enable view caching, you can do so thusly:

```
app.set('view cache', true)
```

Out of the box, Express supports Pug, EJS, and JSHTML. We've already discussed Pug, and I find little to recommend EJS or JSHTML (neither go far enough, syntactically, for my taste). So we'll need to add a Node package that provides Handlebars support for Express:

```
npm install express-handlebars
```

Then we'll link it into Express (*ch07/00/meadowlark.js* in the companion repo):

```
const expressHandlebars = require('express-handlebars')
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
})
app.set('view engine', 'handlebars')
```

express-handlebars expects Handlebars templates to have the *.handlebars* extension. I've grown used to this, but if it's too wordy for you, you can change the extension to the also common *.hbs* when you create the express-handlebars instance: app.engine('handlebars', expressHandlebars({ extname: '.hbs' })).

Views and Layouts

A *view* usually represents an individual page on your website (though it could represent an Ajax-loaded portion of a page, an email, or anything else for that matter). By default, Express looks for views in the *views* subdirectory. A *layout* is a special kind of view—essentially, a template for templates. Layouts are essential because most (if not all) of the pages on your site will have an almost identical layout. For example, they must have an <html> element and a <title> element, they usually all load the same CSS files, and so on. You don't want to have to duplicate that code for every single page, which is where layouts come in. Let's look at a bare-bones layout file:

```
<!doctype html>
<html>
  <head>
    <title>Meadowlark Travel</title>
    <link rel="stylesheet" href="/css/main.css">
  </head>
  <body>
    {{{body}}}
  </body>
</html>
```

Notice the text inside the <body> tag: {{{body}}}. That's so the view engine knows where to render the content of your view. It's important to use three curly brackets instead of two: our view is most likely to contain HTML, and we don't want Handlebars trying to escape it. Note that there's no restriction on where you place the {{{body}}} field. For example, if you were building a responsive layout in Bootstrap, you would probably want to put your view inside a container <div>. Also, common page elements like headers and footers usually live in the layout, not the view. Here's an example:

```
<!-- ... -->
<body>
  <div class="container">
    <header>
      <div class="container">
        <h1>Meadowlark Travel</h1>
        <img src="/img/logo.png" alt="Meadowlark Travel Logo">
      </div>
    </header>
    <div class="container">
```

```
      {{{body}}}
    </div>
    <footer>&copy; 2019 Meadowlark Travel</footer>
  </div>
</body>
```

In Figure 7-2, we see how the template engine combines the view, layout, and context. The important thing that this diagram makes clear is the order of operations. The *view is rendered first*, before the layout. At first, this may seem counterintuitive: the view is being rendered *inside* the layout, so shouldn't the layout be rendered first? While it could technically be done this way, there are advantages to doing it in reverse. Particularly, it allows the view itself to further customize the layout, which will come in handy when we discuss *sections* later in this chapter.

Because of the order of operations, you can pass a property called `body` into the view, and it will render correctly in the view. However, when the layout is rendered, the value of `body` will be overwritten by the rendered view.

Using Layouts (or Not) in Express

Chances are, most (if not all) of your pages will use the same layout, so it doesn't make sense to keep specifying the layout every time we render a view. You'll notice that when we created the view engine, we specified the name of the default layout:

```
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
})
```

By default, Express looks for views in the *views* subdirectory, and layouts in *views/layouts*. So if you have a view *views/foo.handlebars*, you can render it this way:

```
app.get('/foo', (req, res) => res.render('foo'))
```

It will use *views/layouts/main.handlebars* as the layout. If you don't want to use a layout at all (meaning you'll have to have all of the boilerplate in the view), you can specify `layout: null` in the context object:

```
app.get('/foo', (req, res) => res.render('foo', { layout: null }))
```

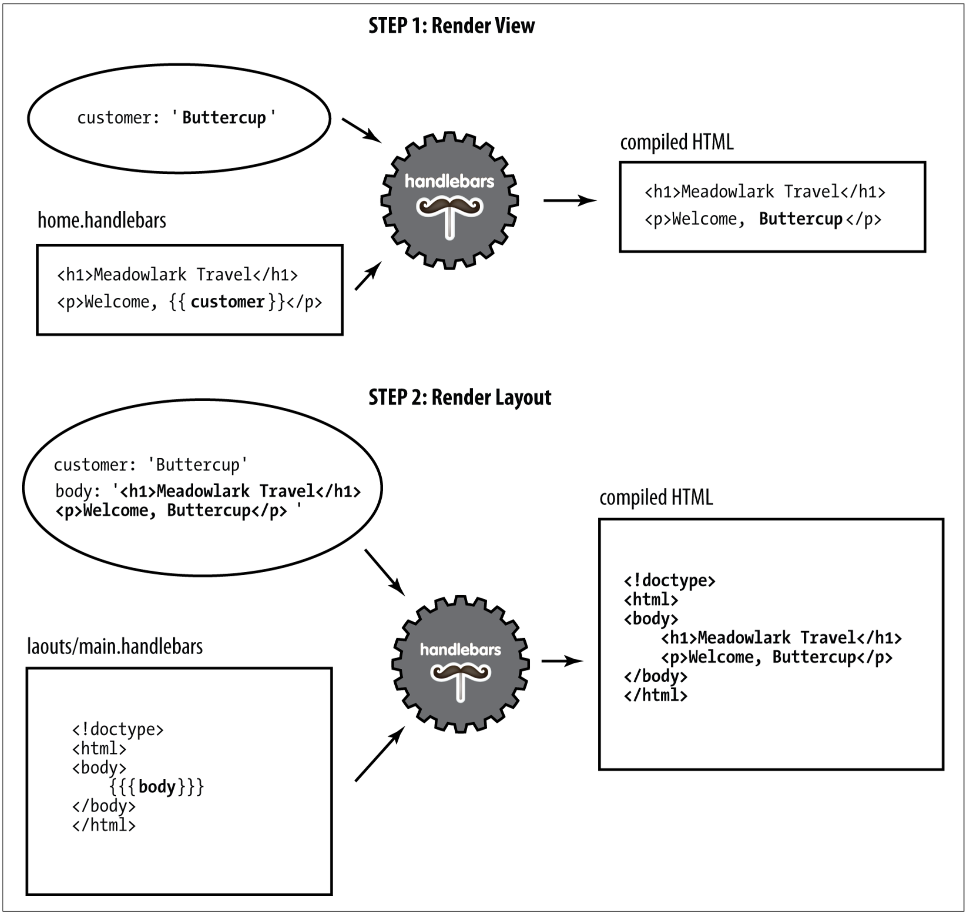


Figure 7-2. Rendering a view with a layout

Or, if we want to use a different template, we can specify the template name:

```
app.get('/foo', (req, res) => res.render('foo', { layout: 'microsite' }))
```

This will render the view with layout *views/layouts/microsite.handlebars*.

Keep in mind that the more templates you have, the more basic HTML layout you have to maintain. On the other hand, if you have pages that are substantially different in layout, it may be worth it; you have to find a balance that works for your projects.

Sections

One technique I'm borrowing from Microsoft's excellent *Razor* template engine is the idea of *sections*. Layouts work well if all of your view fits neatly within a single element in your layout, but what happens when your view needs to inject itself into different parts of your layout? A common example of this is a view needing to add something to the `<head>` element or to insert a `<script>`, which is sometimes the very last thing in the layout, for performance reasons.

Neither Handlebars nor `express-handlebars` has a built-in way to do this. Fortunately, Handlebars helpers make this really easy. When we instantiate the Handlebars object, we'll add a helper called `section` (*ch07/01/meadowlark.js* in the companion repo):

```
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
  helpers: {
    section: function(name, options) {
      if(!this._sections) this._sections = {}
      this._sections[name] = options.fn(this)
      return null
    },
  },
}))
```

Now we can use the `section` helper in a view. Let's add a view (*views/section-test.handlebars*) to add something to the `<head>` and a script:

```
{{#section 'head'}}
  <!-- we want Google to ignore this page -->
  <meta name="robots" content="noindex">
{{/section}}

<h1>Test Page</h1>
<p>We're testing some script stuff.</p>

{{#section 'scripts'}}
  <script>
    document.querySelector('body')
      .insertAdjacentHTML('beforeEnd', '<small>(scripting works!)</small>')
  </script>
{{/section}}
```

Now in our layout, we can place the sections just as we place `{{{body}}}`:

```
{{#section 'head'}}
  <!-- we want Google to ignore this page -->
  <meta name="robots" content="noindex">
{{/section}}

<h1>Test Page</h1>
<p>We're testing some script stuff.</p>

{{#section 'scripts'}}
  <script>
    const div = document.createElement('div')
    div.appendChild(document.createTextNode('(scripting works!)'))
    document.querySelector('body').appendChild(div)
  </script>
{{/section}}
```

Partials

Very often, you'll have components that you want to reuse on different pages (sometimes called *widgets* in frontend circles). One way to achieve that with templates is to use *partials* (so named because they don't render a whole view or a whole page). Let's imagine we want a Current Weather component that displays the current weather conditions in Portland, Bend, and Manzanita. We want this component to be reusable so we can easily put it on whatever page we want, so we'll use a partial. First, we create a partial file, *views/partials/weather.handlebars*:

```
<div class="weatherWidget">
  {{#each partials.weatherContext}}
    <div class="location">
      <h3>{{location.name}}</h3>
      <a href="{{location.forecastUrl}}">
        <img src="{{iconUrl}}" alt="{{weather}}">
        {{weather}}, {{temp}}
      </a>
    </div>
  {{/each}}
  <small>Source: <a href="https://www.weather.gov/documentation/services-web-api">
    National Weather Service</a></small>
</div>
```

Note that we namespace our context by starting with `partials.weatherContext`. Since we want to be able to use the partial on any page, it's not practical to pass the context in for every view, so instead we use `res.locals` (which is available to every view). But because we don't want to interfere with the context specified by individual views, we put all partial context in the `partials` object.

> `express-handlebars` allows you to pass in partial templates as part of the context. For example, if you add `partials.foo = "Template!"` to your context, you can render this partial with `{{> foo}}`. This usage will override any *.handlebars* view files, which is why we used `partials.weatherContext` earlier, instead of `partials.weather`, which would override *views/partials/weather.handlebars*.

In Chapter 19, we'll see how to get current weather information from the free National Weather Service API. For now, we're just going to use dummy data returned from a function we'll call `getWeatherData`.

In this example, we want this weather data to be available to any view, and the best mechanism for that is middleware (which we'll learn more about in Chapter 10). Our middleware will inject the weather data into the `res.locals.partials` object, which will make it available as the context for our partial.

To make our middleware more testable, we'll put it in its own file, *lib/middleware/weather.js* (*ch07/01/lib/middleware/weather.js* in the companion repo):

```
const getWeatherData = () => Promise.resolve([
  {
    location: {
      name: 'Portland',
      coordinates: { lat: 45.5154586, lng: -122.6793461 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PQR/112,103/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra,40?size=medium',
    weather: 'Chance Showers And Thunderstorms',
    temp: '59 F',
  },
  {
    location: {
      name: 'Bend',
      coordinates: { lat: 44.0581728, lng: -121.3153096 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PDT/34,40/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra_sct,50?size=medium',
    weather: 'Scattered Showers And Thunderstorms',
    temp: '51 F',
  },
  {
    location: {
      name: 'Manzanita',
      coordinates: { lat: 45.7184398, lng: -123.9351354 },
    },
    forecastUrl: 'https://api.weather.gov/gridpoints/PQR/73,120/forecast',
    iconUrl: 'https://api.weather.gov/icons/land/day/tsra,90?size=medium',
    weather: 'Showers And Thunderstorms',
    temp: '55 F',
  },
])

const weatherMiddleware = async (req, res, next) => {
  if(!res.locals.partials) res.locals.partials = {}
  res.locals.partials.weatherContext = await getWeatherData()
  next()
}

module.exports = weatherMiddleware
```

Now that everything is set up, all we have to do is use the partial in a view. For example, to put our widget on the home page, edit *views/home.handlebars*:

```
<h2>Home</h2>
{{> weather}}
```

The `{{> partial_name}}` syntax is how you include a partial in a view: `express-handlebars` will know to look in *views/partials* for a view called *partial_name.handlebars* (or *weather.handlebars*, in our example).

`express-handlebars` supports subdirectories, so if you have a lot of partials, you can organize them. For example, if you have some social media partials, you could put them in the *views/partials/social* directory and include them using `{{> social/facebook}}`, `{{> social/twitter}}`, etc.

Perfecting Your Templates

Your templates are at the heart of your website. A good template structure will save you development time, promote consistency across your website, and reduce the number of places that layout quirks can hide. To achieve these benefits, though, you must spend some time crafting your templates carefully. Deciding how many templates you should have is an art; generally, fewer is better, but there is a point of diminishing returns, depending on the uniformity of your pages. Your templates are also your first line of defense against cross-browser compatibility issues and valid HTML. They should be lovingly crafted and maintained by someone who is well versed in frontend development. A great place to start—especially if you're new—is HTML5 Boilerplate. In the previous examples, we've been using a minimal HTML5 template to fit the book format, but for our actual project, we'll be using HTML5 Boilerplate.

Another popular place to start with your template are third-party themes. Sites like Themeforest and WrapBootstrap have hundreds of ready-to-use HTML5 themes that you can use as a starting place for your template. Using a third-party theme starts with taking the primary file (usually *index.html*), renaming it to *main.handlebars* (or whatever you choose to call your layout file), and placing any resources (CSS, JavaScript, images) in the *public* directory you use for static files. Then you'll have to edit the template file and figure out where you want to put the `{{{body}}}` expression.

Depending on the elements of your template, you may want to move some of them into partials. A great example is a *hero* (a tall banner designed to grab the user's attention. If the hero appears on every page (probably a poor choice), you would leave the hero in the template file. If it appears on only one page (usually the home page), then it would go only in that view. If it appears on several—but not all—pages, then you might consider putting it in a partial. The choice is yours, and herein lies the artistry of making a unique, captivating website.

Conclusion

We've seen how templating can make our code easier to write, read, and maintain. Thanks to templates, we don't have to painfully cobble together HTML from JavaScript strings; we can write HTML in our favorite editor and use a compact and easy-to-read templating language to make it dynamic.

Now that we've seen how to format our content for display, we'll turn our attention to how to get data *into* our system with HTML forms.

CHAPTER 8

Form Handling

The usual way you collect information from your users is to use HTML *forms*. Whether you let the browser submit the form normally, use Ajax, or employ fancy frontend controls, the underlying mechanism is generally still an HTML form. In this chapter, we'll discuss the different methods for handling forms, form validation, and file uploads.

Sending Client Data to the Server

Broadly speaking, your two options for sending client data to the server are the querystring and the request body. Normally, if you're using the querystring, you're making a `GET` request, and if you're using the request body, you're using a `POST` request. (The HTTP protocol doesn't prevent you from doing it the other way around, but there's no point to it: best to stick to standard practice here.)

It is a common misperception that `POST` is secure and `GET` is not: in reality, both are secure if you use HTTPS, and neither is secure if you don't. If you're not using HTTPS, an intruder can look at the body data for a `POST` just as easily as the querystring of a `GET` request. However, if you're using `GET` requests, your users will see all of their input (including hidden fields) in the querystring, which is ugly and messy. Also, browsers often place limits on querystring length (there is no such restriction for body length). For these reasons, I generally recommend using `POST` for form submission.

HTML Forms

This book is focusing on the server side, but it's important to understand some basics about constructing HTML forms. Here's a simple example:

```
<form action="/process" method="POST">
    <input type="hidden" name="hush" val="hidden, but not secret!">
    <div>
        <label for="fieldColor">Your favorite color: </label>
        <input type="text" id="fieldColor" name="color">
    </div>
    <div>
        <button type="submit">Submit</button>
    </div>
</form>
```

Notice the method is specified explicitly as `POST` in the `<form>` tag; if you don't do this, it defaults to `GET`. The `action` attribute specifies the URL that will receive the form when it's posted. If you omit this field, the form will be submitted to the same URL the form was loaded from. I recommend that you always provide a valid `action`, even if you're using Ajax (this is to prevent you from losing data; see Chapter 22 for more information).

From the server's perspective, the important attributes in the `<input>` fields are the `name` attributes: that's how the server identifies the field. It's important to understand that the `name` attribute is distinct from the `id` attribute, which should be used for styling and frontend functionality only (it is not passed to the server).

Note the hidden field: this will not render in the user's browser. However, you should not use it for secret or sensitive information; all the user has to do is examine the page source, and the hidden field will be exposed.

HTML does not restrict you from having multiple forms on the same page (this was an unfortunate restriction of some early server frameworks; ASP, I'm looking at you). I recommend keeping your forms logically consistent; a form should contain all the fields you would like submitted at once (optional/empty fields are OK) and none that you don't. If you have two different actions on a page, use two different forms. An example of this would be to have a form for a site search and a separate form for signing up for an email newsletter. It is possible to use one large form and figure out what action to take based on what button a person clicked, but it is a headache and often not friendly for people with disabilities (because of the way accessibility browsers render forms).

When the user submits the form in this example, the */process* URL will be invoked, and the field values will be transmitted to the server in the request body.

Encoding

When the form is submitted (either by the browser or via Ajax), it must be encoded somehow. If you don't explicitly specify an encoding, it defaults to `application/x-www-form-urlencoded` (this is just a lengthy media type for "URL encoded"). This is a basic, easy-to-use encoding that's supported by Express out of the box.

If you need to upload files, things get more complicated. There's no easy way to send files using URL encoding, so you're forced to use the `multipart/form-data` encoding type, which is not handled directly by Express.

Different Approaches to Form Handling

If you're not using Ajax, your only option is to submit the form through the browser, which will reload the page. However, *how* the page is reloaded is up to you. There are two things to consider when processing forms: what path handles the form (the action) and what response is sent to the browser.

If your form uses `method="POST"` (which is recommended), it is quite common to use the same path for displaying the form and processing the form: these can be distinguished because the former is a `GET` request, and the latter is a `POST` request. If you take this approach, you can omit the `action` attribute on the form.

The other option is to use a separate path to process the form. For example, if your contact page uses the path */contact*, you might use the path */process-contact* to process the form (by specifying `action="/process-contact"`). If you use this approach, you have the option of submitting the form via `GET` (which I do not recommend; it needlessly exposes your form fields on the URL). Using a separate endpoint for form submission might be preferred if you have multiple URLs that use the same submission mechanism (for example, you might have an email sign-up box on multiple pages on the site).

Whatever path you use to process the form, you have to decide what response to send back to the browser. Here are your options:

Direct HTML response

After processing the form, you can send HTML directly back to the browser (a view, for example). This approach will produce a warning if the user attempts to reload the page and can interfere with bookmarking and the Back button, and for these reasons, it is not recommended.

302 redirect

While this is a common approach, it is a misuse of the original meaning of the 302 (Found) response code. HTTP 1.1 added the 303 (See Other) response code, which is preferable. Unless you have reason to target browsers made before 1996, you should use 303 instead.

303 redirect
: The 303 (See Other) response code was added in HTTP 1.1 to address the misuse of the 302 redirect. The HTTP specification specifically indicates that the browser should use a `GET` request when following a 303 redirect, regardless of the original method. This is the recommended method for responding to a form submission request.

Since the recommendation is that you respond to a form submission with a 303 redirect, the next question is, "Where does the redirection point to?" The answer to that is up to you. Here are the most common approaches:

Redirect to dedicated success/failure pages
: This method requires that you dedicate URLs for appropriate success or failure messages. For example, if the user signs up for promotional emails but there was a database error, you might want to redirect to */error/database*. If a user's email address were invalid, you could redirect to */error/invalid-email*, and if everything was successful, you could redirect to */promo-email/thank-you*. One of the advantages of this method is that it's analytics friendly: the number of visits to your */promo-email/thank-you* page should roughly correlate to the number of people signing up for your promotional email. It is also straightforward to implement. It has some downsides, however. It does mean you have to allocate URLs to every possibility, which means pages to design, write copy for, and maintain. Another disadvantage is that the user experience can be suboptimal: users like to be thanked, but then they have to navigate back to where they were or where they want to go next. This is the approach we'll be using for now: we'll switch to using flash messages (not to be confused with Adobe Flash) in Chapter 9.

Redirect to the original location with a flash message
: For small forms that are scattered throughout your site (like an email sign-up, for example), the best user experience is not to interrupt the user's navigation flow. That is, provide a way to submit an email address without leaving the page. One way to do this, of course, is Ajax, but if you don't want to use Ajax (or you want your fallback mechanism to provide a good user experience), you can redirect back to the page the user was originally on. The easiest way to do this is to use a hidden field in the form that's populated with the current URL. Since you want there to be some feedback that the user's submission was received, you can use flash messages.

Redirect to a new location with a flash message
: Large forms generally have their own page, and it doesn't make sense to stay on that page once you've submitted the form. In this situation, you have to make an intelligent guess about where the user might want to go next and redirect accord-

ingly. For example, if you're building an admin interface, and you have a form to create a new vacation package, you might reasonably expect your user to want to go to the admin page that lists all vacation packages after submitting the form. However, you should still employ a flash message to give the user feedback about the result of the submission.

If you are using Ajax, I recommend a dedicated URL. It's tempting to start Ajax handlers with a prefix (for example, */ajax/enter*), but I discourage this approach: it's attaching implementation details to a URL. Also, as we'll see shortly, your Ajax handler should handle regular browser submissions as a fail-safe.

Form Handling with Express

If you're using `GET` for your form handling, your fields will be available on the `req.query` object. For example, if you have an HTML input field with a name attribute of `email`, its value will be passed to the handler as `req.query.email`. There's really not much more that needs to be said about this approach; it's just that simple.

If you're using `POST` (which I recommend), you'll have to link in middleware to parse the URL-encoded body. First, install the `body-parser` middleware (`npm install body-parser`); then, link it in (*ch08/meadowlark.js* in the companion repo):

```
const bodyParser = require('body-parser')
app.use(bodyParser.urlencoded({ extended: true }))
```

Once you've linked in `body-parser`, you'll find that `req.body` now becomes available for you, and that's where all of your form fields will be made available. Note that `req.body` doesn't prevent you from using the querystring. Let's go ahead and add a form to Meadowlark Travel that lets the user sign up for a mailing list. For demonstration's sake, we'll use the querystring, a hidden field, and visible fields in */views/newsletter-signup.handlebars*:

```
<h2>Sign up for our newsletter to receive news and specials!</h2>
<form class="form-horizontal" role="form"
    action="/newsletter-signup/process?form=newsletter" method="POST">
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <div class="form-group">
    <label for="fieldName" class="col-sm-2 control-label">Name</label>
    <div class="col-sm-4">
      <input type="text" class="form-control"
      id="fieldName" name="name">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
    <div class="col-sm-4">
      <input type="email" class="form-control" required
          id="fieldEmail" name="email">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-4">
      <button type="submit" class="btn btn-primary">Register</button>
    </div>
  </div>
</form>
```

Note we are using Bootstrap styles, as we will be throughout the rest of the book. If you are unfamiliar with Bootstrap, you may want to refer to the Bootstrap documentation.

We've already linked in our body parser, so now we need to add handlers for our newsletter sign-up page, processing function, and thank-you page (*ch08/lib/handlers.js* in the companion repo):

```
exports.newsletterSignup = (req, res) => {
  // we will learn about CSRF later...for now, we just
  // provide a dummy value
  res.render('newsletter-signup', { csrf: 'CSRF token goes here' })
}
exports.newsletterSignupProcess = (req, res) => {
  console.log('Form (from querystring): ' + req.query.form)
  console.log('CSRF token (from hidden form field): ' + req.body._csrf)
  console.log('Name (from visible form field): ' + req.body.name)
  console.log('Email (from visible form field): ' + req.body.email)
  res.redirect(303, '/newsletter-signup/thank-you')
}
exports.newsletterSignupThankYou = (req, res) =>
  res.render('newsletter-signup-thank-you')
```

(If you haven't already, create a *views/newsletter-signup-thank-you.handlebars* file.)

Lastly, we'll link our handlers into our application (*ch08/meadowlark.js* in the companion repo):

```
app.get('/newsletter-signup', handlers.newsletterSignup)
app.post('/newsletter-signup/process', handlers.newsletterSignupProcess)
app.get('/newsletter-signup/thank-you', handlers.newsletterSignupThankYou)
```

That's all there is to it. Note that in our handler, we're redirecting to a "thank you" view. We could render a view here, but if we did, the URL field in the visitor's browser would remain */process*, which could be confusing. Issuing a redirect solves that problem.

It's important that you use a 303 (or 302) redirect, not a 301 redirect in this instance. 301 redirects are "permanent," meaning your browser may cache the redirection destination. If you use a 301 redirect and try to submit the form a second time, your browser may bypass the `/process` handler altogether and go directly to */thank-you* since it correctly believes the redirect to be permanent. The 303 redirect, on the other hand, tells your browser, "Yes, your request is valid, and you can find your response here," and does not cache the redirect destination.

With most frontend frameworks, it is more common to see form data sent in JSON form with the `fetch` API, which we'll be looking at next. However, it's still good to understand how browsers handle form submission by default, as you will still find forms implemented this way.

Let's turn our attention to form submission with `fetch`.

Using Fetch to Send Form Data

Using the `fetch` API to send JSON-encoded form data is a much more modern approach that gives you more control over the client/server communication and allows you to have fewer page refreshes.

Since we are not making round-trip requests to the server, we no longer have to worry about redirects and multiple user URLs (we'll still have a separate URL for the form processing itself), and for that reason, we'll just consolidate our entire "newsletter signup experience" under a single URL called */newsletter*.

Let's start with the frontend code. The contents of the HTML form itself don't need to be changed (the fields and layout are all the same), but we don't need to specify an `action` or `method`, and we'll wrap our form in a container `<div>` element that will make it easier to display our "thank you" message:

```
<div id="newsletterSignupFormContainer">
  <form class="form-horizontal role="form" id="newsletterSignupForm">
    <!-- the rest of the form contents are the same... -->
  </form>
</div>
```

Then we'll have a script that intercepts the form submit event and cancels it (using `Event#preventDefault`) so we can handle the form processing ourselves (*ch08/views/newsletter.handlebars* in the companion repo):

```
<script>
  document.getElementById('newsletterSignupForm')
    .addEventListener('submit', evt => {
      evt.preventDefault()
      const form = evt.target
```

```
        const body = JSON.stringify({
          _csrf: form.elements._csrf.value,
          name: form.elements.name.value,
          email: form.elements.email.value,
        })
        const headers = { 'Content-Type': 'application/json' }
        const container =
          document.getElementById('newsletterSignupFormContainer')
        fetch('/api/newsletter-signup', { method: 'post', body, headers })
          .then(resp => {
            if(resp.status < 200 || resp.status >= 300)
              throw new Error(`Request failed with status ${resp.status}`)
            return resp.json()
          })
          .then(json => {
            container.innerHTML = '<b>Thank you for signing up!</b>'
          })
          .catch(err => {
            container.innerHTML = `<b>We're sorry, we had a problem ` +
              `signing you up. Please <a href="/newsletter">try again</a>`
          })
    })
</script>
```

Now in our server file (*meadowlark.js*), make sure we're linking in middleware that can parse JSON bodies, before we specify our two endpoints:

```
app.use(bodyParser.json())

//...

app.get('/newsletter', handlers.newsletter)
app.post('/api/newsletter-signup', handlers.api.newsletterSignup)
```

Note that we're putting our form-processing endpoint at a URL starting with `api`; this is a common technique to distinguish between user (browser) endpoints and API endpoints meant to be accessed with `fetch`.

Now we'll add those endpoints to our *lib/handlers.js* file:

```
exports.newsletter = (req, res) => {
  // we will learn about CSRF later...for now, we just
  // provide a dummy value
  res.render('newsletter', { csrf: 'CSRF token goes here' })
}
exports.api = {
  newsletterSignup: (req, res) => {
    console.log('CSRF token (from hidden form field): ' + req.body._csrf)
    console.log('Name (from visible form field): ' + req.body.name)
    console.log('Email (from visible form field): ' + req.body.email)
    res.send({ result: 'success' })
  },
}
```

We can do whatever processing we need in the form processing handler; usually we would be saving the data to the database. If there are problems, we send back a JSON object with an `err` property (instead of `result: success`).

> In this example, we're assuming all Ajax requests are looking for JSON, but there's no requirement that Ajax must use JSON for communication (as a matter of fact, Ajax used to be an acronym in which the "X" stood for XML). This approach is very JavaScript-friendly, as JavaScript is adept in handling JSON. If you're making your Ajax endpoints generally available or if you know your Ajax requests might be using something other than JSON, you should return an appropriate response *exclusively* based on the `Accepts` header, which we can conveniently access through the `req.accepts` helper method. If you're responding based only on the `Accepts` header, you might want to also look at `res.format`, which is a handy convenience method that makes it easy to respond appropriately depending on what the client expects. If you do that, you'll have to make sure to set the `dataType` or `accepts` property when making Ajax requests with JavaScript.

File Uploads

We've already mentioned that file uploads bring a raft of complications. Fortunately, there are some great projects that help make file handling a snap.

There are four popular and robust options for multipart form processing: busboy, multiparty, formidable, and multer. I have used all four, and they're all good, but I feel multiparty is the best maintained, and so we'll use it here.

Let's create a file upload form for a Meadowlark Travel vacation photo contest (*views/contest/vacation-photo.handlebars*):

```
<h2>Vacation Photo Contest</h2>

<form class="form-horizontal" role="form"
    enctype="multipart/form-data" method="POST"
    action="/contest/vacation-photo/{{year}}/{{month}}">
  <input type="hidden" name="_csrf" value="{{csrf}}">
  <div class="form-group">
    <label for="fieldName" class="col-sm-2 control-label">Name</label>
    <div class="col-sm-4">
      <input type="text" class="form-control"
      id="fieldName" name="name">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
    <div class="col-sm-4">
```

```
      <input type="email" class="form-control" required
          id="fieldEmail" name="email">
    </div>
  </div>
  <div class="form-group">
    <label for="fieldPhoto" class="col-sm-2 control-label">Vacation photo</label>
    <div class="col-sm-4">
      <input type="file" class="form-control" required  accept="image/*"
          id="fieldPhoto" name="photo">
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-2 col-sm-4">
      <button type="submit" class="btn btn-primary">Register</button>
    </div>
  </div>
</form>
```

Note that we must specify `enctype="multipart/form-data"` to enable file uploads. We're also restricting the type of files that can be uploaded by using the `accept` attribute (which is optional).

Now we need to create route handlers, but we have something of a dilemma. We want to maintain our ability to easily test our route handlers, which will be complicated by multipart form processing (in the same way we use middleware to process other types of body encoding before we even get to our handlers). Since we don't want to test multipart form decoding ourselves (we can assume this is done thoroughly by multiparty), we'll keep our handlers "pure" by passing them the already-processed information. Since we don't know what that looks like yet, we'll start with the Express plumbing in *meadowlark.js*:

```
const multiparty = require('multiparty')

app.post('/contest/vacation-photo/:year/:month', (req, res) => {
  const form = new multiparty.Form()
  form.parse(req, (err, fields, files) => {
    if(err) return res.status(500).send({ error: err.message })
    handlers.vacationPhotoContestProcess(req, res, fields, files)
  })
})
```

We're using multiparty's `parse` method to parse the request data into the data fields and the files. This method will store the files in a temporary directory on the server, and that information will be returned in the `files` array passed back.

So now we have extra information to pass to our (testable) route handler: the fields (which won't be in `req.body` as in previous examples since we're using a different body parser) and information about the file(s) that were collected. Now that we know what that looks like, we can write our route handler:

```
exports.vacationPhotoContestProcess = (req, res, fields, files) => {
  console.log('field data: ', fields)
  console.log('files: ', files)
  res.redirect(303, '/contest/vacation-photo-thank-you')
}
```

(Year and month are being specified as *route parameters*, which you'll learn about in Chapter 14.) Go ahead and run this and examine the console log. You'll see that your form fields come across as you would expect: as an object with properties corresponding to your field names. The `files` object contains more data, but it's relatively straightforward. For each file uploaded, you'll see there are properties for size, the path it was uploaded to (usually a random name in a temporary directory), and the original name of the file that the user uploaded (just the filename, not the whole path, for security and privacy reasons).

What you do with this file is now up to you: you can store it in a database, copy it to a more permanent location, or upload it to a cloud-based file storage system. Remember that if you're relying on local storage for saving files, your application won't scale well, making this a poor choice for cloud-based hosting. We will be revisiting this example in Chapter 13.

File Uploads with Fetch

Happily, using `fetch` for file uploads is nearly identical to letting the browser handle it. The hard work of file uploads is really in the encoding, which is being handled for us with middleware.

Consider this JavaScript to send our form contents using `fetch`:

```
<script>
  document.getElementById('vacationPhotoContestForm')
    .addEventListener('submit', evt => {
      evt.preventDefault()
      const body = new FormData(evt.target)
      const container =
        document.getElementById('vacationPhotoContestFormContainer')
      const url = '/api/vacation-photo-contest/{{year}}/{{month}}'
      fetch(url, { method: 'post', body })
        .then(resp => {
          if(resp.status < 200 || resp.status >= 300)
            throw new Error(`Request failed with status ${resp.status}`)
          return resp.json()
        })
        .then(json => {
          container.innerHTML = '<b>Thank you for submitting your photo!</b>'
        })
        .catch(err => {
          container.innerHTML = `<b>We're sorry, we had a problem processing ` +
            `your submission.  Please <a href="/newsletter">try again</a>`
        })
    })
  </script>
```

The important detail to note here is that we convert the form element to a FormData object, which fetch can accept directly as the request body. That's all there is to it! Because the encoding is exactly the same as it was when we let the browser handle it, our handler is almost exactly the same. We just want to return a JSON response instead of a redirect:

```
exports.api.vacationPhotoContest = (req, res, fields, files) => {
  console.log('field data: ', fields)
  console.log('files: ', files)
  res.send({ result: 'success' })
}
```

Improving File Upload UI

The browser's built-in `<input>` control for file uploads is, shall we say, a bit lacking from a UI perspective. You've probably seen drag-and-drop interfaces and file upload buttons that are styled more attractively.

The good news is that the techniques you've learned here will apply to almost all of the popular "fancy" file upload components. At the end of the day, most of them are putting a pretty face on the same form upload mechanism.

Some of the most popular file upload frontends are as follows:

- jQuery File Upload
- Uppy (this one has the benefit of offering support for many popular upload targets)
- file-upload-with-preview (this one gives you full control; you have access to an array of file objects that you can use to construct a `FormData` object to use with `fetch`)

Conclusion

In this chapter, you learned the various techniques to use for processing forms. We explored the traditional way forms are handled by browsers (letting the browser issue a `POST` request to the server with the form contents and rendering the response from the server, usually a redirect) as well as the increasingly ubiquitous approach of preventing the browser from submitting the form and handling it ourselves with `fetch`.

We learned about the common ways forms are encoded:

`application/x-www-form-urlencoded`
: Default and easy-to-use encoding typically associated with traditional form processing

`application/json`
: Common for (nonfile) data sent with `fetch`

`multipart/form-data`
: The encoding to use when you need to transfer files

Now that we've covered how to get user data into our server, let's turn our attention to *cookies* and *sessions*, which also help synchronize the server and the client.

CHAPTER 9
Cookies and Sessions

In this chapter, you'll learn how to use cookies and sessions to provide a better experience to your users by remembering their preferences from page to page, and even between browser sessions.

HTTP is a *stateless* protocol. That means that when you load a page in your browser and then you navigate to another page on the same website, neither the server nor the browser has any intrinsic way of knowing that it's the same browser visiting the same site. Another way of saying this is that the way the web works is that *every HTTP request contains all the information necessary for the server to satisfy the request*.

This is a problem, though: if the story ended there, we could never log in to anything. Streaming media wouldn't work. Websites wouldn't be able to remember your preferences from one page to the next. So there needs be a way to build state on top of HTTP, and that's where cookies and sessions enter the picture.

Cookies, unfortunately, have gotten a bad name thanks to the nefarious things that people have done with them. This is unfortunate because cookies are really quite essential to the functioning of the "modern web" (although HTML5 has introduced some new features, like local storage, that could be used for the same purpose).

The idea of a cookie is simple: the server sends a bit of information, and the browser stores it for some configurable period of time. It's really up to the server what the particular bit of information is. Often it's just a unique ID number that identifies a specific browser so that the illusion of state can be maintained.

There are some important things you need to know about cookies:

Cookies are not secret from the user
: All cookies that the server sends to the client are available for the client to look at. There's no reason you can't send something encrypted to protect its contents, but

there's seldom any need for this (at least if you're not doing anything nefarious!). *Signed* cookies, which we'll discuss in a bit, can obfuscate the contents of the cookie, but this is in no way cryptographically secure from prying eyes.

The user can delete or disallow cookies

Users have full control over cookies, and browsers make it possible to delete cookies in bulk or individually. Unless you're up to no good, there's no real reason for users to do this, but it is useful during testing. Users can also disallow cookies, which is more problematic because only the simplest web applications can make do without cookies.

Regular cookies can be tampered with

Whenever a browser makes a request of your server that has an associated cookie and you blindly trust the contents of that cookie, you are opening yourself up for attack. The height of foolishness, for example, would be to execute code contained in a cookie. To ensure cookies aren't tampered with, use signed cookies.

Cookies can be used for attacks

A category of attacks called *cross-site scripting* (XSS) attacks has sprung up in recent years. One technique of XSS attacks involves malicious JavaScript modifying the contents of cookies. This is an additional reason not to trust the contents of cookies that come back to your server. Using signed cookies helps (tampering will be evident in a signed cookie whether the user or malicious JavaScript modified it), and there's also a setting that specifies that cookies are to be modified only by the server. These cookies can be limited in usefulness, but they are certainly safer.

Users will notice if you abuse cookies

If you set a lot of cookies on your users' computers or store a lot of data, it will irritate your users, which is something you should avoid. Try to keep your use of cookies to a minimum.

Prefer sessions over cookies

For the most part, you can use *sessions* to maintain state, and it's generally wise to do so. It's easier, you don't have to worry about abusing your users' storage, and it can be more secure. Sessions rely on cookies, of course, but with sessions, Express will be doing the heavy lifting for you.

> Cookies are not magic: when the server wants the client to store a cookie, it sends a header called `Set-Cookie` containing name/value pairs, and when a client sends a request to a server for which it has cookies, it sends multiple `Cookie` request headers containing the value of the cookies.

Externalizing Credentials

To make cookies secure, a *cookie secret* is necessary. The cookie secret is a string that's known to the server and used to encrypt secure cookies before they're sent to the client. It's not a password that has to be remembered, so it can just be a random string. I usually use a random password generator inspired by xkcd to generate the cookie secret or simply a random number.

It's a common practice to externalize third-party credentials, such as the cookie secret, database passwords, and API tokens (Twitter, Facebook, etc.). This not only eases maintenance (by making it easy to locate and update credentials), but also allows you to omit the credentials file from your version control system. This is especially critical for open source repositories hosted on GitHub or other public source control repositories.

To that end, we're going to externalize our credentials in a JSON file. Create a file called *.credentials.development.json*:

```
{
  "cookieSecret": "...your cookie secret goes here"
}
```

This will be the credentials file for our development work. In this way, you could have different credentials files for production, test, or other environments, which will come in handy.

We're going to add a layer of abstraction on top of this credentials file to make it easier to manage our dependencies as our application grows. Our version will be very simple. Create a file called *config.js*:

```
const env = process.env.NODE_ENV || 'development'
const credentials = require(`./.credentials.${env}`)
module.exports = { credentials }
```

Now, to make sure we don't accidentally add credentials to our repository, add *.credentials.** to your *.gitignore* file. To import your credentials into your application, all you need to do is this:

```
const { credentials } = require('./config')
```

We'll be using this same file to store other credentials later, but for now, all we need is our cookie secret.

If you're following along by using the companion repository, you'll have to create your own credentials file, as it is not included in the repository.

Cookies in Express

Before you start setting and accessing cookies in your app, you need to include the `cookie-parser` middleware. First, use `npm install cookie-parser`, and then (*ch09/meadowlark.js* in the companion repo):

```
const cookieParser = require('cookie-parser')
app.use(cookieParser(credentials.cookieSecret))
```

Once you've done this, you can set a cookie or a signed cookie anywhere you have access to a response object:

```
res.cookie('monster', 'nom nom')
res.cookie('signed_monster', 'nom nom', { signed: true })
```

Signed cookies take precedence over unsigned cookies. If you name your signed cookie `signed_monster`, you cannot have an unsigned cookie with the same name (it will come back as `undefined`).

To retrieve the value of a cookie (if any) sent from the client, just access the `cookie` or `signedCookie` properties of the request object:

```
const monster = req.cookies.monster
const signedMonster = req.signedCookies.signed_monster
```

You can use any string you want for a cookie name. For example, we could have used `\'signed monster'` instead of `\'signed_monster'`, but then we would have to use the bracket notation to retrieve the cookie: `req.signedCookies[\'signed monster']`. For this reason, I recommend using cookie names without special characters.

To delete a cookie, use `req.clearCookie`:

```
res.clearCookie('monster')
```

When you set a cookie, you can specify the following options:

`domain`
: Controls the domains the cookie is associated with; this allows you to assign cookies to specific subdomains. Note that you cannot set a cookie for a different domain than the server is running on; it will simply do nothing.

`path`
: Controls the path this cookie applies to. Note that paths have an implicit wildcard after them; if you use a path of / (the default), it will apply to all pages on your site. If you use a path of */foo*, it will apply to the paths */foo*, */foo/bar*, etc.

`maxAge`
: Specifies how long the client should keep the cookie before deleting it, in milliseconds. If you omit this, the cookie will be deleted when you close your browser. (You can also specify a date for expiration with the `expires` option, but the syntax is frustrating. I recommend using `maxAge`.)

`secure`
: Specifies that this cookie will be sent only over a secure (HTTPS) connection.

`httpOnly`
: Setting this to `true` specifies the cookie will be modified only by the server. That is, client-side JavaScript cannot modify it. This helps prevent XSS attacks.

`signed`
: Setting this to `true` signs this cookie, making it available in `res.signedCookies` instead of `res.cookies`. Signed cookies that have been tampered with will be rejected by the server, and the cookie value will be reset to its original value.

Examining Cookies

As part of your testing, you'll probably want a way to examine the cookies on your system. Most browsers have a way to view individual cookies and the values they store. In Chrome, open the developer tools, and select the Application tab. In the tree on the left, you'll see Cookies. Expand that, and you'll see the site you're currently visiting listed. Click that, and you will see all the cookies associated with this site. You can also right-click the domain to clear all cookies or right-click an individual cookie to remove it specifically.

Sessions

Sessions are really just a more convenient way to maintain state. To implement sessions, *something* has to be stored on the client; otherwise, the server wouldn't be able to identify the client from one request to the next. The usual method of doing this is a cookie that contains a unique identifier. The server then uses that identifier to retrieve the appropriate session information.

Cookies aren't the only way to accomplish this: during the height of the "cookie scare" (when cookie abuse was rampant), many users were simply turning off cookies, and other ways to maintain state were devised, such as decorating URLs with session information. These techniques were messy, difficult, and inefficient, and they are best left in the past. HTML5 provides another option for sessions called *local storage*, which offers an advantage over cookies if you need to store larger amounts of data. See the MDN documentation for `Window.localStorage` for more information about this option.

Broadly speaking, there are two ways to implement sessions: store everything in the cookie or store only a unique identifier in the cookie and everything else on the server. The former are called *cookie-based sessions* and merely represent a convenience over using cookies. However, it still means that everything you add to the session will be stored on the client's browser, which is an approach I don't recommend. I recommend this approach only if you know that you will be storing just a small amount of information, that you don't mind the user having access to the information, and that it won't be growing out of control over time. If you want to take this approach, see the `cookie-session` middleware.

Memory Stores

If you would rather store session information on the server, which I recommend, you have to have somewhere to store it. The entry-level option is memory sessions. They are easy to set up, but they have a huge downside: when you restart the server (which you will be doing a lot of over the course of this book!), your session information disappears. Even worse, if you scale out by having multiple servers (see Chapter 12), a different server could service a request every time; session data would sometimes be there, and sometimes not. This is clearly an unacceptable user experience. However, for our development and testing needs, it will suffice. We'll see how to permanently store session information in Chapter 13.

First, install `express-session` (`npm install express-session`); then, after linking in the cookie parser, link in `express-session` (*ch09/meadowalrk.js* in the companion repo):

```
const expressSession = require('express-session')
// make sure you've linked in cookie middleware before
// session middleware!
app.use(expressSession({
    resave: false,
    saveUninitialized: false,
    secret: credentials.cookieSecret,
}))
```

The `express-session` middleware accepts a configuration object with the following options:

`resave`
: Forces the session to be saved back to the store even if the request wasn't modified. Setting this to `false` is generally preferable; see the `express-session` documentation for more information.

`saveUninitialized`
: Setting this to `true` causes new (uninitialized) sessions to be saved to the store, even if they haven't been modified. Setting this to `false` is generally preferable and is required when you need to get the user's permission before setting a cookie. See the `express-session` documentation for more information.

`secret`
: The key (or keys) used to sign the session ID cookie. This can be the same key used for `cookie-parser`.

`key`
: The name of the cookie that will store the unique session identifier. Defaults to `connect.sid`.

`store`
: An instance of a session store. Defaults to an instance of `MemoryStore`, which is fine for our current purposes. We'll see how to use a database store in Chapter 13.

`cookie`
: Cookie settings for the session cookie (`path`, `domain`, `secure`, etc.). Regular cookie defaults apply.

Using Sessions

Once you've set up sessions, using them couldn't be simpler; just use properties of the request object's `session` variable:

```
req.session.userName = 'Anonymous'
const colorScheme = req.session.colorScheme || 'dark'
```

Note that with sessions, we don't have to use the request object for retrieving the value and the response object for setting the value; it's all performed on the request object. (The response object does not have a `session` property.) To delete a session, you can use JavaScript's `delete` operator:

```
req.session.userName = null       // this sets 'userName' to null,
                                  // but doesn't remove it

delete req.session.colorScheme    // this removes 'colorScheme'
```

Using Sessions to Implement Flash Messages

Flash messages (not to be confused with Adobe Flash) are simply a way to provide feedback to users in a way that's not disruptive to their navigation. The easiest way to implement flash messages is to use sessions (you can also use the querystring, but in addition to those having uglier URLs, the flash messages will be included in a bookmark, which is probably not what you want). Let's set up our HTML first. We'll be using Bootstrap's alert messages to display our flash messages, so make sure you have Bootstrap linked in (see Bootstrap's "getting started" documentation; you can link in the Bootstrap CSS and JavaScript files in your main template—there is an example in the companion repo). In your template file, somewhere prominent (usually directly below your site's header), add the following:

```
{{#if flash}}
  <div class="alert alert-dismissible alert-{{flash.type}}">
    <button type="button" class="close"
      data-dismiss="alert" aria-hidden="true">&times;</button>
    <strong>{{flash.intro}}</strong> {{{flash.message}}}
  </div>
{{/if}}
```

Note that we use three curly brackets for `flash.message`; this will allow us to provide some simple HTML in our messages (we might want to emphasize words or include hyperlinks). Now let's add some middleware to add the `flash` object to the context if there's one in the session. After we've displayed a flash message once, we want to remove it from the session so it isn't displayed on the next request. We'll create some middleware to check the session to see whether there's a flash message and, if there is, transfer it to the `res.locals` object, making it available to the views. We'll put our middleware in a file called *lib/middleware/flash.js*:

```
module.exports = (req, res, next) => {
  // if there's a flash message, transfer
  // it to the context, then clear it
  res.locals.flash = req.session.flash
  delete req.session.flash
  next()
})
```

And in our *meadowalrk.js* file, we'll link in the flash message middleware, before any of our view routes:

```
const flashMiddleware = require('./lib/middleware/flash')
app.use(flashMiddleware)
```

Now let's see how to actually use the flash message. Imagine we're signing up users for a newsletter and we want to redirect them to the newsletter archive after they sign up. This is what our form handler might look like:

```
// slightly modified version of the official W3C HTML5 email regex:
// https://html.spec.whatwg.org/multipage/forms.html#valid-e-mail-address
const VALID_EMAIL_REGEX = new RegExp('^[a-zA-Z0-9.!#$%&\'*+\/=?^_`{|}~-]+@' +
 '[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?' +
 '(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)+$')

app.post('/newsletter', function(req, res){
  const name = req.body.name || '', email = req.body.email || ''
  // input validation
  if(VALID_EMAIL_REGEX.test(email)) {
   req.session.flash = {
    type: 'danger',
    intro: 'Validation error!',
    message: 'The email address you entered was not valid.',
   }
   return res.redirect(303, '/newsletter')
  }
  // NewsletterSignup is an example of an object you might create; since
  // every implementation will vary, it is up to you to write these
  // project-specific interfaces. This simply shows how a typical
  // Express implementation might look in your project.
  new NewsletterSignup({ name, email }).save((err) => {
    if(err) {
     req.session.flash = {
      type: 'danger',
      intro: 'Database error!',
      message: 'There was a database error; please try again later.',
     }
     return res.redirect(303, '/newsletter/archive')
    }
    req.session.flash = {
     type: 'success',
     intro: 'Thank you!',
     message: 'You have now been signed up for the newsletter.',
    };
    return res.redirect(303, '/newsletter/archive')
  })
})
```

Note that we're careful to distinguish between input validation and database errors. Remember that even if we do input validation on the frontend (and you should), you should also perform it on the backend, because malicious users can circumvent front-end validation.

Flash messages are a great mechanism to have available in your website, even if other methods are more appropriate in certain areas (for example, flash messages aren't always appropriate for multiform "wizards" or shopping cart checkout flows). Flash messages are also great during development, because they are an easy way to provide feedback, even if you replace them with a different technique later. Adding support for flash messages is one of the first things I do when setting up a website, and we'll be using this technique throughout the rest of the book.

Because the flash message is being transferred from the session to `res.locals.flash` in middleware, you have to perform a redirect for the flash message to be displayed. If you want to display a flash message without redirecting, set `res.locals.flash` instead of `req.session.flash`.

The example in this chapter used browser form submission with redirects because the use of sessions to control UI like this is typically not used in applications that use Ajax for form submission. In that event, you would want to indicate any errors in the JSON returned from the form handler and have the frontend modify the DOM to dynamically display error messages. That's not to say that sessions aren't useful for frontend rendered applications, but they are seldom used for this purpose.

What to Use Sessions For

Sessions are useful whenever you want to save a user preference that applies across pages. Most commonly, sessions are used to provide user authentication information: you log in, and a session is created. After that, you don't have to log in again every time you reload the page. Sessions can be useful even without user accounts, though. It's quite common for sites to remember how you like things sorted or what date format you prefer—all without your having to log in.

While I encourage you to prefer sessions over cookies, it's important to understand how cookies work (especially because they enable sessions to work). It will help you with diagnosing issues and understanding the security and privacy considerations of your application.

Conclusion

Understanding cookies and sessions gives us a better understanding of how web applications maintain the illusion of state when the underlying protocol (HTTP) is stateless. We learned techniques for handling cookies and sessions to control the user's experience.

We've also been writing middleware as we went along without too much explanation of middleware. In the next chapter, we're going to dive into middleware and learn everything there is to know about it!

CHAPTER 10
Middleware

By now, we've already had some exposure to middleware: we've used existing middleware (`body-parser`, `cookie-parser`, `static`, and `express-session`, to name a few), and we've even written some of our own (for adding weather data to our template context, configuring flash messages, and our 404 handler). But what is middleware, exactly?

Conceptually, *middleware* is a way to encapsulate functionality—specifically, functionality that operates on an HTTP request to your application. Practically, middleware is simply a function that takes three arguments: a request object, a response object, and a `next()` function, which will be explained shortly. (There is also a form that takes four arguments, for error handling, which will be covered at the end of this chapter.)

Middleware is executed in what's known as a *pipeline*. You can imagine a physical pipe, carrying water. The water gets pumped in at one end, and then there are gauges and valves before the water gets where it's going. The important part about this analogy is that *order matters*; if you put a pressure gauge before a valve, it has a different effect than if you put the pressure gauge after the valve. Similarly, if you have a valve that injects something into the water, everything "downstream" from that valve will contain the added ingredient. In an Express app, you insert middleware into the pipeline by calling `app.use`.

Prior to Express 4.0, the pipeline was complicated by your having to link in the *router* explicitly. Depending on where you linked in the router, routes could be linked in out of order, making the pipeline sequence less clear when you mix middleware and route handlers. In Express 4.0, middleware and route handlers are invoked in the order in which they were linked in, making the sequence much clearer.

It's common practice to have the last middleware in your pipeline be a catchall handler for any request that doesn't match any other routes. This middleware usually returns a status code of 404 (Not Found).

So how is a request "terminated" in the pipeline? That's what the `next` function passed to each middleware does: if you *don't* call `next()`, the request terminates with that middleware.

Middleware Principles

Learning how to think flexibly about middleware and route handlers is key to understanding how Express works. Here are the things you should keep in mind:

- Route handlers (`app.get`, `app.post`, etc.—often referred to collectively as `app.METHOD`) can be thought of as middleware that handles only a specific HTTP verb (`GET`, `POST`, etc.). Conversely, middleware can be thought of as a route handler that handles all HTTP verbs (this is essentially equivalent to `app.all`, which handles any HTTP verb; there are some minor differences with exotic verbs such as `PURGE`, but for the common verbs, the effect is the same).
- Route handlers require a path as their first parameter. If you want that path to match any route, simply use `\*`. Middleware can also take a path as its first parameter, but it is optional (if it is omitted, it will match any path, as if you had specified `*`).
- Route handlers and middleware take a callback function that takes two, three, or four parameters (technically, you could also have zero or one parameters, but there is no sensible use for these forms). If there are two or three parameters, the first two parameters are the request and response objects, and the third parameter is the `next` function. If there are four parameters, it becomes *error-handling* middleware, and the first parameter becomes an error object, followed by the request, response, and next objects.
- If you *don't* call `next()`, the pipeline will be terminated, and no more route handlers or middleware will be processed. If you don't call `next()`, you should send a response to the client (`res.send`, `res.json`, `res.render`, etc.); if you don't, the client will hang and eventually time out.
- If you *do* call `next()`, it's generally inadvisable to send a response to the client. If you do, middleware or route handlers further down the pipeline will be executed, but any client responses they send will be ignored.

Middleware Examples

If you want to see this in action, let's try some really simple middleware (*ch10/00-simple-middleware.js* in the companion repo):

```
app.use((req, res, next) => {
  console.log(`processing request for ${req.url}....`)
  next()
})

app.use((req, res, next) => {
  console.log('terminating request')
  res.send('thanks for playing!')
  // note that we do NOT call next() here...this terminates the request
})

app.use((req, res, next) => {
  console.log(`whoops, i'll never get called!`)
})
```

Here we have three examples of middleware. The first one simply logs a message to the console before passing on the request to the next middleware in the pipeline by calling `next()`. Then the next middleware actually handles the request. Note that if we omitted the `res.send` here, no response would ever be returned to the client. Eventually, the client would time out. The last middleware will never execute, because all requests are terminated in the prior middleware.

Now let's consider a more complicated, complete example (*ch10/01-routing-example.js* in the companion repo):

```
const express = require('express')
const app = express()

app.use((req, res, next) => {
  console.log('\n\nALLWAYS')
  next()
})

app.get('/a', (req, res) => {
  console.log('/a: route terminated')
  res.send('a')
})
app.get('/a', (req, res) => {
  console.log('/a: never called');
})
app.get('/b', (req, res, next) => {
  console.log('/b: route not terminated')
  next()
})
app.use((req, res, next) => {
  console.log('SOMETIMES')
  next()
})
app.get('/b', (req, res, next) => {
  console.log('/b (part 2): error thrown' )
  throw new Error('b failed')
})
app.use('/b', (err, req, res, next) => {
  console.log('/b error detected and passed on')
  next(err)
})
app.get('/c', (err, req) => {
  console.log('/c: error thrown')
  throw new Error('c failed')
})
app.use('/c', (err, req, res, next) => {
  console.log('/c: error detected but not passed on')
  next()
})

app.use((err, req, res, next) => {
  console.log('unhandled error detected: ' + err.message)
  res.send('500 - server error')
})

app.use((req, res) => {
  console.log('route not handled')
  res.send('404 - not found')
})

const port = process.env.PORT || 3000
app.listen(port, () => console.log( `Express started on http://localhost:${port}` +
  '; press Ctrl-C to terminate.'))
```

Before trying this example, imagine what the result will be. What are the different routes? What will the client see? What will be printed on the console? If you can correctly answer all of those questions, you've got the hang of routes in Express! Pay particular attention to the difference between a request to */b* and a request to */c*; in both instances, there was an error, but one results in a 404, and the other results in a 500.

Note that middleware *must* be a function. Keep in mind that in JavaScript, it's quite easy (and common) to return a function from a function. For example, you'll note that `express.static` is a function, but we actually invoke it, so it must return another function. Consider the following:

```
app.use(express.static)         // this will NOT work as expected

console.log(express.static())   // will log "function", indicating
                                // that express.static is a function
                                // that itself returns a function
```

Note also that a module can export a function, which can in turn be used directly as middleware. For example, here's a module called *lib/tourRequiresWaiver.js* (Meadowlark Travel's rock-climbing packages require a liability waiver):

```
module.exports = (req,res,next) => {
  const { cart } = req.session
  if(!cart) return next()
  if(cart.items.some(item => item.product.requiresWaiver)) {
    cart.warnings.push('One or more of your selected ' +
      'tours requires a waiver.')
  }
  next()
}
```

We could link this middleware in like so (*ch10/02-item-waiver.example.js* in the companion repo):

```
const requiresWaiver = require('./lib/tourRequiresWaiver')
app.use(requiresWaiver)
```

More commonly, though, you would export an object that contains properties that are middleware. For example, let's put all of our shopping cart validation code in *lib/cartValidation.js*:

```
module.exports = {

  resetValidation(req, res, next) {
    const { cart } = req.session
    if(cart) cart.warnings = cart.errors = []
    next()
  },

  checkWaivers(req, res, next) {
    const { cart } = req.session
    if(!cart) return next()
    if(cart.items.some(item => item.product.requiresWaiver)) {
      cart.warnings.push('One or more of your selected ' +
        'tours requires a waiver.')
    }
    next()
  },

  checkGuestCounts(req, res, next) {
    const { cart } = req.session
    if(!cart) return next()
    if(cart.items.some(item => item.guests > item.product.maxGuests )) {
      cart.errors.push('One or more of your selected tours ' +
        'cannot accommodate the number of guests you ' +
        'have selected.')
    }
    next()
  },
}
```

Then you could link the middleware in like this (*ch10/03-more-cart-validation.js* in the companion repo):

```
const cartValidation = require('./lib/cartValidation')

app.use(cartValidation.resetValidation)
app.use(cartValidation.checkWaivers)
app.use(cartValidation.checkGuestCounts)
```

> In the previous example, we have middleware aborting early with the statement `return next()`. Express doesn't expect middleware to return a value (and it doesn't do anything with any return values), so this is just a shortened way of writing `next(); return`.

Common Middleware

While there are thousands of middleware projects on npm, there are a handful that are common and fundamental, and at least some of these will be found in every nontrivial Express project. Some of this middleware was so common that it was actually bundled with Express, but it has long since been moved into individual packages. The only middleware still bundled with Express itself is `static`.

This list attempts to cover the most common middleware:

`basicauth-middleware`
: Provides basic access authorization. Keep in mind that basic auth offers only the most basic security, and you should use basic auth *only* over HTTPS (otherwise, usernames and passwords are transmitted in the clear). You should use basic auth only when you need something quick and easy *and* you're using HTTPS.

`body-parser`
: Provides parsing for HTTP request bodies. Provides middleware for parsing both URL-encoded and JSON-encoded bodies, as well as others.

`busboy`, `multiparty`, `formidable`, `multer`
: All of these middleware options parse request bodies encoded with `multipart/form-data`.

`compression`
: Compresses response data with gzip or deflate. This is a good thing, and your users will thank you, especially those on slow or mobile connections. It should be linked in early, before any middleware that might send a response. The only thing that I recommend linking in before `compress` is debugging or logging middle-

ware (which do not send responses). Note that in most production environments, compression is handled by a proxy like NGINX, making this middleware unnecessary.

`cookie-parser`
: Provides cookie support. See Chapter 9.

`cookie-session`
: Provides cookie-storage session support. I do not generally recommend this approach to sessions. It must be linked in after `cookie-parser`. See Chapter 9.

`express-session`
: Provides session ID (stored in a cookie) session support. Defaults to a memory store, which is not suitable for production and can be configured to use a database store. See Chapter 9 and Chapter 13.

`csurf`
: Provides protection against cross-site request forgery (CSRF) attacks. This uses sessions, so it must be linked in after `express-session` middleware. Unfortunately, simply linking in this middleware does not magically protect against CSRF attacks; see Chapter 18 for more information.

`serve-index`
: Provides directory listing support for static files. There is no need to include this middleware unless you specifically need directory listing.

`errorhandler`
: Provides stack traces and error messages to the client. I do not recommend linking this in on a production server, as it exposes implementation details, which can have security or privacy consequences. See Chapter 20 for more information.

`serve-favicon`
: Serves the *favicon* (the icon that appears in the title bar of your browser). This is not strictly necessary; you can simply put a *favicon.ico* in the root of your static directory, but this middleware can improve performance. If you use it, it should be linked in high in the middleware stack. It also allows you to designate a filename other than *favicon.ico*.

`morgan`
: Provides automated logging support; all requests will be logged. See Chapter 20 for more information.

`method-override`
: Provides support for the `x-http-method-override` request header, which allows browsers to "fake" using HTTP methods other than `GET` and `POST`. This can be useful for debugging. This is needed only if you're writing APIs.

`response-time`
: Adds the `X-Response-Time` header to the response, providing the response time in milliseconds. You usually don't need this middleware unless you are doing performance tuning.

`static`
: Provides support for serving static (public) files. You can link in this middleware multiple times, specifying different directories. See Chapter 17 for more details.

`vhost`
: Virtual hosts (vhosts), a term borrowed from Apache, makes subdomains easier to manage in Express. See Chapter 14 for more information.

Third-Party Middleware

Currently, there is no comprehensive "store" or index for third-party middleware. Almost all Express middleware, however, will be available on npm, so if you search npm for "Express" and "middleware," you'll get a pretty good list. The official Express documentation also contains a useful list of middleware.

Conclusion

In this chapter, we delved into what middleware is, how to write our own, and how it's processed as part of an Express application. If you're starting to think that an Express application is simply a collection of middleware, you're starting to understand Express! Even the route handlers we've been using heretofore are just specialized cases of middleware.

In the next chapter, we'll be looking at another common infrastructure need: sending email (and you had better believe there is going to be some middleware involved!).

CHAPTER 11

Sending Email

One of the primary ways your application can communicate with the world is email. From user registration to password reset instructions to promotional emails, the ability to send email is an important feature. In this chapter, you'll learn how to format and send email with Node and Express to help communicate with your users.

Neither Node nor Express has any built-in way of sending email, so we have to use a third-party module. The package I recommend is Andris Reinman's excellent *Nodemailer*. Before we dive into configuring Nodemailer, let's get some email basics out of the way.

SMTP, MSAs, and MTAs

The lingua franca for sending email is the Simple Mail Transfer Protocol (SMTP). While it is possible to use SMTP to send an email directly to the recipient's mail server, this is generally a bad idea: unless you are a "trusted sender" like Google or Yahoo!, chances are your email will be tossed directly into the spam bin. It's better to use a mail submission agent (MSA), which will deliver the email through trusted channels, reducing the chance that your email will be marked as spam. In addition to ensuring that your email arrives, MSAs handle nuisances like temporary outages and bounced emails. The final piece of the equation is the mail transfer agent (MTA), which is the service that actually sends the email to its final destination. For the purposes of this book, *MSA*, *MTA*, and *SMTP server* are essentially equivalent.

So you'll need access to an MSA. While it is possible to get started using a free consumer email service such as Gmail, Outlook, or Yahoo!, these services are no longer as friendly to automated emails as they once were (in an effort to cut down on abuse). Fortunately, there are a couple of excellent email services to choose from that have a free option for low-volume use: Sendgrid and Mailgun. I've used both services, and I like them both. The examples in this book will be using SendGrid.

If you're working for an organization, the organization itself may have an MSA; you can contact your IT department and ask them if there's an SMTP relay available for sending automated emails.

If you're using SendGrid or Mailgun, go ahead and set up your account now. For SendGrid, you'll need to create an API key (which will be your SMTP password).

Receiving Email

Most websites only need the ability to *send* email, like password reset instructions and promotional emails. However, some applications need to receive email as well. A good example is an issue-tracking system that sends out an email when someone updates an issue, and if you reply to that email, the issue is automatically updated with your response.

Unfortunately, receiving email is much more involved and will not be covered in this book. If this is functionality you need, you should allow your mail provider to maintain the mailbox and have a periodic process to access it with an IMAP agent such as imap-simple.

Email Headers

An email message consists of two parts: the header and the body (very much like an HTTP request). The *header* contains information about the email: who it's from, who it's addressed to, the date it was received, the subject, and more. Those are the headers that are normally displayed to the user in an email application, but there are many more headers. Most email clients allow you to look at the headers; if you've never done so, I recommend you take a look. The headers give you all the information about how the email got to you; every server and MTA that the email passed through will be listed in the header.

It often comes as a surprise to people that some headers, like the "from" address, can be set arbitrarily by the sender. When you specify a "from" address other than the account from which you're sending, it's often referred to as *spoofing*. There is nothing preventing you from sending an email with the "from" address Bill Gates <*billg@microsoft.com*>. I'm not recommending that you try this, just driving home the point that you can set certain headers to be whatever you want. Sometimes there are legitimate reasons to do this, but you should never abuse it.

An email you send *must* have a "from" address, however. This can sometimes cause problems when sending automated email, which is why you often see email with a return addresses like DO NOT REPLY <*do-not-reply@meadowlarktravel.com*>.

Whether you want to take this approach or have automated emails come from an address like Meadowlark Travel <*info@meadowlarktravel.com*> is up to you; if you take the latter approach, though, you should be prepared to respond to emails that come to *info@meadowlarktravel.com*.

Email Formats

When the internet was new, all email was simply ASCII text. The world has changed a lot since then, and people want to send email in different languages and do more sophisticated things like include formatted text, images, and attachments. This is where things start to get ugly: email formats and encoding are a horrible jumble of techniques and standards.

Fortunately, we won't really have to address these complexities. Nodemailer will handle that for us. What's important for you to know is that your email can be either plain text (Unicode) or HTML.

Almost all modern email applications support HTML email, so it's generally pretty safe to format your emails in HTML. Still, there are "text purists" out there who eschew HTML email, so I recommend always including both text and HTML email. If you don't want to have to write text and HTML email, Nodemailer supports a shortcut that will automatically generate the plain text version from the HTML.

HTML Email

HTML email is a topic that could fill an entire book. Unfortunately, it's not as simple as just writing HTML as you would for your site: most mail clients support only a small subset of HTML. Mostly, you have to write HTML as if it were still 1996; it's not much fun. In particular, you have to go back to using tables for layout (cue sad music).

If you have experience with browser compatibility issues with HTML, you know what a headache it can be. Email compatibility issues are much worse. Fortunately, there are some things that can help.

First, I encourage you to read MailChimp's excellent article about writing HTML email. It does a good job covering the basics and explaining the things you need to keep in mind when writing HTML email.

The next is a real time-saver: HTML Email Boilerplate. It's essentially a very well-written, rigorously tested template for HTML email.

Finally, there's testing. You've read up on how to write HTML email, and you're using HTML Email Boilerplate, but testing is the only way to know for sure your email is not going to explode on Lotus Notes 7 (yes, people still use it). Feel like installing 30 different mail clients to test one email? I didn't think so. Fortunately, there's a great service that does it for you: Litmus. It's not an inexpensive service; plans start at about $100 a month. But if you send a lot of promotional emails, it's hard to beat.

On the other hand, if your formatting is modest, there's no need for an expensive testing service like Litmus. If you're sticking to things like headers, bold/italic text, horizontal rules, and some image links, you're pretty safe.

Nodemailer

First, we need to install the Nodemailer package:

```
npm install nodemailer
```

Then, require the `nodemailer` package and create a Nodemailer instance (a *transport* in Nodemailer parlance):

```
const nodemailer = require('nodemailer')

const mailTransport = nodemailer.createTransport({

  auth: {
    user: credentials.sendgrid.user,
    pass: credentials.sendgrid.password,
  }
})
```

Notice we're using the credentials module we set up in Chapter 9. You'll need to update your *.credentials.development.json* file accordingly:

```
{
  "cookieSecret": "your cookie secret goes here",
  "sendgrid": {
    "user": "your sendgrid username",
    "password": "your sendgrid password"
  }
}
```

Common configuration options for SMTP are the port, authentication type, and TLS options. However, most of the major mail services use the default options. To find out what settings to use, consult your mail service documentation (try searching for *sending SMTP email* or *SMTP configuration* or *SMTP relay*). If you're having trouble sending SMTP email, you may need to check the options; see the Nodemailer documentation for a complete list of supported options.

If you're following along with the companion repo, you'll find that there aren't any settings in the credentials file. In the past, I have had many readers contact me asking why the file is missing or empty. I intentionally don't provide valid credentials for the same reason you should be careful with your credentials! I trust you very much, dear reader, but not so much that I'm going to give you my email password!

Sending Mail

Now that we have our mail transport instance, we can send mail. We'll start with a simple example that sends text mail to only one recipient (*ch11/00-smtp.js* in the companion repo):

```
try {
  const result = await mailTransport.sendMail({
    from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
    to: 'joecustomer@gmail.com',
    subject: 'Your Meadowlark Travel Tour',
    text: 'Thank you for booking your trip with Meadowlark Travel.  ' +
      'We look forward to your visit!',
  })
  console.log('mail sent successfully: ', result)
} catch(err) {
  console.log('could not send mail: ' + err.message)
}
```

In the code samples in this section, I'm using fake email addresses like *joecustomer@gmail.com*. For verification purposes, you'll probably want to change those email addresses to an email you control so you can see what's happening. Otherwise, poor *joecustomer@gmail.com* is going to be getting a lot of nonsense email!

You'll notice that we're handling errors here, but it's important to understand that no errors doesn't necessarily mean your email was delivered successfully to the *recipient*. The callback's `error` parameter will be set only if there was a problem communicating with the MSA (such as a network or authentication error). If the MSA was unable to deliver the email (for example, because of an invalid email address or an unknown user), you will have to check your account activity in your mail service, which you can do either from the admin interface or through an API.

If you need your system to automatically determine whether the email was delivered successfully, you'll have to use your mail service's API. Consult the API documentation for your mail service for more information.

Sending Mail to Multiple Recipients

Nodemail supports sending mail to multiple recipients by using commas (*ch11/01-multiple-recipients.js* in the companion repo):

```
try {
  const result = await mailTransport.sendMail({
    from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
    to: 'joe@gmail.com, "Jane Customer" <jane@yahoo.com>, ' +
      'fred@hotmail.com',
    subject: 'Your Meadowlark Travel Tour',
    text: 'Thank you for booking your trip with Meadowlark Travel.  ' +
      'We look forward to your visit!',
  })
  console.log('mail sent successfully: ', result)
} catch(err) {
  console.log('could not send mail: ' + err.message)
}
```

Note that, in this example, we mixed plain email addresses (*joe@gmail.com*) with email addresses specifying the recipient's name ("Jane Customer" <*jane@yahoo.com*>). This is allowed syntax.

When sending email to multiple recipients, you must be careful to observe the limits of your MSA. SendGrid, for example, recommends limiting the number of recipients (SendGrid recommends no more than a thousand in one email). If you're sending bulk email, you probably want to deliver multiple messages, each with multiple recipients (*ch11/02-many-recipients.js* in the companion repo):

```
// largeRecipientList is an array of email addresses
const recipientLimit = 100
const batches = largeRecipientList.reduce((batches, r) => {
  const lastBatch = batches[batches.length - 1]
  if(lastBatch.length < recipientLimit)
    lastBatch.push(r)
  else
    batches.push([r])
  return batches
}, [[]])
try {
  const results = await Promise.all(batches.map(batch =>
    mailTransport.sendMail({
      from: '"Meadowlark Travel", <info@meadowlarktravel.com>',
      to: batch.join(', '),
      subject: 'Special price on Hood River travel package!',
      text: 'Book your trip to scenic Hood River now!',
    })
  ))
  console.log(results)
} catch(err) {
  console.log('at least one email batch failed: ' + err.message)
}
```

Better Options for Bulk Email

While you can certainly send bulk email with Nodemailer and an appropriate MSA, you should think carefully before going this route. A responsible email campaign must provide a way for people to unsubscribe from your promotional emails, and that is not a trivial task. Multiply that by every subscription list you maintain (perhaps you have a weekly newsletter and a special announcements campaign, for example). This is an area in which it's best not to reinvent the wheel. Services like Emma, Mailchimp, and Campaign Monitor offer everything you need, including great tools for monitoring the success of your email campaigns. They're very affordable, and I highly recommend using them for promotional emails, newsletters, etc.

Sending HTML Email

So far, we've just been sending plain-text email, but most people these days expect something a little prettier. Nodemailer allows you to send both HTML and plaintext versions in the same email, allowing the email client to choose which version is displayed (usually HTML) (*ch11/03-html-email.js* in the companion repo):

```
const result = await mailTransport.sendMail({
  from: '"Meadowlark Travel" <info@meadowlarktravel.com>',
  to: 'joe@gmail.com, "Jane Customer" <jane@yahoo.com>, ' +
    'fred@hotmail.com',
  subject: 'Your Meadowlark Travel Tour',
  html: '<h1>Meadowlark Travel</h1>\n<p>Thanks for book your trip with ' +
    'Meadowlark Travel.  <b>We look forward to your visit!</b>',
  text: 'Thank you for booking your trip with Meadowlark Travel.  ' +
    'We look forward to your visit!',
})
```

Providing both HTML and text versions is a lot of work, especially if very few of your users prefer text-only email. If you want to save some time, you can write your emails in HTML and use a package like html-to-formatted-text to automatically generate text from your HTML. (Just keep in mind that it won't be as high quality as hand-crafted text; HTML won't always translate cleanly.)

Images in HTML Email

While it is possible to embed images in HTML email, I strongly discourage it. They bloat your email messages, and it isn't generally considered good practice. Instead, you should make images you want to use in email available on your web server and link appropriately from the email.

It is best to have a dedicated location in your static assets folder for email images. You should even keep assets that you use both on your site and in emails separate. It reduces the chance of negatively affecting the layout of your emails.

Let's add some email resources in our Meadowlark Travel project. In your *public* directory, create a subdirectory called *email*. You can place your *logo.png* in there, and any other images you want to use in your email. Then, in your email, you can use those images directly:

```
<img src="//meadowlarktravel.com/email/logo.png"
  alt="Meadowlark Travel Logo">
```

It should be obvious that you do not want to use *localhost* when sending out email to other people; they probably won't even have a server running, much less on port 3000! Depending on your mail client, you might be able to use *localhost* in your email for testing purposes, but it won't work outside of your computer. In Chapter 17, we'll discuss some techniques to smooth the transition from development to production.

Using Views to Send HTML Email

So far, we've been putting our HTML in strings in JavaScript, a practice you should try to avoid. Our HTML has been simple enough, but take a look at HTML Email Boilerplate: do you want to put all that boilerplate in a string? Absolutely not.

Fortunately, we can leverage views to handle this. Let's consider our "Thank you for booking your trip with Meadowlark Travel" email example, which we'll expand a little bit. Let's imagine that we have a shopping cart object that contains our order information. That shopping cart object will be stored in the session. Let's say the last step in our ordering process is a form that's processed by */cart/checkout*, which sends a confirmation email. Let's start by creating a view for the thank-you page, *views/cart-thank-you.handlebars*:

```
<p>Thank you for booking your trip with Meadowlark Travel,
  {{cart.billing.name}}!</p>
<p>Your reservation number is {{cart.number}}, and an email has been
sent to {{cart.billing.email}} for your records.</p>
```

Then we'll create an email template for the email. Download HTML Email Boilerplate, and put in *views/email/cart-thank-you.handlebars*. Edit the file, and modify the body:

```
<table cellpadding="0" cellspacing="0" border="0" id="backgroundTable">
  <tr>
    <td valign="top">
      <table cellpadding="0" cellspacing="0" border="0" align="center">
        <tr>
          <td width="200" valign="top"><img class="image_fix"
            src="//placehold.it/100x100"
            alt="Meadowlark Travel" title="Meadowlark Travel"
            width="180" height="220" /></td>
        </tr>
        <tr>
          <td width="200" valign="top"><p>
            Thank you for booking your trip with Meadowlark Travel,
            {{cart.billing.name}}.</p><p>Your reservation number
            is {{cart.number}}.</p></td>
        </tr>
        <tr>
          <td width="200" valign="top">Problems with your reservation?
          Contact Meadowlark Travel at
          <span class="mobile_link">555-555-0123</span>.</td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```

Because you can't use *localhost* addresses in email, if your site isn't live yet, you can use a placeholder service for any graphics. For example, *http://placehold.it/100x100* dynamically serves a 100-pixel-square graphic you can use. This technique is used quite often for for-placement-only (FPO) images and layout purposes.

Now we can create a route for our cart Thank-you page (*ch11/04-rendering-html-email.js* in the companion repo):

```
app.post('/cart/checkout', (req, res, next) => {
 const cart = req.session.cart
 if(!cart) next(new Error('Cart does not exist.'))
 const name = req.body.name || '', email = req.body.email || ''
 // input validation
 if(!email.match(VALID_EMAIL_REGEX))
  return res.next(new Error('Invalid email address.'))
 // assign a random cart ID; normally we would use a database ID here
 cart.number = Math.random().toString().replace(/^0\.0*/, '')
 cart.billing = {
  name: name,
  email: email,
 }
 res.render('email/cart-thank-you', { layout: null, cart: cart },
  (err,html) => {
    console.log('rendered email: ', html)
    if(err) console.log('error in email template')
    mailTransport.sendMail({
     from: '"Meadowlark Travel": info@meadowlarktravel.com',
     to: cart.billing.email,
     subject: 'Thank You for Book your Trip with Meadowlark Travel',
     html: html,
     text: htmlToFormattedText(html),
    })
     .then(info => {
```

```
              console.log('sent! ', info)
              res.render('cart-thank-you', { cart: cart })
            })
            .catch(err => {
              console.error('Unable to send confirmation: ' + err.message)
            })
        }
      )
    })
```

Note that we're calling `res.render` twice. Normally, you call it only once (calling it twice will display only the results of the first call). However, in this instance, we're circumventing the normal rendering process the first time we call it: notice that we provide a callback. Doing that prevents the results of the view from being rendered to the browser. Instead, the callback receives the rendered view in the parameter `html`: all we have to do is take that rendered HTML and send the email! We specify `layout: null` to prevent our layout file from being used, because it's all in the email template (an alternate approach would be to create a separate layout file for emails and use that instead). Lastly, we call `res.render` again. This time, the results will be rendered to the HTML response as normal.

Encapsulating Email Functionality

If you're using email a lot throughout your site, you may want to encapsulate the email functionality. Let's assume you always want your site to send email from the same sender ("Meadowlark Travel" <*info@meadowlarktravel.com*>) and you always want the email to be sent in HTML with automatically generated text. Create a module called *lib/email.js* (*ch11/lib/email.js* in the companion repo):

```
const nodemailer = require('nodemailer')
const htmlToFormattedText = require('html-to-formatted-text')

module.exports = credentials => {

  const mailTransport = nodemailer.createTransport({
    host: 'smtp.sendgrid.net',
    auth: {
      user: credentials.sendgrid.user,
      pass: credentials.sendgrid.password,
    },
  })

  const from = '"Meadowlark Travel" <info@meadowlarktravel.com>'
  const errorRecipient = 'youremail@gmail.com'

  return {
    send: (to, subject, html) =>
      mailTransport.sendMail({
        from,
        to,
        subject,
        html,
        text: htmlToFormattedText(html),
      }),
  }

}
```

Now all we have to do to send an email is the following (*ch11/05-email-library.js* in the companion repo):

```
const emailService = require('./lib/email')(credentials)

emailService.send(email, "Hood River tours on sale today!",
  "Get 'em while they're hot!")
```

Conclusion

In this chapter, you learned the basics of how email is delivered on the internet. If you were following along, you set up a free email service (most likely SendGrid or Mailgun) and used the service to send text and HTML email. You learned how we can use the same template rendering mechanism we use for rendering HTML in our Express applications to render HTML for email.

Email remains an important way your application can communicate to your users. Be mindful not to abuse this power! If you're like me, you have an inbox overflowing with automated email that you mostly ignore. When it comes to automated emails, less is more. There are legitimate and useful reasons your application could send an email to your users, but you should always ask yourself, "Do my users *really* want this email? Is there another way to communicate this information?"

Now that we've covered some basic infrastructure that we'll need to create applications, we'll spend some time talking about the eventual production launch of our application and the kinds of things we'll want to consider to make that launch successful.

CHAPTER 12

Production Concerns

While it may feel premature to start discussing production concerns at this point, you can save yourself a lot of time and suffering down the line if you start thinking about production early on. Launch day will be here before you know it.

In this chapter, you'll learn about Express's support for different execution environments, methods to scale your website, and how to monitor your website's health. You'll see how you can simulate a production environment for testing and development and also how to perform stress testing so you can identify production problems before they happen.

Execution Environments

Express supports the concept of *execution environments*: a way to run your application in production, development, or test mode. You could actually have as many different environments as you want. For example, you could have a staging environment, or a training environment. However, keep in mind that development, production, and test are "standard" environments, and both Express and third-party middleware often make decisions based on those environments. In other words, if you have a "staging" environment, there's no way to make it automatically inherit the properties of a production environment. For this reason, I recommend you stick with the standards of production, development, and test.

While it is possible to specify the execution environment by calling `app.set('env', \'production')`, it is inadvisable to do so; it means your app will always run in that environment, no matter what the situation. Worse, it may start running in one environment and then switch to another.

It’s preferable to specify the execution environment by using the environment variable NODE_ENV. Let’s modify our app to report on the mode it’s running in by calling app.get('env’):

```
const port = process.env.PORT || 3000
app.listen(port, () => console.log(`Express started in ` +
  `${app.get('env')} mode at http://localhost:${port}` +
  `; press Ctrl-C to terminate.`))
```

If you start your server now, you’ll see you’re running in development mode; it’s the default if you don’t specify otherwise. Let’s try putting it in production mode:

```
$ export NODE_ENV=production
$ node meadowlark.js
```

If you’re using Unix/BSD, there’s a handy syntax that allows you to modify the environment only for the duration of that command:

```
$ NODE_ENV=production node meadowlark.js
```

This will run the server in production mode, but once the server terminates, the NODE_ENV environment variable won’t be modified. I’m particularly fond of this shortcut, and it reduces the chance that I accidentally leave environment variables set to values that I don’t necessarily want for everything.

> If you start Express in production mode, you may notice warnings about components that are not suitable for use in production mode. If you’ve been following along with the examples in this book, you’ll see that connect.session is using a memory store, which is not suitable for a production environment. Once we switch to a database store in Chapter 13, this warning will disappear.

Environment-Specific Configuration

Just changing the execution environment won’t do much, though Express will log more warnings to the console in production mode (for example, informing you of modules that are deprecated and will be removed in the future). Also, in production mode, view caching is enabled by default (see Chapter 7).

Mainly, the execution environment is a tool for you to leverage, allowing you to easily make decisions about how your application should behave in the different environments. As a word of caution, you should try to minimize the differences between your development, test, and production environments. That is, you should use this feature sparingly. If your development or test environments differ wildly from production, you are increasing your chances of different behavior in production, which is a recipe for more defects (or harder-to-find ones). Some differences are inevitable;

for example, if your app is highly database driven, you probably don't want to be messing with the production database during development, and that would be a good candidate for environment-specific configuration. Another low-impact area is more verbose logging. There are a lot of things you might want to log in development that are unnecessary to record in production.

Let's add some logging to our server. The twist is that we want different behavior for production and development. For development, we can leave the defaults, but for production, we want to log to a file. We'll use morgan (don't forget npm install morgan), which is the most common logging middleware (*ch12/00-logging.js* in the companion repo):

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

switch(app.get('env')) {
  case 'development':
    app.use(morgan('dev'))
    break
  case 'production':
    const stream = fs.createWriteStream(__dirname + '/access.log',
      { flags: 'a' })
    app.use(morgan('combined', { stream }))
    break
}
```

If you start the server as you normally would (node meadowlark.js) and visit the site, you'll see the activity logged to the console. To see how the application behaves in production mode, run it with NODE_ENV=production instead. Now if you visit the application, you won't see any activity on the terminal (probably what we want for a production server), but all of the activity is logged in Apache's Combined Log Format, which is a staple for many server tools.

We accomplished this by creating an appendable ({ flags: *a* }) write stream and passing it to the morgan configuration. Morgan has many options; to see them all, check out the morgan documentation.

> In the previous example, we're using __dirname to store the request log in a subdirectory of the project itself. If you take this approach, you will want to add log to your *.gitignore* file. Alternatively, you could take a more Unix-like approach and save the logs in a subdirectory of */var/log*, as Apache does by default.

I will stress again that you should use your best judgment when making environment-specific configuration choices. Always keep in mind that when your site is live, your production instances will be running in production mode (or they should be). Whenever you're tempted to make a development-specific modification, you should always think first about how that might have QA consequences in production. We'll see a more robust example of environment-specific configuration in Chapter 13.

Running Your Node Process

So far, we've been running our application by invoking it directly with `node` (for example, `node meadowlark.js`). This is fine for development and testing, but it has disadvantages for production. Notably, there are no protections if your app crashes or gets terminated. A robust *process manager* can address this problem.

Depending on your hosting solution, you may not need a process manager if one is provided by the hosting solution itself. That is, the hosting provider will give you a configuration option to point to your application file, and it will handle the process management.

But if you need to manage the process yourself, there are two popular options for process managers:

- Forever
- PM2

Since production environments can vary widely, we won't go into the specifics of setting up and configuring a process manager. Both Forever and PM2 have excellent documentation, and you can install and use them on your development machine to learn how to configure them.

I have used them both, and I don't have a strong preference. Forever is a little more straightforward and easy to get started, and PM2 offers more features.

If you want to experiment with a process manager without investing a lot of time, I recommend giving Forever a try. You can try it in two steps. First, install Forever:

```
npm install -g forever
```

Then, start your application with Forever (run this from your application root):

```
forever start meadowlark.js
```

Your application is now running…and it will stay running even if you close your terminal window! You can restart the process with `forever restart meadowlark.js` and stop it with `forever stop meadowlark.js`.

Getting started with PM2 is a little more involved but is worth looking into if you need to use your own process manager for production.

Scaling Your Website

These days, scaling usually means one of two things: scaling up or scaling out. *Scaling up* refers to making servers more powerful: faster CPUs, better architecture, more cores, more memory, etc. *Scaling out*, on the other hand, simply means more servers. With the increased popularity of cloud computing and the ubiquity of virtualization, server computational power is becoming less relevant, and scaling out is usually the most cost-effective method for scaling websites according to your needs.

When developing websites for Node, you should always consider the possibility of scaling out. Even if your application is tiny (maybe it's even an intranet application that will always have a limited audience) and will never conceivably need to be scaled out, it's a good habit to get into. After all, maybe your next Node project will be the next Twitter, and scaling out will be essential. Fortunately, Node's support for scaling out is very good, and writing your application with this in mind is painless.

The most important thing to remember when building a website designed to be scaled out is persistence. If you're used to relying on file-based storage for persistence, *stop right there*. That way lies madness.

My first experience with this problem was nearly disastrous. One of our clients was running a web-based contest, and the web application was designed to inform the first 50 winners that they would receive a prize. With that particular client, we were unable to easily use a database because of some corporate IT restrictions, so most persistence was achieved by writing flat files. I proceeded just as I always had, saving each entry to a file. Once the file had recorded 50 winners, no more people would be notified that they had won. The problem is that the server was load-balanced, so half the requests were served by one server, and the other half by another. One server notified 50 people that they had won…and so did the other server. Fortunately, the prizes were small (fleece blankets) and not something expensive like iPads, and the client took their lumps and handed out 100 prizes instead of 50 (I offered to pay for the extra 50 blankets out of pocket for my mistake, but they generously refused to take me up on my offer).

The moral of this story is that unless you have a filesystem that's accessible to *all* of your servers, you should not rely on the local filesystem for persistence. The exceptions are read-only data, like logging, and backups. For example, I have commonly backed up form submission data to a local flat file in case the database connection failed. In the case of a database outage, it is a hassle to go to each server and collect the files, but at least no damage has been done.

Scaling Out with App Clusters

Node itself supports *app clusters*, a simple, single-server form of scaling out. With app clusters, you can create an independent server for each core (CPU) on the system (having more servers than the number of cores will not improve the performance of your app). App clusters are good for two reasons: first, they can help maximize the performance of a given server (the hardware or virtual machine), and second, it's a low-overhead way to test your app under parallel conditions.

Let's go ahead and add cluster support to our website. While it's quite common to do all of this work in your main application file, we are going to create a second application file that will run the app in a cluster, using the nonclustered application file we've been using all along. To enable that, we have to make a slight modification to *meadowlark.js* first (see *ch12/01-server.js* in the companion repo for a simplified example):

```
function startServer(port) {
  app.listen(port, function() {
    console.log(`Express started in ${app.get('env')} ` +
      `mode on http://localhost:${port}` +
      `; press Ctrl-C to terminate.`)
  })
}

if(require.main === module) {
  // application run directly; start app server
  startServer(process.env.PORT || 3000)
} else {
  // application imported as a module via "require": export
  // function to create server
  module.exports = startServer
}
```

If you recall from Chapter 5, if `require.main === module`, it means the script has been run directly; otherwise, it has been called with `require` from another script.

Then, we create a new script, *meadowlark-cluster.js* (see *ch12/01-cluster* in the companion repo for a simplified example):

```
const cluster = require('cluster')

function startWorker() {
  const worker = cluster.fork()
  console.log(`CLUSTER: Worker ${worker.id} started`)
}

if(cluster.isMaster){

  require('os').cpus().forEach(startWorker)

  // log any workers that disconnect; if a worker disconnects, it
  // should then exit, so we'll wait for the exit event to spawn
  // a new worker to replace it
  cluster.on('disconnect', worker => console.log(
    `CLUSTER: Worker ${worker.id} disconnected from the cluster.`
  ))

  // when a worker dies (exits), create a worker to replace it
  cluster.on('exit', (worker, code, signal) => {
    console.log(
      `CLUSTER: Worker ${worker.id} died with exit ` +
      `code ${code} (${signal})`
    )
    startWorker()
  })

} else {

    const port = process.env.PORT || 3000
    // start our app on worker; see meadowlark.js
    require('./meadowlark.js')(port)

}
```

When this JavaScript is executed, it will be either in the context of master (when it is run directly, with `node meadowlark-cluster.js`) or in the context of a worker, when Node's cluster system executes it. The properties `cluster.isMaster` and `cluster.isWorker` determine which context you're running in. When we run this script, it's executing in master mode, and we start a worker using `cluster.fork` for each CPU in the system. Also, we respawn any dead workers by listening for `exit` events from workers.

Finally, in the `else` clause, we handle the worker case. Since we configured *meadowlark.js* to be used as a module, we simply import it and immediately invoke it (remember, we exported it as a function that starts the server).

Now start up your new clustered server:

```
node meadowlark-cluster.js
```

If you are using virtualization (like Oracle's VirtualBox), you may have to configure your VM to have multiple CPUs. By default, virtual machines often have a single CPU.

Assuming you're on a multicore system, you should see some number of workers started. If you want to see evidence of different workers handling different requests, add the following middleware before your routes:

```
const cluster = require('cluster')

app.use((req, res, next) => {
  if(cluster.isWorker)
    console.log(`Worker ${cluster.worker.id} received request`)
  next()
})
```

Now you can connect to your application with a browser. Reload a few times and see how you can get a different worker out of the pool on each request. (You may not be able to; Node is designed to handle large numbers of connections, and you may not be able to stress it sufficiently simply by reloading your browser; later we'll explore stress testing, and you'll be able to better see the cluster in action.)

Handling Uncaught Exceptions

In the asynchronous world of Node, uncaught exceptions are of particular concern. Let's start with a simple example that doesn't cause too much trouble (I encourage you to follow along with these examples):

```
app.get('/fail', (req, res) => {
  throw new Error('Nope!')
})
```

When Express executes route handlers, it wraps them in a try/catch block, so this isn't actually an uncaught exception. This won't cause too much of a problem: Express will log the exception on the server side, and the visitor will get an ugly stack dump. However, your server is stable, and other requests will continue to be served correctly. If we want to provide a "nice" error page, create a file *views/500.handlebars* and add an error handler after all of your routes:

```
app.use((err, req, res, next) => {
  console.error(err.message, err.stack)
  app.status(500).render('500')
})
```

It's always a good practice to provide a custom error page; it not only looks more professional to your users when errors do occur, but also allows you to take action when errors occur. For example, this error handler would be a good place to notify your dev team that an error occurred. Unfortunately, this helps only for exceptions that Express can catch. Let's try something worse:

```
app.get('/epic-fail', (req, res) => {
  process.nextTick(() =>
    throw new Error('Kaboom!')
  )
})
```

Go ahead and try it. The result is considerably more catastrophic: it brings your whole server down! In addition to not displaying a friendly error message to your user, now your server is down, and *no* requests are being served. This is because setTimeout is executing *asynchronously*; execution of the function with the exception is being deferred until Node is idle. The problem is, when Node is idle and gets around to executing the function, it no longer has context about the request it was being served from, so it has no recourse but to unceremoniously shut down the whole server, because now it's in an undefined state. (Node can't know the purpose of the function or its caller, so it can no longer assume that any further functions will work correctly.)

> process.nextTick is similar to calling setTimeout with an argument of 0, but it's more efficient. We're using it here for demonstration purposes; it's not something you would generally use in server-side code. However, in coming chapters, we will be dealing with many things that execute asynchronously, such as database access, filesystem access, and network access, to name a few, and they are all subject to this problem.

There is action that we can take to handle uncaught exceptions, but *if Node can't determine the stability of your application, neither can you*. In other words, if there is an uncaught exception, the only recourse is to shut down the server. The best we can do in this circumstance is to shut down as gracefully as possible and have a failover mechanism. The easiest failover mechanism is to use a cluster. If your application is operating in clustered mode and one worker dies, the master will spawn another worker to take its place. (You don't even need multiple workers; a cluster with one worker will suffice, though the failover may be slightly slower.)

So with that in mind, how can we shut down as gracefully as possible when confronted with an unhandled exception? Node's mechanism for dealing with this is the uncaughtException event. (Node also has a mechanism called *domains*, but this module has been deprecated, and its use is no longer recommended.)

```
process.on('uncaughtException', err => {
  console.error('UNCAUGHT EXCEPTION\n', err.stack);
  // do any cleanup you need to do here...close
  // database connections, etc.
  process.exit(1)
})
```

It's unrealistic to expect that your application will never experience uncaught exceptions, but you should have a mechanism in place to record the exception and notify you when it happens, and you should take it seriously. Try to determine why it happened so you can fix it. Services like Sentry, Rollbar, Airbrake, and New Relic are a great way to record these kinds of errors for analysis. For example, to use Sentry, first you have to register for a free account, at which point you will receive a data source name (DSN), and then you can modify your exception handler:

```
const Sentry = require('@sentry/node')
Sentry.init({ dsn: '** YOUR DSN GOES HERE **' })

process.on('uncaughtException', err => {
  // do any cleanup you need to do here...close
  // database connections, etc.
  Sentry.captureException(err)
  process.exit(1)
})
```

Scaling Out with Multiple Servers

Although scaling out using clustering can maximize the performance of an individual server, what happens when you need more than one server? That's where things get a little more complicated. To achieve this kind of parallelism, you need a *proxy* server. (It's often called a *reverse proxy* or *forward-facing proxy* to distinguish it from proxies commonly used to access external networks, but I find this language to be confusing and unnecessary, so I will simply refer to it as a proxy.)

Two very popular options are NGINX (pronounced "engine X") and HAProxy. NGINX servers in particular are springing up like weeds. I recently did a competitive analysis for my company and found upward of 80% of our competitors were using NGINX. NGINX and HAProxy are both robust, high-performance proxy servers and are capable of the most demanding applications. (If you need proof, consider that Netflix, which accounts for as much as 15% of *all internet traffic*, uses NGINX.)

There are also some smaller Node-based proxy servers, such as node-http-proxy. This is a great option if your needs are modest, or for development. For production, I recommend using NGINX or HAProxy (both are free, though they offer support for a fee).

Installing and configuring a proxy is beyond the scope of this book, but it is not as hard as you might think (especially if you use node-http-proxy or another lightweight proxy). For now, using clusters gives us some assurance that our website is ready for scaling out.

If you do configure a proxy server, make sure you tell Express that you are using a proxy and that it should be trusted:

```
app.enable('trust proxy')
```

Doing this will ensure that `req.ip`, `req.protocol`, and `req.secure` will reflect the details about the connection between the *client and the proxy*, not between the client and your app. Also, `req.ips` will be an array that indicates the original client IP and the names or IP addresses of any intermediate proxies.

Monitoring Your Website

Monitoring your website is one of the most important—and most often overlooked—QA measures you can take. The only thing worse than being up at 3 a.m. fixing a broken website is being woken up at 3 a.m. by your boss because the website is down (or, worse still, arriving in the morning to realize that your client just lost $10,000 in sales because the website had been down all night and no one noticed).

There's nothing you can do about failures: they are as inevitable as death and taxes. However, if there is one thing you can do to convince your boss and your clients that you are great at your job, it's to *always* know about failures before they do.

Third-Party Uptime Monitors

Having an uptime monitor running on your website's server is as effective as having a smoke alarm in a house that nobody lives in. It might be able to catch errors if a certain page goes down, but if the whole *server* goes down, it may go down without even sending out an SOS. That's why your first line of defense should be third-party uptime monitors. UptimeRobot is free for up to 50 monitors and is simple to configure. Alerts can go to email, SMS (text message), Twitter, or Slack (among others). You can monitor for the return code from a single page (anything other than a 200 is considered an error) or to check for the presence or absence of a keyword on the page. Keep in mind that if you use a keyword monitor, it may affect your analytics (you can exclude traffic from uptime monitors in most analytics services).

If your needs are more sophisticated, there are other, more expensive services out there such as Pingdom and Site24x7.

Stress Testing

Stress testing (or *load testing*) is designed to give you some confidence that your server will function under the load of hundreds or thousands of simultaneous requests. This is another deep area that could be the subject for a whole book: stress testing can be arbitrarily sophisticated, and how complicated you want to get depends largely on the nature of your project. If you have reason to believe that your site could be massively popular, you might want to invest more time in stress testing.

Let's add a simple stress test using Artillery. First, install Artillery by running `npm install -g artillery`; then edit your *package.json* file and add the following to the `scripts` section:

```
"scripts": {
  "stress": "artillery quick --count 10 -n 20 http://localhost:3000/"
}
```

This will simulate 10 “virtual users” (`--count 10`), each of whom will send 20 requests (`-n 20`) to your server.

Make sure your application is running (in a separate terminal window, for example) and then run `npm run stress`. You’ll see statistics like this:

```
Started phase 0, duration: 1s @ 16:43:37(-0700) 2019-04-14
Report @ 16:43:38(-0700) 2019-04-14
Elapsed time: 1 second
  Scenarios launched:  10
  Scenarios completed: 10
  Requests completed:  200
  RPS sent: 147.06
  Request latency:
    min: 1.8
    max: 10.3
    median: 2.5
    p95: 4.2
    p99: 5.4
  Codes:
    200: 200

All virtual users finished
Summary report @ 16:43:38(-0700) 2019-04-14
  Scenarios launched:  10
  Scenarios completed: 10
  Requests completed:  200
  RPS sent: 145.99
  Request latency:
    min: 1.8
    max: 10.3
    median: 2.5
    p95: 4.2
    p99: 5.4
  Scenario counts:
    0: 10 (100%)
  Codes:
    200: 200
```

This test was run on my development laptop. You can see Express didn’t take more than 10.3 milliseconds to serve any requests, and 99% of them were served in under 5.4 milliseconds. I can’t offer concrete guidance about what kind of numbers you should be looking for, but to ensure a snappy application, you should be looking for total connection times under 50 milliseconds. (Don’t forget that this is just the time it takes the server to deliver the data to the client; the client still has to render it, which takes time, so the less time you spend transmitting the data, the better.)

If you stress test your application regularly and benchmark it, you’ll be able to recognize problems. If you just finished a feature and you find that your connection times have tripled, you might want to do some performance tuning on your new feature!

Conclusion

I hope this chapter has given you some insight into the things you'll want to think about as you approach the launch of your application. There is a lot of detail that goes into a production application, and while you can't anticipate everything that might happen when you launch, the more you can anticipate, the better off you'll be. To paraphrase Louis Pasteur, fortune favors the prepared.

CHAPTER 13
Persistence

All but the simplest websites and web applications are going to require *persistence* of some kind; that is, some way to store data that's more permanent than volatile memory so that your data will survive server crashes, power outages, upgrades, and relocations. In this chapter, we'll be discussing the options available for persistence and demonstrating both document databases and relational databases. Before we jump in to databases, however, we'll start with the most basic form of persistence: filesystem persistence.

Filesystem Persistence

One way to achieve persistence is to simply save data to so-called flat files (*flat* because there's no inherent structure in a file; it's just a sequence of bytes). Node makes filesystem persistence possible through the `fs` (filesystem) module.

Filesystem persistence has some drawbacks. In particular, it doesn't scale well. The minute you need more than one server to meet traffic demands, you will run into problems with filesystem persistence, unless all of your servers have access to a shared filesystem. Also, because flat files have no inherent structure, the burden of locating, sorting, and filtering data will be on your application. For these reasons, you should favor databases over filesystems for storing data. The one exception is storing binary files, such as images, audio files, or videos. While many databases can handle this type of data, they rarely do so more efficiently than a filesystem (though information *about* the binary files is usually stored in a database to enable searching, sorting, and filtering).

If you do need to store binary data, keep in mind that filesystem storage still has the problem of not scaling well. If your hosting doesn't have access to a shared filesystem (which is usually the case), you should consider storing binary files in a database (which usually requires some configuration so the database doesn't grind to a stop) or a cloud-based storage service, like Amazon S3 or Microsoft Azure Storage.

Now that we have the caveats out of the way, let's look at Node's filesystem support. We'll revisit the vacation photo contest from Chapter 8. In our application file, let's fill in the handler that processes that form (*ch13/00-mongodb/lib/handlers.js* in the companion repo):

```
const pathUtils = require('path')
const fs = require('fs')

// create directory to store vacation photos (if it doesn't already exist)
const dataDir = pathUtils.resolve(__dirname, '..', 'data')
const vacationPhotosDir = pathUtils.join(dataDir, 'vacation-photos')
if(!fs.existsSync(dataDir)) fs.mkdirSync(dataDir)
if(!fs.existsSync(vacationPhotosDir)) fs.mkdirSync(vacationPhotosDir)

function saveContestEntry(contestName, email, year, month, photoPath) {
  // TODO...this will come later
}

// we'll want these promise-based versions of fs functions later
const { promisify } = require('util')
const mkdir = promisify(fs.mkdir)
const rename = promisify(fs.rename)

exports.api.vacationPhotoContest = async (req, res, fields, files) => {
  const photo = files.photo[0]
  const dir = vacationPhotosDir + '/' + Date.now()
  const path = dir + '/' + photo.originalFilename
  await mkdir(dir)
  await rename(photo.path, path)
  saveContestEntry('vacation-photo', fields.email,
    req.params.year, req.params.month, path)
  res.send({ result: 'success' })
}
```

There's a lot going on there, so let's break it down. We first create a directory to store the uploaded files (if it doesn't already exist). You'll probably want to add the *data* directory to your *.gitignore* file so you don't accidentally commit uploaded files. Recall from Chapter 8 that we're handling the actual file upload in *meadowlark.js* and calling our handler with the files already decoded. What we get is an object (`files`) that contains the information about the uploaded files. Since we want to prevent collisions, we can't just use the filename the user uploaded (in case two users both upload *portland.jpg*). To avoid this problem, we create a unique directory based on the timestamp; it's pretty unlikely that two users will both upload *portland.jpg* in the same millisecond! Then we rename (move) the uploaded file (our file processor will have given it a temporary name, which we can get from the `path` property) to our constructed name.

Finally, we need some way to associate the files that users upload with their email addresses (and the month and year of the submission). We could encode this information into the file or directory names, but we are going to prefer storing this information in a database. Since we haven't learned how to do that yet, we're going to encapsulate that functionality in the `vacationPhotoContest` function and complete that function later in this chapter.

> In general, you should never trust anything that the user uploads because it's a possible vector for your website to be attacked. For example, a malicious user could easily take a harmful executable, rename it with a *.jpg* extension, and upload it as the first step in an attack (hoping to find some way to execute it at a later point). Likewise, we are taking a little risk here by naming the file using the `name` property provided by the browser; someone could also abuse this by inserting special characters into the filename. To make this code completely safe, we would give the file a random name, taking only the extension (making sure it consists only of alphanumeric characters).

Even though filesystem persistence has its drawbacks, it's frequently used for intermediate file storage, and it's useful to know how to use the Node filesystem library. However, to address the deficiencies of filesystem storage, let's turn our attention to cloud persistence.

Cloud Persistence

Cloud storage is becoming increasingly popular, and I highly recommend you take advantage of one of these inexpensive, robust services.

When using cloud services, there's a certain amount of up-front work you have to do. Obviously, you have to create an account, but you also have to understand how your application authenticates with the cloud service, and it's also helpful to understand some basic terminology (for example, AWS calls its file storage mechanism *buckets*, while Azure calls them *containers*). It's beyond the scope of this book to detail all of that information, and it is well-documented:

- AWS: Getting Started in Node.js
- Azure for JavaScript and Node.js Developers

The good news is that once you get past this initial configuration, using cloud persistence is quite easy. Here's an example of how easy it is to save a file to an Amazon S3 account:

```
const filename = 'customerUpload.jpg'

s3.putObject({
  Bucket: 'uploads',
  Key: filename,
  Body: fs.readFileSync(__dirname + '/tmp/ + filename),
})
```

See the AWS SDK documentation for more information.

Here's an example of how to do the same thing with Microsoft Azure:

```
const filename = 'customerUpload.jpg'

const blobService = azure.createBlobService()
blobService.createBlockBlobFromFile('uploads', filename, __dirname +
  '/tmp/' + filename)
```

See the Microsoft Azure documentation for more information.

Now that we know a couple of techniques for file storage, let's consider the storage of structured data with databases.

Database Persistence

All except the simplest websites and web applications require a database. Even if the bulk of your data is binary and you're using a shared filesystem or cloud storage, the chances are you'll want a database to help catalog that binary data.

Traditionally, the word *database* is shorthand for *relational database management system* (RDBMS). Relational databases, such as Oracle, MySQL, PostgreSQL, or SQL Server, are based on decades of research and formal database theory. It is a technology that is quite mature at this point, and the power of these databases is unquestionable. However, we now have the luxury of expanding our ideas of what constitutes a database. NoSQL databases have come into vogue in recent years, and they're challenging the status quo of internet data storage.

It would be foolish to claim that NoSQL databases are somehow better than relational databases, but they do have certain advantages (and vice versa). While it is quite easy to integrate a relational database with Node apps, there are NoSQL databases that seem almost to have been designed for Node.

The two most popular types of NoSQL databases are *document databases* and *key-value* databases. Document databases excel at storing objects, which makes them a natural fit for Node and JavaScript. Key-value databases, as the name implies, are extremely simple and are a great choice for applications with data schemas that are easily mapped into key-value pairs.

I feel that document databases represent the optimal compromise between the constraints of relational databases and the simplicity of key-value databases, and for that reason, we will be using a document database for our first example. MongoDB is the leading document database and is robust and established at this point.

For our second example, we'll be using PostgreSQL, a popular and robust open source RDBMS.

A Note on Performance

The simplicity of NoSQL databases is a double-edged sword. Carefully planning a relational database can be an involved task, but the benefit of that careful planning is a database that offers excellent performance. Don't be fooled into thinking that because NoSQL databases are generally simpler, there isn't an art and a science to tuning them for maximum performance.

Relational databases have traditionally relied on their rigid data structures and decades of optimization research to achieve high performance. NoSQL databases, on the other hand, have embraced the distributed nature of the internet and, like Node, have instead focused on concurrency to scale performance (relational databases also support concurrency, but this is usually reserved for the most demanding applications).

Planning for database performance and scalability is a large, complex topic that is beyond the scope of this book. If your application requires a high level of database performance, I recommend starting with Kristina Chodorow and Michael Dirolf's *MongoDB: The Definitive Guide* (O'Reilly).

Abstracting the Database Layer

In this book, we'll be implementing the same features and demonstrating how to do that with two databases (and not just two databases but two substantially different database architectures). While the objective in this book is to cover two popular options for database architecture, it reflects a real-world scenario: switching a major component of your web application midproject. This could happen for many reasons. Usually it boils down to discovering that a different technology is going to be more cost-effective or allow you to implement necessary features more quickly.

Whenever possible, there is value in *abstracting* your technology choices, which refers to writing some kind of API layer to generalize the underlying technology choices. If done right, it reduces the cost of switching out the component in question. However, it comes at a cost: writing the abstraction layer is one more thing you have to write and maintain.

Happily, our abstraction layer will be very small, as we're supporting only a handful of features for the purposes of this book. For now, the features will be as follows:

- Returning a list of active vacations from the database
- Storing the email address of users who want to be notified when certain vacations are in season

While this seems simple enough, there are a lot of details here. What does a vacation look like? Do we always want to get all the vacations from the database, or do we want to be able to filter them or paginate them? How do we identify vacations? And so on.

We're going to keep our abstraction layer simple for the purposes of this book. We'll contain it in a file called *db.js* that will export two methods that we'll start by just providing dummy implementations:

```
module.exports = {
  getVacations: async (options = {}) => {
    // let's fake some vacation data:
    const vacations = [
      {
        name: 'Hood River Day Trip',
        slug: 'hood-river-day-trip',
        category: 'Day Trip',
        sku: 'HR199',
        description: 'Spend a day sailing on the Columbia and ' +
          'enjoying craft beers in Hood River!',
        location: {
          // we'll use this for geocoding later in the book
          search: 'Hood River, Oregon, USA',
        },
        price: 99.95,
        tags: ['day trip', 'hood river', 'sailing', 'windsurfing', 'breweries'],
        inSeason: true,
        maximumGuests: 16,
        available: true,
        packagesSold: 0,
      }
    ]
    // if the "available" option is specified, return only vacations that match
    if(options.available !== undefined)
      return vacations.filter(({ available }) => available === options.available)
    return vacations
  },
  addVacationInSeasonListener: async (email, sku) => {
    // we'll just pretend we did this...since this is
    // an async function, a new promise will automatically
    // be returned that simply resolves to undefined
  },
}
```

This sets an expectation about how our database implementation should look to the application…and all we have to do is make our databases conform to that interface. Note that we're introducing the concept of vacation "availability"; we're doing this so we can easily disable vacations temporarily instead of deleting them from the database. An example use case for this would be a bed and breakfast that contacts you to let you know they are closed for several months for remodeling. We're keeping this separate from the concept of being "in season" because we may want to list out-of-season vacations on the website because people like to plan in advance.

We also include some very generic "location" information; we'll be getting more specific about this in Chapter 19.

Now that we have an abstraction foundation for our database layer, let's look at how we can implement database storage with MongoDB.

Setting Up MongoDB

The difficulty involved in setting up a MongoDB instance varies with your operating system. For this reason, we'll be avoiding the problem altogether by using an excellent free MongoDB hosting service, mLab.

> mLab is not the only MongoDB service available. The MongoDB company itself is now offering free and low-cost database hosting through its product MongoDB Atlas. Free accounts are not recommended for production purposes, though. Both mLab and MongoDB Atlas offer production-ready accounts, so you should look into their pricing before making a choice. It will be less hassle to stay with the same hosting service when you make the switch to production.

Getting started with mLab is simple. Just go to *https://mlab.com* and click Sign Up. Fill out the registration form and log in, and you'll be at your home screen. Under Databases, you'll see "no databases at this time." Click "Create new," and you will be taken to a page with some options for your new database. The first thing you'll select is a cloud provider. For a free (sandbox) account, the choice is largely irrelevant, though you should look for a data center near you (not every data center will offer sandbox accounts, however). Select SANDBOX, and choose a region. Then choose a database name, and click through to Submit Order (it's still an order even though it's free!). You will be taken back to the list of your databases, and after a few seconds, your database will be available for use.

Having a database set up is half the battle. Now we have to know how to access it with Node, and that's where Mongoose comes in.

Mongoose

While there's a low-level driver available for MongoDB, you'll probably want to use an object document mapper (ODM). The most popular ODM for MongoDB is *Mongoose*.

One of the advantages of JavaScript is that its object model is extremely flexible. If you want to add a property or method to an object, you just do it, and you don't need to worry about modifying a class. Unfortunately, that kind of freewheeling flexibility can have a negative impact on your databases because they can become fragmented and hard to optimize. Mongoose attempts to strike a balance by introducing *schemas* and *models* (combined, schemas and models are similar to classes in traditional object-oriented programming). The schemas are flexible but still provide some necessary structure for your database.

Before we get started, we'll need to install the Mongoose module:

```
npm install mongoose
```

Then we'll add our database credentials to our *.credentials.development.json* file:

```
"mongo": {
    "connectionString": "your_dev_connection_string"
  }
}
```

You'll find your connection string on the database page in mLab. From your home screen, click the appropriate database. You'll see a box with your MongoDB connection URI (it starts with *mongodb://*). You'll also need a user for your database. To create one, click Users, and then "Add database user."

Notice that we could establish a second set of credentials for production by creating a *.credentials.production.js* file and using `NODE_ENV=production`; you'll want to do this when it's time to go live!

Now that we have all the configuration done, let's actually make a connection to the database and do something useful!

Database Connections with Mongoose

We'll start by creating a connection to our database. We'll put our database initialization code in *db.js*, along with the dummy API we created earlier (*ch13/00-mongodb/db.js* in the companion repo):

```
const mongoose = require('mongoose')
const { connectionString } = credentials.mongo
if(!connectionString) {
  console.error('MongoDB connection string missing!')
  process.exit(1)
}
```

```
mongoose.connect(connectionString)
const db = mongoose.connection
db.on('error' err => {
  console.error('MongoDB error: ' + err.message)
  process.exit(1)
})
db.once('open', () => console.log('MongoDB connection established'))

module.exports = {
  getVacations: async () => {
    //...return fake vacation data
  },
  addVacationInSeasonListener: async (email, sku) => {
    //...do nothing
  },
}
```

Any file that needs to access the database can simply import *db.js*. However, we want the initialization to happen right away, before we need the API, so we'll go ahead and import this from *meadowlark.js* (where we don't need to do anything with the API):

```
require('./db')
```

Now that we're connecting to the database, it's time to consider how we're going to structure data that we're transferring to and from the database.

Creating Schemas and Models

Let's create a vacation package database for Meadowlark Travel. We start by defining a schema and creating a model from it. Create the file *models/vacation.js* (*ch13/00-mongodb/models/vacation.js* in the companion repo):

```
const mongoose = require('mongoose')

const vacationSchema = mongoose.Schema({
  name: String,
  slug: String,
  category: String,
  sku: String,
  description: String,
  location: {
    search: String,
    coordinates: {
      lat: Number,
      lng: Number,
    },
  },
  price: Number,
  tags: [String],
  inSeason: Boolean,
  available: Boolean,
  requiresWaiver: Boolean,
  maximumGuests: Number,
  notes: String,
  packagesSold: Number,
})

const Vacation = mongoose.model('Vacation', vacationSchema)
module.exports = Vacation
```

This code declares the properties that make up our vacation model, and the types of those properties. You'll see there are several string properties, some numeric properties, two Boolean properties, and an array of strings (denoted by `[String]`). At this point, we can also define methods on our schema. Each product has a stock keeping unit (SKU); even though we don't think about vacations being "stock items," the concept of an SKU is pretty standard for accounting, even when tangible goods aren't being sold.

Once we have the schema, we create a model using `mongoose.model`: at this point, `Vacation` is very much like a class in traditional object-oriented programming. Note that we have to define our methods before we create our model.

> Because of the nature of floating-point numbers, you should always be careful with financial computations in JavaScript. We could store our prices in cents instead of dollars, which would help, but it doesn't eliminate the problems. For the modest purposes of our travel website, we're not going to worry about it, but if your application involves very large or very small financial amounts (for example, fractional cents from interest or volume trading), you should consider using a library such as currency.js or decimal.js-light. Also, JavaScript's BigInt built-in object, which is available as of Node 10 (with limited browser support as I write this), can be used for this purpose.

We are exporting the `Vacation` model object created by Mongoose. While we could use this model directly, that would be undermining our effort to provide a database abstraction layer. So we will choose to import it only from the *db.js* file and let the rest of our application use its methods. Add the `Vacation` model to *db.js*:

```
const Vacation = require('./models/vacation')
```

All of our structures are now defined, but our database isn't very interesting because there's nothing actually in it. Let's make it useful by seeding it with some data.

Seeding Initial Data

We don't yet have any vacation packages in our database, so we'll add some to get us started. Eventually, you may want to create a way to manage products, but for the purposes of this book, we're just going to do it in code (*ch13/00-mongodb/db.js* in the companion repo):

```
Vacation.find((err, vacations) => {
  if(err) return console.error(err)
  if(vacations.length) return

  new Vacation({
    name: 'Hood River Day Trip',
    slug: 'hood-river-day-trip',
    category: 'Day Trip',
    sku: 'HR199',
    description: 'Spend a day sailing on the Columbia and ' +
      'enjoying craft beers in Hood River!',
    location: {
      search: 'Hood River, Oregon, USA',
    },
    price: 99.95,
    tags: ['day trip', 'hood river', 'sailing', 'windsurfing', 'breweries'],
    inSeason: true,
    maximumGuests: 16,
    available: true,
    packagesSold: 0,
  }).save()

  new Vacation({
    name: 'Oregon Coast Getaway',
    slug: 'oregon-coast-getaway',
    category: 'Weekend Getaway',
    sku: 'OC39',
    description: 'Enjoy the ocean air and quaint coastal towns!',
    location: {
      search: 'Cannon Beach, Oregon, USA',
    },
    price: 269.95,
    tags: ['weekend getaway', 'oregon coast', 'beachcombing'],
    inSeason: false,
    maximumGuests: 8,
    available: true,
    packagesSold: 0,
  }).save()

  new Vacation({
      name: 'Rock Climbing in Bend',
      slug: 'rock-climbing-in-bend',
      category: 'Adventure',
      sku: 'B99',
      description: 'Experience the thrill of climbing in the high desert.',
      location: {
        search: 'Bend, Oregon, USA',
      },
      price: 289.95,
      tags: ['weekend getaway', 'bend', 'high desert', 'rock climbing'],
      inSeason: true,
      requiresWaiver: true,
      maximumGuests: 4,
      available: false,
      packagesSold: 0,
      notes: 'The tour guide is currently recovering from a skiing accident.',
  }).save()
})
```

There are two Mongoose methods being used here. The first, `find`, does just what it says. In this case, it's finding all instances of `Vacation` in the database and invoking the callback with that list. We're doing that because we don't want to keep re-adding our seed vacations: if there are already vacations in the database, it's been seeded, and we can go on our merry way. The first time this executes, though, `find` will return an empty list, so we proceed to create two vacations and then call the `save` method on them, which saves these new objects to the database.

Now that data is *in* the database, it's time to get it back out!

Retrieving Data

We've already seen the `find` method, which is what we'll use to display a list of vacations. However, this time we're going to pass an option to `find` that will filter the data. Specifically, we want to display only vacations that are currently available.

Create a view for the products page, *views/vacations.handlebars*:

```
<h1>Vacations</h1>
{{#each vacations}}
  <div class="vacation">
    <h3>{{name}}</h3>
    <p>{{description}}</p>
    {{#if inSeason}}
      <span class="price">{{price}}</span>
      <a href="/cart/add?sku={{sku}}" class="btn btn-default">Buy Now!</a>
    {{else}}
      <span class="outOfSeason">We're sorry, this vacation is currently
      not in season.
      {{! The "notify me when this vacation is in season"
          page will be our next task. }}
      <a href="/notify-me-when-in-season?sku={{sku}}">Notify me when
      this vacation is in season.</a>
    {{/if}}
  </div>
{{/each}}
```

Now we can create route handlers that hook it all up. In *lib/handlers.js* (don't forget to import `../db`), we create the handler:

```
exports.listVacations = async (req, res) => {
  const vacations = await db.getVacations({ available: true })
  const context = {
    vacations: vacations.map(vacation => ({
      sku: vacation.sku,
      name: vacation.name,
      description: vacation.description,
      price: '$' + vacation.price.toFixed(2),
      inSeason: vacation.inSeason,
    }))
  }
  res.render('vacations', context)
}
```

We add a route that calls the handler in *meadowlark.js*:

```
app.get('/vacations', handlers.listVacations)
```

If you run this example, you'll see only the one vacation from our dummy database implementation. That's because we've initialized the database and seeded its data, but we haven't replaced the dummy implementation with a real one. So let's do that now. Open *db.js* and modify `getVacations`:

```
module.exports = {
  getVacations: async (options = {}) => Vacation.find(options),
  addVacationInSeasonListener: async (email, sku) => {
    //...
  },
}
```

That was easy! A one-liner. Partially this is because Mongoose is doing a lot of the heavy lifting for us, and the way we've designed our API is similar to the way Mongoose works. When we adapt this later to PostgreSQL, you'll see we have to do a little more work.

> The astute reader may worry that our database abstraction layer isn't doing much to "protect" its technology-neutral objective. For example, a developer may read this code and see that they can pass any Mongoose options along to the vacation model, and then the application would be using features that are specific to Mongoose, which will make it harder to switch databases. We could take some steps to prevent this. Instead of just passing things to Mongoose, we could look for specific options and handle them explicitly, making it clear that any implementation would have to provide those options. But for the sake of this example, we're going to let this slide and keep this code simple.

Most of this should be looking pretty familiar, but there might be some things that surprise you. For instance, how we're handling the view context for the vacation listing might seem odd. Why did we map the products returned from the database to a nearly identical object? One reason is that we want to display the price in a neatly formatted way, so we have to convert it to a formatted string.

We could have saved some typing by doing this:

```
const context = {
  vacations: products.map(vacations => {
    vacation.price = '$' + vacation.price.toFixed(2)
    return vacation
  })
}
```

That would certainly save us a few lines of code, but in my experience, there are good reasons not to pass unmapped database objects directly to views. The view gets a bunch of properties it may not need, possibly in formats that are incompatible with it. Our example is pretty simple so far, but once it starts to get more complicated, you'll probably want to do even more customization of the data that's passed to a view. It also makes it easy to accidentally expose confidential information or information that could compromise the security of your website. For these reasons, I recommend mapping the data that's returned from the database and passing only what's needed onto the view (transforming as necessary, as we did with `price`).

In some variations of the MVC architecture, a third component called a *view model* is introduced. A view model essentially distills and transforms a model (or models) so that it's more appropriate for display in a view. What we're doing here is creating a view model on the fly.

We've come a long way at this point. We're successfully using a database to store information about our vacations. But databases wouldn't be very useful if we couldn't update them. Let's turn our attention to that aspect of interfacing with databases.

Adding Data

We've already seen how we can add data (we added data when we seeded the vacation collection) and how we can update data (we update the count of packages sold when we book a vacation), but let's take a look at a slightly more involved scenario that highlights the flexibility of document databases.

When a vacation is out of season, we display a link that invites the customer to be notified when the vacation is in season again. Let's hook up that functionality. First, we create the schema and model (*models/vacationInSeasonListener.js*):

```
const mongoose = require('mongoose')

const vacationInSeasonListenerSchema = mongoose.Schema({
  email: String,
  skus: [String],
})
const VacationInSeasonListener = mongoose.model('VacationInSeasonListener',
  vacationInSeasonListenerSchema)

module.exports = VacationInSeasonListener
```

Then we'll create our view, *views/notify-me-when-in-season.handlebars*:

```
<div class="formContainer">
  <form class="form-horizontal newsletterForm" role="form"
      action="/notify-me-when-in-season" method="POST">
    <input type="hidden" name="sku" value="{{sku}}">
    <div class="form-group">
      <label for="fieldEmail" class="col-sm-2 control-label">Email</label>
      <div class="col-sm-4">
        <input type="email" class="form-control" required
          id="fieldEmail" name="email">
      </div>
    </div>
    <div class="form-group">
      <div class="col-sm-offset-2 col-sm-4">
        <button type="submit" class="btn btn-default">Submit</button>
      </div>
    </div>
  </form>
</div>
```

Then the route handlers:

```
exports.notifyWhenInSeasonForm = (req, res) =>
  res.render('notify-me-when-in-season', { sku: req.query.sku })

exports.notifyWhenInSeasonProcess = (req, res) => {
  const { email, sku } = req.body
  await db.addVacationInSeasonListener(email, sku)
  return res.redirect(303, '/vacations')
}
```

Finally, we add a real implementation to *db.js*:

```
const VacationInSeasonListener = require('./models/vacationInSeasonListener')

module.exports = {
  getVacations: async (options = {}) => Vacation.find(options),
  addVacationInSeasonListener: async (email, sku) => {
    await VacationInSeasonListener.updateOne(
      { email },
      { $push: { skus: sku } },
      { upsert: true }
    )
  },
}
```

What magic is this? How can we "update" a record in the `VacationInSeasonListener` collection before it even exists? The answer lies in a Mongoose convenience called an *upsert* (a portmanteau of "update" and "insert"). Basically, if a record with the given email address doesn't exist, it will be created. If a record does exist, it will be updated. Then we use the magic variable `$push` to indicate that we want to add a value to an array.

> This code doesn't prevent multiple SKUs from being added to the record if the user fills out the form multiple times. When a vacation comes into season and we find all the customers who want to be notified, we will have to be careful not to notify them multiple times.

We've certainly covered the important bases by now! We learned how to connect to a MongoDB instance, seed it with data, read that data out, and write updates to it! However, you may prefer to use an RDBMS, so let's shift gears and see how we can do the same thing with PostgreSQL instead.

PostgreSQL

Object databases like MongoDB are great and are generally quicker to get started with, but if you're trying to build a robust application, you may put as much work—or more—into structuring your object databases as you would planning out a traditional relational database. Furthermore, you may already have experience with relational databases, or you might have an existing relational database you want to connect with.

Fortunately, there is robust support for every major relational database in the JavaScript ecosystem, and if you want or need to use a relational database, you shouldn't have any problem.

Let's take our vacation database and reimplement it using a relational database. For this example, we'll use PostgreSQL, a popular and sophisticated open source relational database. The techniques and principles we'll use will be similar for any relational database.

Similar to the ODM we used for MongoDB, there are object-relational mapping (ORM) tools available for relational databases. However, since most readers interested in this topic are probably already familiar with relational databases and SQL, we'll use a Node PostgreSQL client directly.

Like MongoDB, we'll use a free online PostgreSQL service. Of course, if you're comfortable installing and configuring your own PostgreSQL database, you are welcome to do that as well. All that will change is the connection string. If you do use your own

PostgreSQL instance, make sure you're using 9.4 or later, because we will be using the JSON data type, which was introduced in 9.4 (as I write this, I am using 11.3).

There are many options for online PostgreSQL; for this example, I'll be using ElephantSQL. Getting started couldn't be simpler: create an account (you can use your GitHub account to log in), and click Create New Instance. All you have to do is give it a name (for example, "meadowlark") and select a plan (you can use their free plan). You'll also specify a region (try to pick the one closest to you). Once you're all set up, you'll find a Details section that lists information about your instance. Copy the URL (connection string), which includes the username, password, and instance location all in one convenient string.

Put that string in your *.credentials.development.json* file:

```
"postgres": {
  "connectionString": "your_dev_connection_string"
}
```

One difference between object databases and RDBMSs is that you typically do more up-front work to define the schema of an RDBMS and use data definition SQL to create the schema before adding or retrieving data. In keeping with this paradigm, we'll do that as a separate step instead of letting our ODM or ORM handle it, as we did with MongoDB.

We could create SQL scripts and use a command-line client to execute the data definition scripts that will create our tables, or we could do this work in JavaScript with the PostgreSQL client API, but in a separate step that's done only once. Since this is a book about Node and Express, we'll do the latter.

First, we'll have to install the `pg` client library (`npm install pg`). Then create *db-init.js*, which will be run only to initialize our database and is distinct from our *db.js* file, which is used every time the server starts up (*ch13/01-postgres/db.js* in the companion repo):

```
const { credentials } = require('./config')

const { Client } = require('pg')
const { connectionString } = credentials.postgres
const client = new Client({ connectionString })

const createScript = `
  CREATE TABLE IF NOT EXISTS vacations (
    name varchar(200) NOT NULL,
    slug varchar(200) NOT NULL UNIQUE,
    category varchar(50),
    sku varchar(20),
    description text,
    location_search varchar(100) NOT NULL,
    location_lat double precision,
    location_lng double precision,
    price money,
    tags jsonb,
    in_season boolean,
    available boolean,
    requires_waiver boolean,
    maximum_guests integer,
    notes text,
    packages_sold integer
  );
`

const getVacationCount = async client => {
  const { rows } = await client.query('SELECT COUNT(*) FROM VACATIONS')
  return Number(rows[0].count)
}

const seedVacations = async client => {
  const sql = `
    INSERT INTO vacations(
      name,
      slug,
      category,
      sku,
      description,
      location_search,
      price,
      tags,
      in_season,
      available,
      requires_waiver,
      maximum_guests,
      notes,
      packages_sold
    ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, $11, $12, $13, $14)
  `
  await client.query(sql, [
    'Hood River Day Trip',
    'hood-river-day-trip',
    'Day Trip',
    'HR199',
    'Spend a day sailing on the Columbia and enjoying craft beers in Hood River!',
    'Hood River, Oregon, USA',
    99.95,
    `["day trip", "hood river", "sailing", "windsurfing", "breweries"]`,
    true,
    true,
    false,
    16,
    null,
    0,
  ])
  // we can use the same pattern to insert other vacation data here...
```

```
}

client.connect().then(async () => {
  try {
    console.log('creating database schema')
    await client.query(createScript)
    const vacationCount = await getVacationCount(client)
    if(vacationCount === 0) {
      console.log('seeding vacations')
      await seedVacations(client)
    }
  } catch(err) {
    console.log('ERROR: could not initialize database')
    console.log(err.message)
  } finally {
    client.end()
  }
})
```

Let's start at the bottom of this file. We take our database client (`client`) and call `connect()` on it, which establishes a database connection and returns a promise. When that promise resolves, we can take actions against the database.

The first thing we do is invoke `client.query(createScript)`, which will create our `vacations` table (also known as a *relation*). If we look at `createScript`, we'll see this is data definition SQL. It's beyond the scope of this book to delve into SQL, but if you're reading this section, I assume you have at least a passing understanding of SQL. One thing you may note is that we use snake_case to name our fields instead of camelCase. That is, what was "inSeason" has become "in_season." While it is possible to use camelCase to name structures in PostgreSQL, you have to quote any identifiers with capital letters, which ends up being more trouble than it's worth. We'll come back to that a little later.

You'll see we're already having to put more thought into our schema. How long can a vacation name be? (We're arbitrarily capping it at 200 characters here.) How long can category names and the SKU be? Notice we're using PostgreSQL's `money` type for the price and making the slug be our primary key (instead of adding a separate ID).

If you're already familiar with relational databases, there won't be anything surprising about this simple schema. However, the way we've handled "tags" might have jumped out at you.

In traditional database design, we would probably create a new table to relate vacations to tags (this is called *normalization*). And we could do that here. But here is where we might decide to strike some compromises between traditional relational database design and doing things in the "JavaScript way." If we went with two tables (`vacations` and `vacation_tags`, for example), we'd have to query data from both tables to create a single object that contains all the information about a vacation, as we had in our MongoDB example. And there may be performance reasons for that extra complexity, but let's assume there isn't, and we just want to be able to quickly determine the tags for a particular vacation. We could make this a text field and separate our tags with commas, but then we would have to parse out our tags, and PostgreSQL gives us a better way in JSON data types. We'll see shortly that by specifying this as JSON (`jsonb`, a binary representation that's usually higher performance), we can store this as a JavaScript array, and a JavaScript array comes out, just as we had in MongoDB.

Finally, we insert our seed data into the database by using the same basic concept as before: if the `vacations` table is empty, we add some initial data; otherwise, we assume we've already done that.

You'll note that inserting our data is a little more unwieldy than it was with MongoDB. There are ways to solve this problem, but for this example, I want to be explicit about the use of SQL. We could write a function to make insert statements more naturally, or we could use an ORM (more on this later). But for now, the SQL gets the job done, and it should be comfortable for anyone who already knows SQL.

Note that although this script is designed to be run only once to initialize and seed our database, we've written it in a way that it's safe to run multiple times. We included the `IF NOT EXISTS` option, and we check to see whether the `vacations` table is empty before adding seed data.

We can now run the script to initialize our database:

```
$ node db-init.js
```

Now that we have our database set up, we can write some code to *use* it in our website.

Database servers can typically handle only a limited number of connections at a time, so web servers usually implement a strategy called *connection pooling* to balance the overhead of establishing a connection with the danger of leaving connections open too long and choking the server. Fortunately, the details of this are handled for you by the PostgreSQL Node client.

We'll take a slightly different strategy with our *db.js* file this time. Instead of a file we just require to establish the database connection, it will return an API that we write that handles the details of communicating with the database.

We also have a decision to make about our vacation model. Recall that when we created our model, we used snake_case for our database schema, but all of our JavaScript code uses camelCase. Broadly speaking, we have three options here:

- Refactor our schema to use camelCase. This will make our SQL uglier because we have to remember to quote our property names correctly.

- Use snake_case in our JavaScript. This is less than ideal because we like standards (right?).
- Use snake_case on the database side, and translate to camelCase on the JavaScript side. This is more work that we have to do, but it keeps our SQL and our JavaScript pristine.

Fortunately, the third option can be done automatically. We could write our own function to do that translation, but we'll rely on a popular utility library called Lodash, which makes it extremely easy. Just run `npm install lodash` to install it.

Right now, our database needs are very modest. All we need to do is fetch all available vacation packages, so our *db.js* file will look like this (*ch13/01-postgres/db.js* in the companion repo):

```
const { Pool } = require('pg')
const _ = require('lodash')

const { credentials } = require('./config')

const { connectionString } = credentials.postgres
const pool = new Pool({ connectionString })

module.exports = {
  getVacations: async () => {
    const { rows } = await pool.query('SELECT * FROM VACATIONS')
    return rows.map(row => {
      const vacation = _.mapKeys(row, (v, k) => _.camelCase(k))
      vacation.price = parseFloat(vacation.price.replace(/^\$/, ''))
      vacation.location = {
        search: vacation.locationSearch,
        coordinates: {
          lat: vacation.locationLat,
          lng: vacation.locationLng,
        },
      }
      return vacation
    })
  }
}
```

Short and sweet! We're exporting a single method called `getVacations` that does as advertised. It also uses Lodash's `mapKeys` and `camelCase` functions to convert our database properties to camelCase.

One thing to note is that we have to handle the `price` attribute carefully. PostgreSQL's `money` type is converted to an already-formatted string by the `pg` library. And for good reason: as we've already discussed, JavaScript has only recently added support for arbitrary precision numeric types (`BigInt`), but there isn't yet a PostgreSQL adapter that takes advantage of that (and it might not be the most efficient data type in any event). We could change our database schema to use a numeric type instead of the `money` type, but we shouldn't let our frontend choices drive our schema. We could also deal with the preformatted strings that are being returned from `pg`, but then we would have to change all of our existing code, which is relying on `price` being a number. Furthermore, that approach would undermine our ability to do numeric calculations on the frontend (such as summing the prices of the items in your cart). For all of these reasons, we're opting to parse the string to a number when we retrieve it from the database.

We also take our location information—which is "flat" in the table—and turn it into a more JavaScript-like structure. We're doing this only to achieve parity with our MongoDB example; we could use the data structured as it is (or modify our MongoDB example to have a flat structure).

The last thing we need to learn to do with PostgreSQL is to update data, so let's fill in the "vacation in season" listener feature.

Adding Data

As with the MongoDB example, we'll use our "vacation in season" listener example. We'll start by adding the following data definition to the `createScript` string in *db-init.js*:

```
CREATE TABLE IF NOT EXISTS vacation_in_season_listeners (
  email varchar(200) NOT NULL,
  sku varchar(20) NOT NULL,
  PRIMARY KEY (email, sku)
);
```

Remember that we took care to write *db-init.js* in a nondestructive fashion so we could run it at any time. So we can just run it again to create the `vacation_in_season_listeners` table.

Now we can modify *db.js* to include a method to update this table:

```
module.exports = {
  //...
  addVacationInSeasonListener: async (email, sku) => {
    await pool.query(
      'INSERT INTO vacation_in_season_listeners (email, sku) ' +
      'VALUES ($1, $2) ' +
      'ON CONFLICT DO NOTHING',
      [email, sku]
    )
  },
}
```

PostgreSQL's `ON CONFLICT` clause essentially enables upserts. In this case, if the exact combination of email and SKU is already present, the user has already registered to be notified, so we don't need to do anything. If we had other columns in this table (such as the date of last registration), we might want to use a more sophisticated ON CONFLICT clause (see the PostgreSQL INSERT documentation for more information). Note also that this behavior is dependent on the way we defined the table. We made email and SKU a composite primary key, meaning that there can't be any duplicates, which in turn necessitated the ON CONFLICT clause (otherwise, the INSERT command would result in an error the second time a user tried to register for a notification on the same vacation).

Now we've seen a complete example of hooking up two types of databases, an object database and an RDBMS. It should be clear that the function of the database is the same: storing, retrieving, and updating data in a consistent and scalable fashion. Because the function is the same, we were able to create an abstraction layer so we could choose a different database technology. The last thing we might need a database for is for persistent session storage, which we hinted at in Chapter 9.

Using a Database for Session Storage

As we discussed in Chapter 9, using a memory store for session data is unsuitable in a production environment. Fortunately, it's easy to use a database as a session store.

While we could use our existing MongoDB or PostgreSQL database for a session store, a full-blown database is overkill for session storage, which is a perfect use case for a key-value database. As I write this, the most popular key-value databases for session stores are Redis and Memcached. In keeping with the other examples in this chapter, we'll be using a free online service to provide a Redis database.

Start by heading over to Redis Labs and create an account. Then create a free subscription and plan. Choose Cache for the plan and give the database a name; you can leave the rest of the settings at their defaults.

You'll reach a View Database screen, and, as I write this, the critical information doesn't populate for a few seconds, so be patient. What you'll want is the Endpoint field and the Redis Password under Access Control & Security (it's hidden by default, but there's a little button next to it that will show it). Take these and put them in your *.credentials.development.json* file:

```
"redis": {
  "url": "redis://:<YOUR PASSWORD>@<YOUR ENDPOINT>"
}
```

Note the slightly odd URL: normally there would be a username before the colon preceding your password, but Redis allows connection with a password only; the colon that separates username from password is still required, however.

We'll be using a package called connect-redis to provide Redis session storage. Once you've installed it (npm install connect-redis), we can set it up in our main application file. We still use expression-session, but now we pass a new property to it, store, which configures it to use a database. Note that we have to pass expressSession to the function returned from connect-redis to get the constructor: this is a pretty common quirk of session stores (*ch13/00-mongodb/meadowlark.js* or *ch13/01-postgres/meadowlark.js* in the companion repo):

```
const expressSession = require('express-session')
const RedisStore = require('connect-redis')(expressSession)

app.use(cookieParser(credentials.cookieSecret))
app.use(expressSession({
  resave: false,
  saveUninitialized: false,
  secret: credentials.cookieSecret,
  store: new RedisStore({
    url: credentials.redis.url,
    logErrors: true,  // highly recommended!
  }),
}))
```

Let's use our newly minted session store for something useful. Imagine we want to be able to display vacation prices in different currencies. Furthermore, we want the site to remember the user's currency preference.

We'll start by adding a currency picker at the bottom of our vacations page:

```
<hr>
<p>Currency:
    <a href="/set-currency/USD" class="currency {{currencyUSD}}">USD</a> |
    <a href="/set-currency/GBP" class="currency {{currencyGBP}}">GBP</a> |
    <a href="/set-currency/BTC" class="currency {{currencyBTC}}">BTC</a>
</p>
```

Now here's a little CSS (you can put this inline in your *views/layouts/main.handlebars* file or link to a CSS file in your *public* directory):

```
a.currency {
  text-decoration: none;
}
.currency.selected {
  font-weight: bold;
  font-size: 150%;
}
```

Lastly, we'll add a route handler to set the currency and modify our route handler for */vacations* to display prices in the current currency (*ch13/00-mongodb/lib/handlers.js* or *ch13/01-postgres/lib/handlers.js* in the companion repo):

```
exports.setCurrency = (req, res) => {
  req.session.currency = req.params.currency
  return res.redirect(303, '/vacations')
}

function convertFromUSD(value, currency) {
  switch(currency) {
    case 'USD': return value * 1
    case 'GBP': return value * 0.79
    case 'BTC': return value * 0.000078
    default: return NaN
  }
}

exports.listVacations = (req, res) => {
  Vacation.find({ available: true }, (err, vacations) => {
    const currency = req.session.currency || 'USD'
    const context = {
      currency: currency,
      vacations: vacations.map(vacation => {
        return {
          sku: vacation.sku,
          name: vacation.name,
          description: vacation.description,
          inSeason: vacation.inSeason,
          price: convertFromUSD(vacation.price, currency),
          qty: vacation.qty,
        }
      })
    }
    switch(currency){
      case 'USD': context.currencyUSD = 'selected'; break
      case 'GBP': context.currencyGBP = 'selected'; break
      case 'BTC': context.currencyBTC = 'selected'; break
    }
    res.render('vacations', context)
  })
}
```

You'll also have to add a route for setting the currency in *meadowlark.js*:

```
app.get('/set-currency/:currency', handlers.setCurrency)
```

This isn't a great way to perform currency conversion, of course. We would want to utilize a third-party currency conversion API to make sure our rates are up-to-date. But this will suffice for demonstration purposes. You can now switch between the various currencies and—go ahead and try it—stop and restart your server. You'll find it remembers your currency preference! If you clear your cookies, the currency preference will be forgotten. You'll notice that now we've lost our pretty currency formatting; it's now more complicated, and I will leave that as an exercise for the reader.

Another reader's exercise would be to make the `set-currency` route general-purpose to make it more useful. Currently, it will always redirect to the vacations page, but what if you wanted to use it on a shopping cart page? See if you can think of one or two ways of solving this problem.

If you look in your database, you'll find there's a new collection called *sessions*. If you explore that collection, you'll find a document with your session ID (property `sid`) and your currency preference.

Conclusion

We've certainly covered a lot of ground in this chapter. For most web applications, the database is at the heart of what makes the application useful. Designing and tuning databases is a vast topic that could span many books, but I hope this has given you the basic tools you need to connect two types of databases and move data around.

Now that we have this fundamental piece in place, we're going to revisit routing and the importance it plays in web applications.

CHAPTER 14
Routing

Routing is one of the most important aspects of your website or web service; fortunately, routing in Express is simple, flexible, and robust. *Routing* is the mechanism by which requests (as specified by a URL and HTTP method) are routed to the code that handles them. As we've already noted, routing used to be file based and simple. For example, if you put the file *foo/about.html* on your website, you would access it from the browser with the path */foo/about.html*. Simple but inflexible. And, in case you hadn't noticed, having *html* in your URL is extremely passé these days.

Before we dive into the technical aspects of routing with Express, we should discuss the concept of *information architecture* (IA). IA refers to the conceptual organization of your content. Having an extensible (but not overcomplicated) IA before you begin thinking about routing will pay huge dividends down the line.

One of the most intelligent and timeless essays on IA is by Tim Berners-Lee, who practically *invented the internet*. You can (and should) read it now: *http://www.w3.org/Provider/Style/URI.html*. It was written in 1998. Let that sink in for a minute; there's not much that was written on internet technology in 1998 that is just as true today as it was then.

From that essay, here is the lofty responsibility we are being asked to take on:

> It is the duty of a Webmaster to allocate URIs which you will be able to stand by in 2 years, in 20 years, in 200 years. This needs thought, and organization, and commitment.
>
> —Tim Berners-Lee

I like to think that if web design ever required professional licensing, like other kinds of engineering, that we would take an oath to that effect. (The astute reader of that article will find humor in the fact that the URL to that article ends with *.html*.)

To make an analogy (that may sadly be lost on the younger audience), imagine that every two years your favorite library completely reordered the Dewey decimal system. You would walk into the library one day and you wouldn't be able to find anything. That's exactly what happens when you redesign your URL structure.

Put some serious thought into your URLs. Will they still make sense in 20 years? (200 years may be a bit of a stretch: who knows if we'll even be using URLs by then. Still, I admire the dedication of thinking that far into the future.) Carefully consider the breakdown of your content. Categorize things logically, and try not to paint yourself into a corner. It's a science, but it's also an art.

Perhaps most important, work with others to design your URLs. Even if you are the best information architect for miles around, you might be surprised at how people look at the same content with a different perspective. I'm not saying that you should try for an IA that makes sense from *everyone's* perspective (because that is usually quite impossible), but being able to see the problem from multiple perspectives will give you better ideas and expose the flaws in your own IA.

Here are some suggestions to help you achieve a lasting IA:

Never expose technical details in your URLs

Have you ever been to a website, noticed that the URL ended in *.asp*, and thought that the website was hopelessly out-of-date? Remember that, once upon a time, ASP was cutting-edge. Though it pains me to say it, so too shall fall JavaScript and JSON and Node and Express. I hope it's not for many, many productive years, but time is not often kind to technology.

Avoid meaningless information in your URLs

Think carefully about every word in your URL. If it doesn't mean anything, leave it out. For example, it always makes me cringe when websites use the word *home* in URLs. Your root URL *is* your home page. You don't need to additionally have URLs like */home/directions* and */home/contact*.

Avoid needlessly long URLs

All things being equal, a short URL is better than a longer URL. However, you should not try to make URLs short at the expense of clarity or SEO. Abbreviations are tempting, but think carefully about them. They should be common and ubiquitous before you immortalize them in a URL.

Be consistent with word separators

It's quite common to separate words with hyphens, and a little less common to do so with underscores. Hyphens are generally considered more aesthetically pleasing than underscores, and most SEO experts recommend them. Whether you choose hyphens or underscores, be consistent in their use.

Never use whitespace or untypable characters

Whitespace in a URL is not recommended. It will usually just be converted to a plus sign (+), leading to confusion. It should be obvious that you should avoid untypable characters, and I caution you strongly against using any characters other than alphanumeric characters, numbers, dashes, and underscores. It may feel clever at the time, but "clever" has a way of not standing the test of time. Obviously, if your website is not for an English audience, you may use non-English characters (using percent codes), though that can cause headaches if you ever want to localize your website.

Use lowercase for your URLs

This one will cause some debate. There are those who feel that mixed case in URLs is not only acceptable, but preferable. I don't want to get in a religious debate over this, but I will point out that the advantage of lowercase is that it can always automatically be generated by code. If you've ever had to go through a website and sanitize thousands of links or do string comparisons, you will appreciate this argument. I personally feel that lowercase URLs are more aesthetically pleasing, but in the end, this decision is up to you.

Routes and SEO

If you want your website to be discoverable (and most people do), then you need to think about SEO and how your URLs can affect it. In particular, if there are certain keywords that are important—*and it makes sense*—consider making them part of the URL. For example, Meadowlark Travel offers several Oregon Coast vacations. To ensure high search engine ranking for these vacations, we use the string "Oregon Coast" in the title, header, body, and meta description, and the URLs start with */vacations/oregon-coast*. The Manzanita vacation package can be found at */vacations/oregon-coast/manzanita*. If, to shorten the URL, we simply used */vacations/manzanita*, we would be losing out on valuable SEO.

That said, resist the temptation to carelessly jam keywords into URLs in an attempt to improve your rankings. It will fail. For example, changing the Manzanita vacation URL to */vacations/oregon-coast-portland-and-hood-river/oregon-coast/manzanita* in an effort to say "Oregon Coast" one more time, and also work the "Portland" and "Hood River" keywords in at the same time, is wrong-headed. It flies in the face of good IA and will likely backfire.

Subdomains

Along with the path, subdomains are the other part of the URL that is commonly used to route requests. Subdomains are best reserved for significantly different parts of your application—for example, a REST API (*api.meadowlarktravel.com*) or an admin interface (*admin.meadowlarktravel.com*). Sometimes subdomains are used for technical reasons. For example, if we were to build our blog with WordPress (while the rest of our site uses Express), it can be easier to use *blog.meadowlarktravel.com* (a better solution would be to use a proxy server, such as NGINX). There are usually SEO consequences to partitioning your content using subdomains, which is why you should generally reserve them for areas of your site that aren't important to SEO, such as admin areas and APIs. Keep this in mind and make sure there's no other option before using a subdomain for content that is important to your SEO plan.

The routing mechanism in Express does not take subdomains into account by default: `app.get(/about)` will handle requests for *http://meadowlarktravel.com/about*, *http://www.meadowlarktravel.com/about*, and *http://admin.meadowlarktravel.com/about*. If you want to handle a subdomain separately, you can use a package called `vhost` (for "virtual host," which comes from an Apache mechanism commonly used for handling subdomains). First, install the package (`npm install vhost`). To test domain-based routing on your dev machine, you'll need some way to "fake" domain names. Fortunately, this is what your *hosts file* is for. On macOS and Linux machines, it can be found at */etc/hosts*, and on Windows, it's at *c:\windows\system32\drivers\etc\hosts*. Add the following to your hosts file (you will need admin privileges to edit it):

```
127.0.0.1 admin.meadowlark.local
127.0.0.1 meadowlark.local
```

This tells your computer to treat `meadowlark.local` and `admin.meadowlark.local` just like regular internet domains but to map them to localhost (127.0.0.1). We use the `.local` top-level domain so as not to get confused (you could use `.com` or any other internet domain, but it would override the real domain, which can lead to frustration).

Then you can use the `vhost` middleware to use domain-aware routing (*ch14/00-subdomains.js* in the companion repo):

```
// create "admin" subdomain...this should appear
// before all your other routes
var admin = express.Router()
app.use(vhost('admin.meadowlark.local', admin))

// create admin routes; these can be defined anywhere
admin.get('*', (req, res) => res.send('Welcome, Admin!'))

// regular routes
app.get('*', (req, res) => res.send('Welcome, User!'))
```

`express.Router()` essentially creates a new instance of the Express router. You can treat this instance just like your original instance (`app`). You can add routes and middleware just as you would to `app`. However, it won't do anything until you add it to `app`. We add it through `vhost`, which binds that router instance to that subdomain.

`express.Router` is also useful for partitioning your routes so that you can link in many route handlers at once. See the Express routing documentation for more information.

Route Handlers Are Middleware

We've already seen basic routing of matching a given path. But what does `app.get(\'/foo', ...)` actually *do*? As we saw in Chapter 10, it's simply a specialized piece of middleware, down to having a `next` method passed in. Let's look at some more sophisticated examples (*ch14/01-fifty-fifty.js* in the companion repo):

```
app.get('/fifty-fifty', (req, res, next) => {
  if(Math.random() < 0.5) return next()
  res.send('sometimes this')
})
app.get('/fifty-fifty', (req,res) => {
  res.send('and sometimes that')
})
```

In the previous example, we have two handlers for the same route. Normally, the first one would win, but in this case, the first one is going to pass approximately half the time, giving the second one a chance. We don't even have to use `app.get` twice: you can use as many handlers as you want for a single `app.get` call. Here's an example that has an approximately equal chance of three different responses (*ch14/02-red-green-blue.js* in the companion repo):

```
app.get('/rgb',
  (req, res, next) => {
    // about a third of the requests will return "red"
    if(Math.random() < 0.33) return next()
    res.send('red')
  },
  (req, res, next) => {
    // half of the remaining 2/3 of requests (so another third)
    // will return "green"
    if(Math.random() < 0.5) return next()
    res.send('green')
  },
  function(req, res){
    // and the last third returns "blue"
    res.send('blue')
  },
)
```

While this may not seem particularly useful at first, it allows you to create generic functions that can be used in any of your routes. For example, let's say we have a mechanism that shows special offers on certain pages. The special offers change frequently, and they're not shown on every page. We can create a function to inject the specials into the `res.locals` property (which you'll remember from Chapter 7) (*ch14/03-specials.js* in the companion repo):

```
async function specials(req, res, next) {
  res.locals.special = await getSpecialsFromDatabase()
  next()
}

app.get('/page-with-specials', specials, (req, res) =>
  res.render('page-with-specials')
)
```

We could also implement an authorization mechanism with this approach. Let's say our user authorization code sets a session variable called `req.session.authorized`. We can use the following to make a reusable authorization filter (*ch14/04-authorizer.js* in the companion repo):

```
function authorize(req, res, next) {
  if(req.session.authorized) return next()
  res.render('not-authorized')
}

app.get('/public', () => res.render('public'))

app.get('/secret', authorize, () => res.render('secret'))
```

Route Paths and Regular Expressions

When you specify a path (like */foo*) in your route, it's eventually converted to a regular expression by Express. Some regular expression metacharacters are available in route paths: `+`, `?`, `*`, `(`, and `)`. Let's look at a couple of examples. Let's say you want the URLs */user* and */username* to be handled by the same route:

```
app.get('/user(name)?', (req, res) => res.render('user'))
```

One of my favorite novelty websites—now sadly defunct—was *http://khaaan.com*. All it was was everyone's favorite starship captain belting his most iconic line. Useless, but made me smile every time. Let's say we want to make our own "KHAAAAAAAAN" page but we don't want our users to have to remember if it's 2 *a*'s or 3 or 10. The following will get the job done:

```
app.get('/khaa+n', (req, res) => res.render('khaaan'))
```

Not all normal regex metacharacters have meaning in route paths, though—only the ones listed earlier. This is important, because periods, which are normally a regex metacharacter meaning "any character," can be used in routes unescaped.

Lastly, if you really need the full power of regular expressions for your route, that is supported:

```
app.get(/crazy|mad(ness)?|lunacy/, (req,res) =>
  res.render('madness')
)
```

I have yet to find a good reason for using regex metacharacters in my route paths, much less full regexes, but it's good to be aware the functionality is there.

Route Parameters

While regex routes may find little day-to-day use in your Expression toolbox, you'll most likely be using route parameters quite frequently. In short, it's a way to turn part of your route into a variable parameter. Let's say in our website we want to have a page for each staff member. We have a database of staff members with bios and pictures. As our company grows, it becomes more and more unwieldy to add a new route for each staff member. Let's see how route parameters can help us (*ch14/05-staff.js* in the companion repo):

```
const staff = {
  mitch: { name: "Mitch",
    bio: 'Mitch is the man to have at your back in a bar fight.' },
  madeline: { name: "Madeline", bio: 'Madeline is our Oregon expert.' },
  walt: { name: "Walt", bio: 'Walt is our Oregon Coast expert.' },
}

app.get('/staff/:name', (req, res, next) => {
  const info = staff[req.params.name]
  if(!info) return next()   // will eventually fall through to 404
  res.render('05-staffer', info)
})
```

Note how we used *:name* in our route. That will match any string (that doesn't include a forward slash) and put it in the `req.params` object with the key `name`. This is a feature we will be using often, especially when creating a REST API. You can have multiple parameters in our route. For example, if we want to break up our staff listing by city, we can use this:

```
const staff = {
  portland: {
    mitch: { name: "Mitch", bio: 'Mitch is the man to have at your back.' },
    madeline: { name: "Madeline", bio: 'Madeline is our Oregon expert.' },
  },
  bend: {
    walt: { name: "Walt", bio: 'Walt is our Oregon Coast expert.' },
  },
}

app.get('/staff/:city/:name', (req, res, next) => {
  const cityStaff = staff[req.params.city]
  if(!cityStaff) return next()  // unrecognized city -> 404
  const info = cityStaff[req.params.name]
```

```
    if(!info) return next()       // unrecognized staffer -> 404
    res.render('staffer', info)
  })
```

Organizing Routes

It may be clear to you already that it would be unwieldy to define all of our routes in the main application file. Not only will that file grow over time, it's also not a great separation of functionality because there's a lot going on in that file already. A simple site may have only a dozen routes or fewer, but a larger site could have hundreds of routes.

So how to organize your routes? Well, how do you *want* to organize your routes? Express is not opinionated about how you organize your routes, so how you do it is limited only by your own imagination.

I'll cover some popular ways to handle routes in the next sections, but at the end of the day, I recommend four guiding principles for deciding how to organize your routes:

Use named functions for route handlers
: Writing route handlers inline by actually defining the function that handles the route right then and there is fine for small applications or prototyping, but it will quickly become unwieldy as your website grows.

Routes should not be mysterious
: This principle is intentionally vague because a large, complex website may by necessity require a more complicated organizational scheme than a 10-page website. At one end of the spectrum is simply putting *all* of the routes for your website in one single file so you know where they are. For large websites, this may be undesirable, so you break the routes out by functional areas. However, even then, it should be clear where you should go to look for a given route. When you need to fix something, the last thing you want to do is have to spend an hour figuring out where the route is being handled. I had an ASP.NET MVC project at work that was a nightmare in this respect. The routes were handled in at least 10 different places, and it wasn't logical or consistent and was often contradictory. Even though I was intimately familiar with that (very large) website, I still had to spend a significant amount of time tracking down where certain URLs were handled.

Route organization should be extensible
: If you have 20 or 30 routes now, defining them all in one file is probably fine. What about in three years when you have 200 routes? It can happen. Whatever method you choose, you should ensure you have room to grow.

Don't overlook automatic view-based route handlers

If your site consists of many pages that are static and have fixed URLs, all of your routes will end up looking like this: `app.get('/static/thing', (req, res) => res.render(\'static/thing'))`. To reduce needless code repetition, consider using an automatic view-based route handler. This approach is described later in this chapter and can be used together with custom routes.

Declaring Routes in a Module

The first step to organizing our routes is getting them all into their own module. There are multiple ways to do this. One approach is to have your module return an array of objects containing method and handler properties. Then you could define the routes in your application file thusly:

```
const routes = require('./routes.js')

routes.forEach(route => app[route.method](route.handler))
```

This method has its advantages and could be well suited to storing our routes dynamically, such as in a database or a JSON file. However, if you don't need that functionality, I recommend passing the `app` instance to the module and letting it add the routes. That's the approach we'll take for our example. Create a file called *routes.js* and move all of our existing routes into it:

```
module.exports = app => {

  app.get('/', (req,res) => app.render('home'))

  //...

}
```

If we just cut and paste, we'll probably run into some problems. For example, if we have inline route handlers that use variables or methods not available in the new context, those references will now be broken. We could add the necessary imports, but hold off on that. We'll be moving the handlers into their own module soon, and we'll solve the problem then.

So how do we link our routes in? Simple: in *meadowlark.js*, we simply import our routes:

```
require('./routes')(app)
```

Or we could be more explicit and add a named import (which we name `addRoutes` to better reflect its nature as a function; we could also name the file this way if we wanted):

```
const addRoutes = require('./routes')

addRoutes(app)
```

Grouping Handlers Logically

To meet our first guiding principle (use named functions for route handlers), we'll need somewhere to put those handlers. One rather extreme option is to have a separate JavaScript file for every handler. It's hard for me to imagine a situation in which this approach would have benefit. It's better to somehow group related functionality together. That makes it easier not only to leverage shared functionality, but also to make changes in related methods.

For now, let's group our functionality into separate files: *handlers/main.js*, where we'll put the home page handler, the "about" handler, and generally any handler that doesn't have another logical home; *handlers/vacations.js*, where vacation-related handlers will go; and so on.

Consider *handlers/main.js*:

```
const fortune = require('../lib/fortune')

exports.home = (req, res) => res.render('home')

exports.about = (req, res) => {
  const fortune = fortune.getFortune()
  res.render('about', { fortune })
}

//...
```

Now let's modify *routes.js* to make use of this:

```
const main = require('./handlers/main')

module.exports = function(app) {

  app.get('/', main.home)
  app.get('/about', main.about)
  //...

}
```

This satisfies all of our guiding principles. */routes.js* is *very* straightforward. It's easy to see at a glance what routes are in your site and where they are being handled. We've also left ourselves plenty of room to grow. We can group related functionality in as many different files as we need. And if *routes.js* ever gets unwieldy, we can use the same technique again and pass the `app` object on to another module that will in turn register more routes (though that is starting to veer into the "overcomplicated" territory—make sure you can really justify an approach that complicated!).

Automatically Rendering Views

If you ever find yourself wishing for the days of old where you could just put an HTML file in a directory and—presto!—your website would serve it, then you're not alone. If your website is content-heavy without a lot of functionality, you may find it a needless hassle to add a route for every view. Fortunately, we can get around this problem.

Let's say you want to add the file *views/foo.handlebars* and just magically have it available on the route */foo*. Let's see how we might do that. In our application file, right before the 404 handler, add the following middleware (*ch14/06-auto-views.js* in the companion repo):

```
const autoViews = {}
const fs = require('fs')
const { promisify } = require('util')
const fileExists = promisify(fs.exists)

app.use(async (req, res, next) => {
  const path = req.path.toLowerCase()
  // check cache; if it's there, render the view
  if(autoViews[path]) return res.render(autoViews[path])
  // if it's not in the cache, see if there's
  // a .handlebars file that matches
  if(await fileExists(__dirname + '/views' + path + '.handlebars')) {
    autoViews[path] = path.replace(/^\//, '')
    return res.render(autoViews[path])
  }
  // no view found; pass on to 404 handler
  next()
})
```

Now we can just add a *.handlebars* file to the *view* directory and have it magically render on the appropriate path. Note that regular routes will circumvent this mechanism (because we placed the automatic view handler after all other routes), so if you have a route that renders a different view for the route */foo*, that will take precedence.

Note that this approach will run into problems if you *delete* a view that had been visited; it will have been added to the `autoViews` object, so subsequent views will try to render it even though it's been deleted, resulting in an error. The problem could be solved by wrapping the rendering in a `try/catch` block and removing the view from `autoViews` when an error is discovered; I will leave this enhancement as a reader's exercise.

Conclusion

Routing is an important part of your project, and there are many more possible approaches to organizing your route handlers than outlined here, so feel free to experiment and find a technique that works for you and your project. I encourage you to favor techniques that are clear and easy to trace. Routing is very much a map from the outside world (the client, usually a browser) to the server-side code that responds to it. If that map is convoluted, it makes it difficult for you to trace the flow of information in your application, which will hinder both development and debugging.

CHAPTER 15

REST APIs and JSON

While we saw some REST API examples in Chapter 8, our paradigm so far has mostly been "process the data on the server side and send formatted HTML to the client." Increasingly, this is not the default mode of operation for web applications. Instead, most modern web applications are single-page applications (SPAs) that receive all of their HTML and CSS in one static bundle and then rely on receiving unstructured data as JSON and manipulating HTML directly. Similarly, the importance of posting forms to communicate changes to the server is giving way to communicating directly using HTTP requests to an API.

So it's time to turn our attention to using Express to provide API endpoints instead of preformatted HTML. This will serve us well in Chapter 16, when we demonstrate how our API could be used to dynamically render an application.

In this chapter, we'll strip down our application to providing a "coming soon" HTML interface: we'll fill that in in Chapter 16. Instead, we'll focus on an API that will provide access to our vacation database and provide API support for registering "out of season" listeners.

Web service is a general term that means any application programming interface (API) that's accessible over HTTP. The idea of web services has been around for quite some time, but until recently, the technologies that enabled them were stuffy, Byzantine, and overcomplicated. There are still systems that use those technologies (such as SOAP and WSDL), and there are Node packages that will help you interface with these systems. We won't be covering those, though. Instead, we will be focused on providing so-called RESTful services, which are much more straightforward to interface with.

The acronym *REST* stands for *representational state transfer*, and the grammatically troubling *RESTful* is used as an adjective to describe a web service that satisfies the principles of REST. The formal description of REST is complicated and steeped in computer science formality, but the basics are that REST is a stateless connection between a client and a server. The formal definition of REST also specifies that the service can be cached and that services can be layered (that is, when you use a REST API, there may be other REST APIs beneath it).

From a practical standpoint, the constraints of HTTP actually make it difficult to create an API that's not RESTful; you'd have to go out of your way to establish state, for example. So our work is mostly cut out for us.

JSON and XML

Vital to providing an API is having a common language to speak in. Part of the communication is dictated for us: we must use HTTP methods to communicate with the server. But past that, we are free to use whatever data language we choose. Traditionally, XML has been a popular choice, and it remains an important markup language. While XML is not particularly complicated, Douglas Crockford saw that there was room for something more lightweight, and JavaScript Object Notation (JSON) was born. In addition to being JavaScript-friendly (though it is by no means proprietary; it is an easy format for any language to parse), it also has the advantage of being generally easier to write by hand than XML.

I prefer JSON over XML for most applications: there's better JavaScript support, and it's a simpler, more compact format. I recommend focusing on JSON and providing XML only if existing systems require XML to communicate with your app.

Our API

We'll plan our API before we start implementing it. In addition to listing vacations and subscribing to "in-season" notifications, we'll add a "delete vacation" endpoint. Since this is a public API, we won't actually delete the vacation. We'll simply mark it as "delete requested" so an administrator can review. For example, you might use this unsecured endpoint to allow vendors to request the removal of vacations from the site, which could then later be reviewed by an administrator. Here are our API endpoints.

`GET /api/vacations`
: Retrieves vacations

`GET /api/vacation/:sku`
: Returns a vacation by its SKU

`POST /api/vacation/:sku/notify-when-in-season`
: Takes `email` as a querystring parameter and adds a notification listener for the specified vacation

`DELETE /api/vacation/:sku`

Requests the deletion of a vacation; takes `email` (the person requesting the deletion) and `notes` as querystring parameters

`[NOTE]`

Example 15-1.

There are many HTTP verbs available. `GET` and `POST` are the most common, followed by `DELETE` and `PUT`. It has become a standard to use `POST` for *creating* something, and `PUT` for *updating* (or modifying) something. The English meaning of these words doesn't support this distinction in any way, so you may want to consider using the path to distinguish between these two operations to avoid confusion. If you want more information about HTTP verbs, I recommend starting with this Tamas Piros article.

There are many ways we could have described our API. Here, we've chosen to use combinations of HTTP methods and paths to distinguish our API calls, and a mix of querystring and body parameters for passing data. As an alternative, we could have had different paths (such as */api/vacations/delete*) with the same method.[1] We could also have passed data in a consistent way. For example, we might have chosen to pass all the necessary information for retrieving parameters in the URL instead of using a querystring: `DEL /api/vacation/:id/:email/:notes`. To avoid excessively long URLs, I recommend using the request body to pass large blocks of data (for example, the deletion request notes).

There is a popular and well-respected convention for JSON APIs, creatively named JSON:API. It's a bit verbose and repetitive for my taste, but I also believe that an imperfect standard is better than no standard at all. While we're not using JSON:API for this book, you will learn everything you need to adopt the conventions laid down by JSON:API. See the JSON:API home page for more information.

API Error Reporting

Error reporting in HTTP APIs is usually achieved through HTTP status codes. If the request returns 200 (OK), the client knows the request was successful. If the request returns 500 (Internal Server Error), the request failed. In most applications, however, not everything can (or should be) categorized coarsely into "success" or "failure." For

[1] If your client can't use different HTTP methods, see this module, which allows you to "fake" different HTTP methods.

example, what if you request something by an ID but that ID doesn't exist? This does not represent a server error. The client has asked for something that doesn't exist. In general, errors can be grouped into the following categories:

Catastrophic errors
: Errors that result in an unstable or unknown state for the server. Usually, this is the result of an unhandled exception. The only safe way to recover from a catastrophic error is to restart the server. Ideally, any pending requests would receive a 500 response code, but if the failure is severe enough, the server may not be able to respond at all, and the request will time out.

Recoverable server errors
: Recoverable errors do not require a server restart, or any other heroic action. The error is a result of an unexpected error condition on the server (for example, a database connection being unavailable). The problem may be transient or permanent. A 500 response code is appropriate in this situation.

Client errors
: Client errors are a result of the client making the mistake—usually missing or invalid parameters. It isn't appropriate to use a 500 response code. After all, the server has not failed. Everything is working normally; the client just isn't using the API correctly. You have a couple of options here: you could respond with a status code of 200 and describe the error in the response body, or you could additionally try to describe the error with an appropriate HTTP status code. I recommend the latter approach. The most useful response codes in this case are 404 (Not Found), 400 (Bad Request), and 401 (Unauthorized). Additionally, the response body should contain an explanation of the specifics of the error. If you want to go above and beyond, the error message would even contain a link to documentation. Note that if the user requests a list of things and there's nothing to return, this is not an error condition. It's appropriate to simply return an empty list.

In our application, we'll be using a combination of HTTP response codes and error messages in the body.

Cross-Origin Resource Sharing

If you're publishing an API, you'll likely want to make the API available to others. This will result in a *cross-site HTTP request*. Cross-site HTTP requests have been the subject of many attacks and have therefore been restricted by the *same-origin policy*, which restricts where scripts can be loaded from. Specifically, the protocol, domain, and port must match. This makes it impossible for your API to be used by another site, which is where cross-origin resource sharing (CORS) comes in. CORS allows you to lift this restriction on a case-by-case basis, even allowing you to list which domains specifically are allowed to access the script. CORS is implemented through the `Access-Control-Allow-Origin` header. The easiest way to implement it in an Express application is to use the `cors` package (`npm install cors`). To enable CORS for your application, use this:

```
const cors = require('cors')

app.use(cors())
```

Because the same-origin API is there for a reason (to prevent attacks), I recommend applying CORS only where necessary. In our case, we want to expose our entire API (but only the API), so we're going to restrict CORS to paths starting with */api*:

```
const cors = require('cors')

app.use('/api', cors())
```

See the package documentation for information about more advanced use of CORS.

Our Tests

If we use HTTP verbs other than `GET`, it can be a hassle to test our API, since browsers only know how to issue `GET` requests (and `POST` requests for forms). There are ways around this, such as the excellent application Postman. However, whether or not you use such a utility, it's good to have automated tests. Before we write tests for our API, we need a way to actually *call* a REST API. For that, we'll be using a Node package called `node-fetch`, which replicates the browser's *fetch* API:

```
npm install --save-dev node-fetch@2.6.0
```

We'll put the tests for the API calls we're going to implement in *tests/api/api.test.js* (*ch15/test/api/api.test.js* in the companion repo):

```
const fetch = require('node-fetch')

const baseUrl = 'http://localhost:3000'

const _fetch = async (method, path, body) => {
  body = typeof body === 'string' ? body : JSON.stringify(body)
  const headers = { 'Content-Type': 'application/json' }
  const res = await fetch(baseUrl + path, { method, body, headers })
  if(res.status < 200 || res.status > 299)
    throw new Error(`API returned status ${res.status}`)
  return res.json()
}

describe('API tests', () => {

  test('GET /api/vacations', async () => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    expect(vacation0.name).toMatch(/\w/)
    expect(typeof vacation0.price).toBe('number')
  })

  test('GET /api/vacation/:sku', async() => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    const vacation = await _fetch('get', '/api/vacation/' + vacation0.sku)
    expect(vacation.name).toBe(vacation0.name)
  })

  test('POST /api/vacation/:sku/notify-when-in-season', async() => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    // at this moment, all we can do is make sure the HTTP request is successful
    await _fetch('post', `/api/vacation/${vacation0.sku}/notify-when-in-season`,
      { email: 'test@meadowlarktravel.com' })
  })

  test('DELETE /api/vacation/:id', async() => {
    const vacations = await _fetch('get', '/api/vacations')
    expect(vacations.length).not.toBe(0)
    const vacation0 = vacations[0]
    // at this moment, all we can do is make sure the HTTP request is successful
    await _fetch('delete', `/api/vacation/${vacation0.sku}`)
  })

})
```

Our test suite starts off with a helper function `_fetch`, which handles some common housekeeping. It will JSON encode the body if it isn't already, add the appropriate headers, and throw an appropriate error if the response status code isn't in the 200s.

We have a single test for each of our API endpoints. I'm not suggesting that these tests are robust or complete; even with this simple API, we could (and should) have several tests for each endpoint. What we have here is more of a starting point that illustrates techniques for testing an API.

There are a couple of important characteristics of these tests that deserve mention. One is that we are relying on the API being already started and running on port 3000. A more robust test suite would find an open port, start the API on that port as part of its setup, and stop it when all the tests have run. Second, this test relies on data already being present in our API. For example, the first test expects there to be at least one vacation, and for that vacation to have a name and a price. In a real application, you may not be able to make these assumptions (for example, you may start with no data, and you may want to test for allowable missing data). Again, a more robust testing framework would have a way of setting and resetting the initial data in the API so you could start from a known state every time. For example, you might have scripts that set up and seed a test database, attach the API to it, and tear it down for every test run. As we saw in Chapter 5, testing is a large and complicated topic, and we can only scratch the surface here.

The first test covers our `GET /api/vacations` endpoint. It fetches all of the vacations, validates that there is at least one, and checks the first one to see if it has a name and a price. We could also conceivably test other data properties. I'll leave it as a reader's exercise to think about which properties are most important to test.

The second test covers our `GET /api/vacation/:sku` endpoint. Since we don't have consistent test data, we start by fetching all of the vacations and getting the SKU from the first one so we can test this endpoint.

Our last two tests cover our `POST /api/vacation/:sku/notify-when-in-season` and `DELETE /api/vacation/:sku` endpoints. Unfortunately, with our current API and testing framework, we can do very little to verify that these endpoints are doing what they are supposed to, so we default to invoking them and trusting the API is doing the right thing when it doesn't return an error. If we wanted to make these tests more robust, we would have to either add endpoints that allow us to verify the actions (for example, an endpoint that determined if a given email was registered for a specific vacation) or somehow give the tests "backdoor" access to our database.

If you run the tests now, they will time out and fail…because we haven't implemented our API or even started our server. So let's get started!

Using Express to Provide an API

Express is quite capable of providing an API. There are various npm modules available that provide helpful functionality (see `connect-rest` and `json-api`, for example), but I find that Express is perfectly capable out of the box, and we'll be sticking with a pure Express implementation.

We'll start by creating the handlers in *lib/handlers.js* (we could create a separate file, such as *lib/api.js*, but let's keep things simple for now):

```
exports.getVacationsApi = async (req, res) => {
  const vacations = await db.getVacations({ available: true })
  res.json(vacations)
}

exports.getVacationBySkuApi = async (req, res) => {
  const vacation = await db.getVacationBySku(req.params.sku)
  res.json(vacation)
}
```

```
exports.addVacationInSeasonListenerApi = async (req, res) => {
  await db.addVacationInSeasonListener(req.params.sku, req.body.email)
  res.json({ message: 'success' })
}

exports.requestDeleteVacationApi = async (req, res) => {
  const { email, notes } = req.body
  res.status(500).json({ message: 'not yet implemented' })
}
```

Then we hook up the API in *meadowlark.js*:

```
app.get('/api/vacations', handlers.getVacationsApi)
app.get('/api/vacation/:sku', handlers.getVacationBySkuApi)
app.post('/api/vacation/:sku/notify-when-in-season',
  handlers.addVacationInSeasonListenerApi)
app.delete('/api/vacation/:sku', handlers.requestDeleteVacationApi)
```

Nothing here should be particularly surprising by now. Note that we're using our database abstraction layer, so it doesn't matter if we use our MongoDB implementation or our PostgreSQL implementation (though you will find minor inconsequential extra fields depending on the implementation, which we could remove if necessary).

I am leaving `requestDeleteVacationsApi` as a reader's exercise, mainly because this functionality could be implemented so many different ways. The simplest approach would be to just modify our vacation schema to have "delete requested" fields that just get updated with the email and notes when the API is called. A more sophisticated approach would be to have a separate table, like a moderation queue, that records the deletion requests separately, referencing the vacation in question, which would better lend itself to administrator use.

Assuming you set up Jest correctly in Chapter 5, you should just be able to run `npm test`, and the API tests will be picked up (Jest will look for any file that ends in `.test.js`). You'll see we have three passing tests and one failing one: the incomplete `DELETE /api/vacation/:sku`.

Conclusion

I hope this chapter has left you asking, "That's it?" At this point, you're probably realizing that the primary function of Express is to respond to HTTP requests. What the requests are for—and how they respond—is entirely up to you. Do they need to respond with HTML? CSS? Plain text? JSON? All easy to do with Express. You could even respond with binary file types. For example, it would not be hard to dynamically construct and return images. In this sense, an API is just another one of the many ways Express can respond.

In the next chapter, we'll put this API to use by building a single-page application, and replicate what we've done in previous chapters in a different way.

CHAPTER 16

Single-Page Applications

The term *single-page application* (SPA) is something of a misnomer, or it is at least confusing two meanings of the word "page." SPAs, from the user's perspective, can (and usually do) still appear to have different pages: the home page, the Vacations page, the About page, and so on. As a matter of fact, you could create a traditional server-side rendered application and an SPA that were indistinguishable to the user.

The "single page" has more to do with where and how the HTML is constructed than the user's experience. In an SPA, the server delivers a single HTML bundle when the user first loads the application,[1] and any changes in the UI (which may appear as different pages to the user) are the result of JavaScript manipulating the DOM in response to user activity or network events.

SPAs still need to communicate frequently with the server, but HTML is usually only sent as part of that first request. After that, only JSON data and static assets are transferred between the client and server.

Understanding the reason for this now-dominant approach to web application development requires a little history….

A Short History of Web Application Development

The way we approach web development has undergone a massive shift in the last 10 years, but one thing has remained relatively consistent: the components involved in a website or web application. Namely:

[1] For performance reasons, the bundle might be split into "chunks" that are loaded as needed (called *lazy loading*), but the principle is the same.

- HTML and the Document Object Model (DOM)
- JavaScript
- CSS
- Static assets (generally multimedia: images and videos, etc.)

Put together by a browser, these components are what provide the user experience.

How that experience is constructed, however, started shifting drastically around 2012. Today, the dominant paradigm for web development is *single-page applications*, or SPAs.

To understand SPAs, we need to understand what to contrast them with, so we're going to go even further back in time, to 1998, the year before the term "Web 2.0" was first whispered, and eight years before jQuery was introduced.

In 1998, the dominant method for delivering web applications was for web servers to send HTML, CSS, JavaScript, and multimedia assets *in response to every request*. Imagine you're watching TV, and you want to change the channel. The metaphorical equivalent here is that you would have to throw away your TV, go buy another one, schlep it into your house, and set it up—just to change the channel (navigate to a different page, even on the same site).

The problem with this approach is that there's a lot of overhead involved. Sometimes the HTML—or large chunks of it—wouldn't change at all. The CSS changed even less. Browsers mitigated some of this overhead cost by caching assets, but the pace of innovation in web applications was straining this model.

In 1999, the term "Web 2.0" was coined to try to describe the richness of experience that people were beginning to expect from websites. The years between 1999 and 2012 saw technological advancements that were laying the groundwork for SPAs.

Clever web developers began to realize that if they were going to keep their users engaged, the overhead of shipping the entire website every time the user wanted to (metaphorically) change the channel was unacceptable. These developers realized that not every change in an application required information from the server, and not every change that required information from the server needed the entire application just to deliver a small change.

In this period from 1999 to 2012, pages were still generally pages: when you first went to a website, you got the HTML, the CSS, and the static assets. When you navigated to a different page, you would get different HTML, different static assets, and sometimes different CSS. However, on each page, the page itself might change in response to user interaction, and instead of asking the server for a whole new application, JavaScript would change the DOM directly. If information needed to be fetched from the server, that information was sent in XML or JSON, without all the attendant HTML.

It was, once again, up to the JavaScript to interpret the data and change the user interface accordingly. In 2006, jQuery was introduced, which significantly eased the burden of DOM manipulation *and* dealing with network requests.

Many of these changes were being driven by the increasing power of computers and —by extension—browsers, Web developers were finding that more and more of the work to make a website or web application look pretty could be done directly on the user's computer instead of being done on the server and then sent to the user.

This shift in approach went into overdrive in the late 2000s, when smartphones were introduced. Now, not only were browsers capable of doing more, but people wanted to access web applications *over wireless networks*. Suddenly, the overhead cost of sending data went up, making it even more attractive to ship as little as possible over the network, and let the browser do as much work as possible.

By 2012, it was common practice to try to send as little information as possible over the network, and do as much as possible in the browser. Like the primordial soup giving rise to the first life, this rich environment provided the conditions for the natural evolution of the this technique: the single-page application.

The idea is simple enough: for any given web application, the HTML, JavaScript, and CSS (if any) are shipped *exactly once*. Once the browser has the HTML, it is up to the JavaScript to make all changes to the DOM to make the user feel that they are navigating to a different page. No more does the server need to send different HTML when you navigate from the home page to the Vacations page, for example.

Of course the server is still involved: it's still responsible for providing up-to-date data, and being the "single source of truth" in a multiuser application. But in an SPA architecture, the way the application appears to the user is no longer the concern of the server: it's the concern of JavaScript and the frameworks that enable this clever illusion.

While Angular is generally considered the first SPA framework, it has been joined by many others: React, Vue, and Ember being the most prominent among Angular's competition.

If you are new to development, SPAs may be your only frame of reference, making this simply some interesting history. But if you're a veteran, you may find the shift confusing and jarring. Whichever group you fall into, this chapter is designed to help you understand how web applications are delivered as SPAs, and what the role of Express is in that.

This history is relevant to Express because the role of the server has changed during this shift in web development techniques. When the first edition of this book was published, Express was still commonly used to serve multi-page applications (along with the APIs that supported Web 2.0–like functionality). Now Express is almost entirely used to serve SPAs, development servers, and APIs, reflecting the changing nature of web development.

Interestingly, there are still valid reasons for a web application to be able to serve a specific page (instead of the "generic" page, which will be reformatted by the browser). While this may seem like we are coming full-circle, or throwing away the gains of SPAs, the technique to do this better mirrors the architecture of SPAs. Called *server-side rendering* (SSR), this technique allows the servers to use the same code that the browser uses to create individual pages to increase first-page load. The key here is that the server doesn't have to do much thinking: it simply uses the same techniques as the browser to generate a specific page. This kind of SSR is usually done to enhance first-page loading experience, and to support search engine optimization. It's a more advanced topic that we won't be covering here, but you should be aware of the practice.f1603.450

Now that we have some insight into how and why SPAs came into being, let's look at the SPA frameworks that are available today.

SPA Technologies

There are many choices for SPA technologies now:

React
: For the moment, React seems to be the king of the SPA hill, though there are former greats (Angular) on one side of it, and ambitious usurpers (Vue) on the other side. Sometime in 2018, React surpassed Angular in usage statics. React is an open source library, but it started its life as a Facebook project, and Facebook is still an active contributor. We'll be using React for our Meadowlark Travel refactor.

Angular
: By most accounts, the "original" SPA, Google's Angular became massively popular but was eventually dethroned by React. In late 2014, Angular announced version 2, which was a massive change from the first version, and alienated many existing users and scared off new ones. I believe this shift (while probably necessary) contributed to React eventually outpacing Angular. Another reason is that Angular is a much larger framework than React. This has advantages and disadvantages: Angular provides a much more complete architecture for building full applications, and there's always a clear "Angular way" to do things, whereas frameworks like React and Vue leave a lot more up to personal choice and creativity. Regardless of which approach is better, bigger frameworks are more ponderous and slow to evolve, which gave React an innovation edge.

Vue.js

An upstart challenger to React, and the brainchild of a single developer, Evan You. In a remarkably short time, it has gained an impressive following, and it is extremely well-liked by its adherents, but it is still far behind React's runaway popularity. I have had some experience with Vue, and I appreciate its clear documentation and lightweight approach, but I have come to prefer React's architecture and philosophy.

Ember

Like Angular, Ember offers a comprehensive application framework. There's a large and active development community and, while not as innovative as React or Vue, it offers a lot of functionality and clarity. I have found I far prefer lighter frameworks, and have stuck with React for this reason.

Polymer

I have no experience with Polymer, but it is backed by Google, which lends it credibility. People seem to be curious about what Polymer is bringing to the table, but I haven't seen a lot of people rushing to adopt it.

If you're looking for a robust out-of-the-box framework, and you don't mind coloring within the lines, you should consider Angular or Ember. If you want room for creative expression and innovation, I recommend React or Vue. I don't yet know where Polymer fits in yet, but it's worth keeping an eye on.

Now that we've seen the players, let's move forward with React, and refactor Meadowlark Travel as an SPA!

Creating a React App

The best way to get started with a React app is to use the `create-react-app` (CRA) utility, which creates all of the boilerplate, developer tooling, and provides a minimal starter application that you can build on. Furthermore, `create-react-app` will keep its configuration up-to-date so you can focus on building your application instead of on framework tooling. That said, if you ever reach the point where you need to configure your tooling, you can "eject" your application: you'll lose the ability to keep up-to-date with the latest CRA tooling, but you'll have full control over all of the application configuration.

Unlike what we've been doing so far, where all of our application artifacts lived alongside our Express application, SPAs are best thought of as a completely separate, independent application. To that end, we'll have *two* application roots instead of one. For clarity, when I'm referring to the directory where your Express application lives, I'll say the *server root*, and for the directory where your React application lives, I'll say the *client root*. The *application root* is where both of those directories now live.

So go to your application root and create a directory called *server*; this is where your Express server will live. Don't create a directory for your client app; CRA will do that for us.

Before we run CRA, we should install Yarn. Yarn is a package manager like npm… actually, yarn is mostly a drop-in replacement for npm. It's not mandatory for React development, but it is the de facto standard, and not using it would be swimming upstream. There are some minor differences in usage between Yarn and npm, but the only one you'll probably notice is that you run `yarn add` instead of `npm install`. To install Yarn, simply follow the Yarn installation instructions.

Once you've installed Yarn, run the following from your application root:

```
yarn create react-app client
```

Now go into your client directory and type `yarn start`. After a few seconds, you'll see a new browser window pop up, with your React app running in it!

Go ahead and leave the terminal window running. CRA has really good support for "hot reloading," so when you make changes in your source code, it will get built *very* quickly and the browser will automatically reload. Once you get used to it, you won't be able to live without it.

React Basics

React has excellent documentation, which I won't re-create here. So if you're new to React, start with the Intro to React tutorial, and then the Main Concepts guide.

You'll find that React is organized around *components*, which are the main building blocks of React. Everything the user sees or interacts with is generally a component in React. Let's take a look at *client/src/App.js* (the contents of yours may differ slightly—CRA does change over time):

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

One of the core concepts in React is that the UI is generated by *functions*. And the simplest React component is just a function that returns HTML, as we see here. You may be looking at this and thinking that it isn't valid JavaScript; it looks like HTML is mixed in! The reality is a little more complicated. React, by default, enables a superset of JavaScript called JSX. JSX allows you to write what looks like HTML. It's not *actually* HTML; it creates React elements, and the purpose of a React element is to (eventually) correspond to a DOM element.

At the end of the day, however, you can think of it as HTML. Here, `App` is a function that will render the HTML corresponding to the JSX it returns.

A couple of things to note: since JSX is close to—but not exactly—HTML, there are some subtle differences. You may have already noticed we use `className` instead of `class`, which is because `class` is a reserved word in JavaScript.

All you have to do to specify HTML is to start an HTML element anywhere an expression is expected. You can also "go back to" JavaScript with curly braces within the HTML. For example:

```
const value = Math.floor(Math.random()*6) + 1
const html = <div>You rolled a {value}!</div>
```

In this example, the `<div>` starts the HTML, and the curly brackets around `value` drop back into JavaScript to provide the number stored in `value`. We could have just as easily inlined the calculation:

```
const html = <div>You rolled a {Math.floor(Math.random()*6) + 1}!</div>
```

Any valid JavaScript expression can be contained within curly brackets within JSX—including other HTML elements! A common use case of this is rendering lists:

```
const colors = ['red', 'green', 'blue']
const html = (
  <ul>
    {colors.map(color =>
      <li key={color}>{color}</li>
    )}
  </ul>
)
```

A couple of things to note about this example. First, note that we mapped over our colors to return the `<li>` elements. This is critical: JSX works entirely by evaluating *expressions*. So the `<ul>` has to contain either an expression or an array of expressions. If you changed the `map` to a `forEach`, you would find that the `<li>` elements would not get rendered. Second, note that the `<li>` elements receive a property `key`: this is a performance concession. For React to know when to re-render the elements in an array, it needs a unique key for each element. Since our array elements are unique, we just used that value, but commonly you would use a an ID or—if nothing else is available—the index of the item in the array.

I encourage you to play around with some of these examples in the JSX in *client/src/App.js* before moving on. If you've left `yarn start` running, every time you save your changes, they will be automatically reflected in the browser, which should speed up your learning cycle.

We have one more topic to touch on before we move on from React basics, and that concept is *state*. Every component can have its own state, which basically means that "data associated with the component that can change." A shopping cart is a great example of this. A shopping cart component's state would contain a list of items; as you add and remove items from the cart, the component's state is changing. It may seem like an overly simple or obvious concept, but most of the details of making a React application come down to effectively designing and managing the state of your components. We'll see an example of state when we tackle the Vacations page.

Let's move on and create our Meadowlark Travel home page.

The Home Page

Recall from our Handlebars views that we had a main "layout" file that established the primary look and feel of our website. Let's start by focusing on what's in the `<body>` tag (except the scripts):

```
<div class="container">
  <header>
    <h1>Meadowlark Travel</h1>
    <a href="/"><img src="/img/logo.png" alt="Meadowlark Travel Logo"></a>
  </header>
  {{{body}}}
</div>
```

This will be pretty easy to refactor into a React component. First, we copy our own logo into the *client/src* directory. Why not the *public* directory? For small or commonly used graphical items, it may be more efficient to inline them in the JavaScript bundle, and the bundler that you got with CRA will make an intelligent choice about that. The example app you got from CRA placed its logo directly in the *client/src* directory, but I still like collecting image assets in a subdirectory, so put our logo (*logo.png*) in *client/src/img/logo.png*.

The only other tricky bit is what to do about {{{body}}}? In our views, this is where another view would be rendered—the content for the specific page you're on. We can replicate the same basic idea in React. Since all content is rendered in the form of components, we're just going to render another component here. We'll start with an empty Home component and build that out in a moment:

```
import React from 'react'
import logo from './img/logo.png'
import './App.css'

function Home() {
  return (<i>coming soon</i>)
}

function App() {
  return (
    <div className="container">
      <header>
        <h1>Meadowlark Travel</h1>
        <img src={logo} alt="Meadowlark Travel Logo" />
      </header>
      <Home />
    </div>
  )
}

export default App
```

We're using the same approach that the sample app did for CSS: we can simply create a CSS file and import it. So we can edit that file and apply whatever styles we need to. We'll keep things basic for this example, though nothing fundamental has changed in the way we style HTML with CSS, so we still have all the tools we're used to.

CRA sets up linting for you, and as you progress through this chapter, you'll probably see warnings (both in the CRA terminal output and in your browser's JavaScript console). This is only because we're adding things incrementally; by the time we reach the end of this chapter, there should be no more warnings…if there are, make sure you haven't missed a step! You can also check the companion repository.

Routing

The core concept of routing we learned about in Chapter 14 hasn't changed: we're still using the URL path to determine what part of the interface the user is seeing. The difference is that it's up to the client application to handle that. Changing the UI based on the route is the client app's responsibility: if the navigation requires new or updated data from the server, that's fine, and it's up to the client app to request that from the server.

There are a lot of options for—and a lot of strong opinions about—routing in React apps. However, there is a dominant library for routing: React Router. There's quite a lot I don't like about React Router, but it's so common that you're bound to come across it. Furthermore, it is a good option to get something basic up and running, and for those two reasons, we'll be using it here.

We'll get started by installing the DOM version of React Router (there's also a version for React Native, for mobile development):

```
yarn add react-router-dom
```

Now we'll hook up the router, and add an About and a Not Found page. We'll also link the site logo back to the home page:

```
import React from 'react'
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from 'react-router-dom'
import logo from './img/logo.png'
import './App.css'

function Home() {
  return (
    <div>
      <h2>Welcome to Meadowlark Travel</h2>
      <p>Check out our "<Link to="/about">About</Link>" page!</p>
    </div>
  )
}

function About() {
  return (<i>coming soon</i>)
}

function NotFound() {
  return (<i>Not Found</i>)
}

function App() {
  return (
    <Router>
      <div className="container">
        <header>
```

```
          <h1>Meadowlark Travel</h1>
          <Link to="/"><img src={logo} alt="Meadowlark Travel Logo" /></Link>
        </header>
        <Switch>
          <Route path="/" exact component={Home} />
          <Route path="/about" exact component={About} />
          <Route component={NotFound} />
        </Switch>
      </div>
    </Router>
  )
}

export default App
```

The first thing to notice is that we're wrapping our entire application in a `<Router>` component. This is what enables the routing, as you might expect. Inside `<Router>`, we can use `<Route>` to conditionally render a component based on the URL path. We've placed our content routes inside a `<Switch>` component: this ensures that only *one* of the components contained therein gets rendered.

There are some subtle differences between the routing we've done with Express and React Router. In Express, we would render the page according to the first successful match (or the 404 page if one couldn't be found). With React Router, the path is simply a "hint" to determine what combination of components should display. In this way, it's more flexible than routing with Express. Because of this, React Router routes behave by default as if they have an asterisk (`*`) at the end. That is, the route `/` would, by default, match *every* page (since they all start with a forward slash). Because of this, we use the `exact` property to make this route behave more like an Express route. Similarly, without the `exact` property, the `/about` route would also match `/about/contact`, which is probably not what we want. For your main content routing, it's likely that you'll want all of your routes (except the Not Found route) to have `exact`. Otherwise, you will have to make sure to arrange them correctly within the `<Switch>` so they match in the correct order.

The second thing to notice is the use of `<Link>`. You might be wondering why we don't just use `<a>` tags. The problem with `<a>` tags is that—without some extra work—the browser will dutifully treat them as "going elsewhere" even if it's on the same site, and it will result in a new HTTP request to the server…and the HTML and the CSS will be downloaded again, defeating the SPA agenda. It will *work* in the sense that when the page loads, React Router will do the right thing, but it won't be as fast or efficient, invoking unnecessary network requests. Seeing the difference is actually an instructive exercise that should drive home the nature of SPAs. As an experiment, create two navigation elements, one using `<Link>` and another using `<a>`:

```
<Link to="/">Home (SPA)</Link>
<a href="/">Home (reload)</Link>
```

Then open your dev tools, open the Network tab, clear the traffic, and click "Preserve log" (on Chrome). Now click the "Home (SPA)" link and notice there's no network traffic at all. Click the "Home (reload)" link and observe the network traffic. And that, in a nutshell, is the nature of an SPA.

Vacations Page—Visual Design

So far we've just been building a pure frontend application…so where does Express come in? Our server is still the single source of truth. In particular, it maintains the database of vacations that we want to display on our site. Fortunately, we've already done most of the work in Chapter 15: we exposed an API that will return our vacations in JSON format, ready for use in a React application.

Before we hook those two things up, however, let's go ahead and build our Vacations page. There won't be any vacations to render, but let's not let that stop us.

In the previous section, we included all of the content pages in *client/src/App.js*, which is generally considered poor practice: its more conventional for each component to live in its own file. So we'll take the time to break our `Vacations` component out into its own component. Create the file *client/src/Vacations.js*:

```
import React, { useState, useEffect } from 'react'
import { Link } from 'react-router-dom'

function Vacations() {
  const [vacations, setVacations] = useState([])
  return (
    <>
      <h2>Vacations</h2>
      <div className="vacations">
        {vacations.map(vacation =>
          <div key={vacation.sku}>
            <h3>{vacation.name}</h3>
            <p>{vacation.description}</p>
            <span className="price">{vacation.price}</span>
          </div>
        )}
      </div>
    </>
  )
}

export default Vacations
```

What we have so far is pretty simple: we're just returning a `<div>` that contains additional `<div>` elements, each of which represents a vacation. So where is this `vacations` variable coming from? In this example, we're using a newer feature of React, called *React hooks*. Prior to hooks, if a component wanted to have its own state (in this case, a list of vacations), you had to use a class implementation. Hooks enable us to have function-based components that have their own state. In our `Vacations` function, we call `useState` to set up our state. Note we pass an empty array to `useState`: that will be the initial value of `vacations` in state (we'll discuss how we populate that shortly). What `setState` returns is an array containing the state value itself (`vacations`) and a way to update the state (`setVacations`).

You may wonder why we can't modify `vacations` directly: it's just an array, so couldn't we call `push` to add vacations to it? We could, but this would be defeating the very purpose of React's state management system, which ensures consistency, performance, and communication between components.

You may also be wondering about what looks like an empty component (`<>...</>`) surrounding our vacations. This is called a *fragment*. The fragment is necessary because every component must render a single element. In our case, we have two elements, the `<h2>` and the `<div>`. The fragment simply provides a "transparent" root element in which to contain these two elements so we can render a single element.

Let's add our `Vacations` component to our application, even though there aren't yet any vacations to show. In *client/src/App.js*, first import your vacations page:

```
import Vacations from './Vacations'
```

Then all we have to do is create a route for it in our router's `<Switch>` component:

```
<Switch>
  <Route path="/" exact component={Home} />
  <Route path="/about" exact component={About} />
  <Route path="/vacations" exact component={Vacations} />
  <Route component={NotFound} />
</Switch>
```

Go ahead and save that; your application should automatically reload, and you can navigate to your */vacations* page, though there isn't much interesting to see yet. Now that we have most of the client infrastructure in place, let's turn our attention to integrating with Express.

Vacations Page—Server Integration

We've already done most of the work necessary for the Vacations page; we have an API endpoint that gets vacations from the database and returns them in JSON format. Now we have to figure out how to get the server and the client communicating.

We can start with our work from Chapter 15; we don't need to add anything to it, but we can take some things away that we no longer need. We can remove the following:

- Handlebars and views support (we'll leave the static middleware, though, for reasons we'll see later).

- Cookies and sessions (our SPA may still use cookies, but it no longer needs the server's help here…and we think about sessions in a completely different way).
- All routes that render a view (we obviously keep the API routes, however).

This leaves us with a much simplified server. So what do we do with it now? The first thing we have to do is address the fact that we've been using port 3000, and the CRA development server also uses port 3000 by default. We could change either, so I'm going to arbitrarily suggest changing the Express port. I usually use 3033—just because I like the sound of that number. You'll recall that we set the default port in our *meadowlark.js*, so we just need to change it:

```
const port = process.env.PORT || 3033
```

We could, of course, use an environment variable to control it, but since we're going to frequently use it together with our SPA dev server, we might as well change the code.

Now that both servers are running, we can communicate between them. But how? In our React app, we could do something like this:

```
fetch('http://localhost:3033/api/vacations')
```

The problem with that approach is that we're going to be making requests like that all over our application…and now we're embedding `http://localhost:3033` all over the place…which isn't going to work in production, and it may not work on your colleague's computer because maybe it needs to use different ports, and maybe the port needs to be different for the testing servers…and on and on. Using this approach is asking for a configuration headache. Yes, you could store the base URL as a variable that you use everywhere, but there's a better way.

In the ideal world, from your application's perspective, everything's hosted from the same place: it's the same protocol, host, and port to get the HTML, the static assets, and the API. It simplifies a lot of things and ensures consistency in your source code. If everything's coming from the same place, you can simply omit the protocol, host and port, and just call `fetch(/api/vacations)`. It's a nice approach, and fortunately very easy to do!

The configuration for CRA comes with *proxy* support, allowing you to pass web requests on to your API. Edit your *client/package.json* file, and add the following:

```
"proxy": "http://localhost:3033",
```

It doesn't matter where you add it. I usually put it between `"private"` and `"dependencies"` just because I like to see it high in the file. Now—as long as your Express server is running on port 3033—your CRA development server will pass API requests through to your Express server.

Now that that configuration is in place, let's use an *effect* (another React hook) to fetch and update vacation data. Here's the entire Vacations component with the useEffect hook:

```
function Vacations() {
  // set up state
  const [vacations, setVacations] = useState([])

  // fetch initial data
  useEffect(() => {
    fetch('/api/vacations')
      .then(res => res.json())
      .then(setVacations)
  }, [])

  return (
    <>
      <h2>Vacations</h2>
      <div className="vacations">
        {vacations.map(vacation =>
          <div key={vacation.sku}>
            <h3>{vacation.name}</h3>
            <p>{vacation.description}</p>
            <span className="price">{vacation.price}</span>
          </div>
        )}
      </div>
    </>
  )
}
```

As before, useState is configuring our component state to have a vacations array, with a companion setter. Now we've added useEffect, which calls our API to retrieve vacations, and then calls that setter asynchronously. Note that we pass in an empty array as the second argument to useEffect; this is a signal to React that this effect should be run only once, when the component is mounted. On the surface, that may seem like an odd way to signal that, but once you learn more about hooks, you'll see that it's actually quite consistent. To learn more about hooks, see the React hooks documentation.

Hooks are relatively new—they were added in version 16.8 in February 2019—so even if you have some experience with React, you may not be familiar with hooks. I firmly believe that hooks are an excellent innovation in the React architecture, and, while they may seem alien at first, you'll find that they actually simplify your components and reduce some of the trickier state-related mistakes that people commonly make.

Now that we've learned how to retrieve data from the server, let's turn our attention to sending information the other way.

Sending Information to the Server

We already have an API endpoint to make changes on the server; we have an endpoint to be emailed when is back in season. Let's go ahead and modify our `Vacations` component to show a sign-up form for vacations that are out of season. In true React fashion, we'll create two new components: we'll break out the individual vacation view into `Vacation` and a `NotifyWhenInSeason` component. We could do it all in one, but the recommended approach to React development is to have many specific-purpose components instead of gigantic multipurpose components (for the sake of brevity, however, we are going to stop short of putting these components in their own files: I'll leave that as a reader's exercise):

```
import React, { useState, useEffect } from 'react'

function NotifyWhenInSeason({ sku }) {
  return (
    <>
      <i>Notify me when this vacation is in season:</i>
      <input type="email" placeholder="(your email)" />
      <button>OK</button>
    </>
  )
}

function Vacation({ vacation }) {
  return (
    <div key={vacation.sku}>
      <h3>{vacation.name}</h3>
      <p>{vacation.description}</p>
      <span className="price">{vacation.price}</span>
      {!vacation.inSeason &&
        <div>
          <p><i>This vacation is not currently in season.</i></p>
          <NotifyWhenInSeason sky={vacation.sku} />
        </div>
      }
    </div>
  )
}

function Vacations() {
  const [vacations, setVacations] = useState([])
  useEffect(() => {
    fetch('/api/vacations')
      .then(res => res.json())
      .then(setVacations)
  }, [])
  return (
    <>
      <h2>Vacations</h2>
      <div className="vacations">
        {vacations.map(vacation =>
          <Vacation key={vacation.sku} vacation={vacation} />
        )}
      </div>
    </>
  )
}

export default Vacations
```

Now, if you have any vacations that have `inSeason` as `false` (and you will, unless you changed your database or initialization scripts), you will update the form. Now let's hook up our button to make the API call. Modify `NotifyWhenInSeason`:

```
function NotifyWhenInSeason({ sku }) {
  const [registeredEmail, setRegisteredEmail] = useState(null)
  const [email, setEmail] = useState('')
  function onSubmit(event) {
    fetch(`/api/vacation/${sku}/notify-when-in-season`, {
        method: 'POST',
        body: JSON.stringify({ email }),
        headers: { 'Content-Type': 'application/json' },
      })
      .then(res => {
        if(res.status < 200 || res.status > 299)
          return alert('We had a problem processing this...please try again.')
        setRegisteredEmail(email)
      })
    event.preventDefault()
  }
  if(registeredEmail) return (
    <i>You will be notified at {registeredEmail} when
    this vacation is back in season!</i>
  )
  return (
    <form onSubmit={onSubmit}>
      <i>Notify me when this vacation is in season:</i>
      <input
        type="email"
        placeholder="(your email)"
        value={email}
        onChange={({ target: { value } }) => setEmail(value)}
        />
      <button type="submit">OK</button>
    </form>
  )
}
```

We're choosing here to have the component track two different values: the email address as the user types it, and the final value after they press OK. The former is a technique known as *controlled components*, and you can read more about it on the

React forms documentation. The latter we're keeping track of so we can know when the user took the action of pressing OK so we can change the UI accordingly. We could have also had a simple boolean "registered," but this allows our UI to remind the user what email they registered with.

We also had to do a little more work with our API communication: we had to specify the method (`POST`), encode the body as JSON, and specify the content type.

Note that we make a decision about which UI to return. If the user has already registered, we return a simple message, and if they haven't, we render the form. This is a very common pattern in React.

Whew! It seems like a lot of work for that small bit of functionality…and pretty crude functionality at that. Our error-handling if there's something wrong with the API call is functional, but less than user-friendly, and while the component will remember which vacations we've signed up for, it will do so only while we're on this page. If we navigate away and come back, we'll see the form again.

There are steps we could take to make this code a little more palatable. For starters, we might write an API wrapper that will handle the messy details of encoding input and determining errors; that will certainly pay dividends as we use more and more API endpoints. There are also many popular form-processing frameworks for React that go a long way to ease the burden of form processing.

Addressing the problem of "remembering" what vacations the user has signed up for is a little trickier. What would really serve us would be a way for our vacation objects to have that information available (whether or not the user had registered). However, our special-purpose component doesn't know anything about the vacation; it's only given the SKU. In the next section, we'll talk about *state management*, which points to a solution to that problem.

State Management

Most of the architectural work that goes into planning and designing a React application is focused around state management—and not usually the state management of single components, but how they share and coordinate state. Our sample application does share some state: the `Vacations` component passes down a vacation object to the `Vacation` component, and the `Vacation` component in turn passes down the vacation's SKU to the `NotifyWhenInSeason` listener. But so far, our information is only flowing *down* the tree; what happens when information needs to go back *up*?

The most common approach is to pass functions around that are responsible for updating state. For example, the `Vacations` component might have a function for modifying a vacation, which it could pass to `Vacation`, which could in turn be passed down to `NotifyWhenInSeason`. When `NotifyWhenInSeason` calls it to modify the vacation, `Vacations`, at the top of the tree, would recognize that things had changed, which would cause it to re-render, which in turns causes all of its descendants to re-render.

It sounds exhausting and complicated, and sometimes it can be, but there are techniques that can help. They are so varied and sometimes complex that we can't completely cover them here (nor is this a book about React), but I can point you to some further reading:

Redux
: Redux is usually the first thing that comes to people's minds when they think about comprehensive state management for React applications. It was one of the first formalized state management architectures, and it is still incredibly popular. In concept, it is extremely simple, and it is still the state management framework that I prefer. Even if you don't end up choosing Redux, I recommend you watch the free tutorial videos by its creator, Dan Abramov.

MobX
: MobX came along after Redux. It has gained an impressive following in a short amount of time and is probably the second most popular state container, behind Redux. MobX can certainly result in code that seems easier to write, but I still feel that Redux has an edge in providing a good framework as your application scales, even with its increased boilerplate.

Apollo
: Apollo isn't a state management library *per se*, but the way its used often takes the place of one. It's essentially a frontend interface for GraphQL--an alternative to REST APIs—that offers a lot of integration with React. If you're using GraphQL (or interested in it), it's definitely worth looking into.

React Context
: React itself has gotten into the game by providing the Context API, now built into React. It accomplishes some of the same things that Redux does with less boilerplate. However, I feel that React Context is less robust and that Redux is a better choice for applications as they grow.

When you start out with React, you can essentially ignore the complexities of state management across your application, but pretty quickly you'll realize the need for a more organized way to manage state. When you reach that point, you'll want to look into some of these options and pick one that resonates with you.

Deployment Options

So far, we've been using CRA's built-in development server—which really is the best choice for development, and I recommend sticking with it. However, when it comes time for deployment, it's not a suitable choice. Fortunately, CRA comes loaded with a build script that creates a bundle optimized for production, and then you have many options. When you're ready to create a deployment bundle, simply run `yarn build`, and a *build* directory will be created. All of the assets in the *build* directory are static and can be deployed anywhere.

My current deployment of choice is to put the CRA build in an AWS S3 bucket with Static Website Hosting turned on. This is far from the only option: every major cloud provider and CDN offers something similar.

In this configuration, we have to create routing so that the API calls are routed to your Express server and your static bundle is served from a CDN. For my AWS deployments, I use AWS CloudFront to perform this routing; the static assets are served from the aforementioned S3 bucket, and the API requests are routed to either an Express server on an EC2 instance, or on a Lambda.

Another option is to let Express do the whole thing. This has the advantage of being able to consolidate your entire application onto a single server, which makes for a pretty simple deployment, and makes management easy. It may not be ideal for scalability or performance, but it's a valid choice for small applications.

To serve your application entirely from Express, simply take contents of the *build* directory that was created when you ran `yarn build`, and copy it into the *public* directory in your Express application. As long as you have your static middleware linked in, it will automatically serve the *index.html* file, which is all you need.

Go ahead and try it: if your Express server is still running on port 3033, you should be able to visit *http://localhost:3033* and see the same application that your CRA dev server is providing!

> In case you're wondering how CRA's dev server works, it uses a package called `webpack-dev-server`, which uses Express under the hood! So it all comes back to Express in the end!

Conclusion

This chapter has only scratched the surface of React, and the technologies that swirl around it. If you want to take a deeper dive into React, *Learning React* by Alex Banks and Eve Porcello (O'Reilly) is a great place to start. This book also covers state management with Redux (however, it does not currently cover hooks). The official React documentation is also comprehensive and well-written.

SPAs have certainly changed the way we think about and deliver web applications, and have enabled significant performance improvements, especially on mobile. Even though Express was written in an era when most HTML was still substantially rendered on the server, it has certainly not made Express obsolete. Quite the contrary, the need to provide APIs to single-page applications has, if anything, given Express new life!

It should also be clear from reading this chapter that it's really all the same game: data getting sent back and forth between browsers and servers. It's only the nature of that data that's changed, and getting used to changing HTML through dynamic DOM manipulation.

CHAPTER 17
Static Content

Static content refers to the resources your app will be serving that don't change on a per-request basis. Here are the usual suspects:

Multimedia
: Images, videos, and audio files. It's quite possible to generate image files on the fly, of course (and video and audio, though that's far less common), but most multimedia resources are static.

HTML
: If our web application is using views to render dynamic HTML, it wouldn't generally qualify as static HTML (though for performance reasons, you may dynamically generate HTML, cache it, and serve it as a static resource). SPA applications, as we've seen, commonly send a single, static HTML file to the client, which is the most common reason to treat HTML as a static resource. Note that requiring the client to use an *.html* extension is not very modern, so most servers now allow static HTML resources to be served without the extension (so `/foo` and `/foo.html` would return the same content).

CSS
: Even if you use an abstracted CSS language like LESS, Sass, or Stylus, at the end of the day, your browser needs plain CSS, which is a static resource.[1]

JavaScript
: Just because the server is running JavaScript doesn't mean there won't be client-side JavaScript. Client-side JavaScript is considered a static resource. Of course,

[1] It is possible to use uncompiled LESS in a browser, with some JavaScript magic. There are performance consequences to this approach, so I don't recommend it.

now the line is starting to get a bit hazy: what if there was common code that we wanted to use on the backend and client side? There are ways to solve this problem, but at the end of the day, the JavaScript that gets sent to the client is generally static.

Binary downloads

This is the catchall category: any PDFs, ZIP files, Word documents, installers, and the like.

If you are building an API only, there may be no static resources. If that's the case, you may skip this chapter.

Performance Considerations

The way you handle static resources significantly impacts the real-world performance of your website, especially if your site is multimedia-heavy. The two primary performance considerations are *reducing the number of requests* and *reducing content size*.

Of the two, reducing the number of (HTTP) requests is more critical, especially for mobile (the overhead of making an HTTP request is significantly higher over a cellular network). Reducing the number of requests can be accomplished in two ways: combining resources and browser caching.

Combining resources is primarily an architectural and frontend concern: as much as possible, small images should be combined into a single sprite. Then use CSS to set the offset and size to display only the portion of the image you want. For creating sprites, I highly recommend the free service SpritePad. It makes generating sprites incredibly easy, and it generates the CSS for you as well. Nothing could be easier. SpritePad's free functionality is probably all you'll ever need, but if you find yourself creating a lot of sprites, you might find their premium offerings worth it.

Browser caching helps reduce HTTP requests by storing commonly used static resources in the client's browser. Though browsers go to great lengths to make caching as automatic as possible, it's not magic: there's a lot you can and should do to enable browser caching of your static resources.

Lastly, we can increase performance by reducing the size of static resources. Some techniques are *lossless* (size reduction can be achieved without losing any data), and some techniques are *lossy* (size reduction is achieved by reducing the quality of static resources). Lossless techniques include minification of JavaScript and CSS, and optimizing PNG images. Lossy techniques include increasing JPEG and video compression levels. We'll be discussing minification and bundling (which also reduces HTTP requests) in this chapter.

The importance of reducing HTTP requests will diminish over time as HTTP/2 becomes more commonplace. One of the primary improvements in HTTP/2 is *request and response multiplexing*, which reduces the overhead of fetching multiple resources in parallel. See "Introduction to HTTP/2" by Ilya Grigorikfor more information.

Content Delivery Networks

When you move your website into production, the static resources must be hosted on the internet *somewhere*. You may be used to hosting them on the same server where all your dynamic HTML is generated. Our example so far has also taken this approach: the Node/Express server we spin up when we type `node meadowlark.js` serves all of the HTML as well as static resources. However, if you want to maximize the performance of your site (or allow for doing so in the future), you will want to make it easy to host your static resources on a *content delivery network* (CDN). A CDN is a server that's optimized for delivering static resources. It leverages special headers (that we'll learn about soon) that enable browser caching.

CDNs also can enable *geographic optimization* (often called *edge caching*); that is, they can deliver your static content from a server that is geographically closer to your client. While the internet is very fast indeed (not operating at the speed of light, exactly, but close enough), it is still faster to deliver data over a hundred miles than a thousand. Individual time savings may be small, but if you multiply across all of your users, requests, and resources, it adds up fast.

Most of your static resources will be referenced in HTML views (`<link>` elements to CSS files, `<script>` references to JavaScript files, `<img>` tags referencing images, and multimedia embedding tags). It is also common to have static references in CSS, usually the `background-image` property. Lastly, static resources are sometimes referenced in JavaScript, such as JavaScript code that dynamically changes or inserts `<img>` tags or the `background-image` property.

You generally don't have to worry about cross-domain resource sharing (CORS) when using a CDN. External resources loaded in HTML aren't subject to CORS policy: you have to enable CORS only for resources that are loaded via Ajax (see Chapter 15).

Designing for CDNs

The architecture of your site will influence how you use a CDN. Most CDNs let you configure routing rules to determine where to send incoming requests. While you can get arbitrarily sophisticated with those routing rules, it usually boils down to sending requests for static assets to one location (usually provided by your CDN) and requests for dynamic endpoints (like dynamic pages or API endpoints) to another.

Choosing and configuring a CDN is a big topic, which I won't get into here, but I will arm you with background knowledge that will help you configure your CDN of choice.

The easiest approach to structuring your application is to make it easy to distinguish dynamic from static assets to make the CDN routing rules as simple as possible. While it's possible to do this using subdomains (dynamic assets are served from `meadowlark.com`, and static assets are served from `static.meadowlark.com`, for example), this approach has extra complications and makes local development more difficult. The easier approach is to use the request paths: everything that starts with `/public/` is a static asset, and everything else is dynamic, for example. The approach may be different if you're generating your content with Express or using Express to provide an API for a single-page application.

Server-Rendered Website

If you're using Express to render your dynamic HTML, it's easier to say, "Everything that starts with `/static/` is a static asset, and everything else is dynamic." With this approach, all of your (dynamically generated) URLs would be whatever you want them to be (as long as they don't start with `/static/`, of course!), and all of your static assets will be prefixed with `/static/`:

```
<img src="/static/img/meadowlark-logo-1.png" alt="Meadowlark Logo">
Welcome to <a href="/about">Meadowlark Travel</a>.
```

So far in this book, we've been using the Express `static` middleware as if it were hosting all of the static assets at the root. That is, if we put a static asset *foo.png* in the *public* directory, we reference it with the URL path */foo.png*, not */static/foo.png*. We could, of course, create a subdirectory *static* inside our existing *public* directory, so */public/static/foo.png* would have the URL */static/foo.png* but that seems a little silly. Fortunately, the `static` middleware saves us from that silliness. All we have to do is specify a different path when we call `app.use`:

```
app.use('/static', express.static('public'))
```

Now we can use the same URL structure in our development environment that we will in production. If we're careful about keeping our *public* directory in sync with what's in our CDN, we can reference the same static assets in both places, and move seamlessly between development and production.

When we configure routing for our CDN (you'll have to consult your CDN's documentation for this), your routing will look like this:

URL path	Routing destination / origin
/static/*	Static CDN file store
/* (everything else)	Your Node/Express server, proxy, or load balancer

Single-Page Applications

Single-page applications will typically be the opposite of a server-rendered website: only the API will be routed to your server (for example, any request prefixed with /*api*), and everything else will be rerouted to your static file store.

As we saw in Chapter 16, you will have some way to create a production bundle for your application, which will include all of the static resources, which you'll upload to your CDN. Then all you have to do is make sure routing to your API is configured correctly. So your routing will look like this:

URL path	Routing destination / origin
/api/*	Your Node/Express server, proxy, or load balancer
/* (everything else)	Static CDN file store

Now that we've seen how we might structure an application so we can seamlessly move from development to production, let's turn our attention to what's actually happening with caching and how it improves performance.

Caching Static Assets

Whether you're using Express to serve static assets or using a CDN, it's helpful to understand the HTTP response headers your browser uses to determine when and how to cache static assets:

`Expires`/`Cache-Control`

These two headers tell your browser the maximum amount of time a resource can be cached. They are taken seriously by the browser: if they inform the browser to cache something for a month, it simply won't re-download it for a month, as long as it stays in the cache. It's important to understand that a browser may remove the image from the cache prematurely, for reasons you have no control over. For example, the user could clear the cache manually, or the browser could clear your resource to make room for other resources the user is visiting more frequently. You need one only of these headers, and `Expires` is more broadly supported, so it's preferable to use that one. If the resource is in the cache, and it has not expired yet, the browser will not issue a `GET` request at all, which improves performance, especially on mobile.

`Last-Modified`/`ETag`

These two tags provide a versioning of sorts: if the browser needs to fetch the resource, it will examine these tags *before* downloading the content. A `GET` request is still issued to the server, but if the values returned by these headers satisfy the browser that the resource hasn't changed, it will not proceed to download the file. As the name indicates, `Last-Modified` allows you to specify the date the resource was last modified. `ETag` allows you to use an arbitrary string, which is usually a version string or a content hash.

When serving static resources, you should use the `Expires` header *and* either `Last-Modified` or `ETag`. The Express built-in `static` middleware sets `Cache-Control`, but doesn't handle either `Last-Modified` or `ETag`. So, while it's suitable for development, it's not a great solution for deployment.

If you choose to host your static resources on a CDN, such as Amazon CloudFront, Microsoft Azure, Fastly, Cloudflare, Akamai, or StackPath, the advantage is that they will handle most of these details for you. You will be able to fine-tune the details, but the defaults provided by any of these services are usually good out of the box.

Changing Your Static Content

Caching significantly improves the performance of your website, but it isn't without its consequences. In particular, if you change any of your static resources, clients may not see them until the cached versions expire in your browser. Google recommends you cache for a month, preferably a year. Imagine a user who uses your website every day on the same browser: that person might not see your updates for a whole year!

Clearly this is an undesirable situation, and you can't just tell your users to clear their cache. The solution is cache busting. *Cache busting* is a technique for giving you control of when your user's browser is forced to re-download an asset. Usually this amounts to versioning the asset (*main.2.css* or *main.css?version=2*) or adding some kind of hash (*main.e16b7e149dccfcc399e025e0c454bf77.css*). Whatever technique you use, when you update the asset, the resource name changes, and the browser knows it needs to download it.

We can do the same thing with our multimedia assets. Let's take our logo, for example (*/static/img/meadowlark_logo.png*). If we host it on a CDN for maximum performance, specifying an expiration of one year, and then change the logo, your users may not see the updated logo for up to a year. However, if you rename your logo /

static/img/meadowlark_logo-1.png (and reflect that name change in your HTML), the browser will be forced to download it, because it looks like a new resource.

If you're using a single-page application framework, such as `create-react-app` or similar, they will provide a build step that will create production-ready resource bundles that have hashes appended.

If you're starting from scratch, you'll probably want to look into a *bundler* (which is what the SPA frameworks use under the hood). Bundlers combine your JavaScript, CSS, and some other types of static assets into as few as possible, and minify the result (making it as small as possible). Bundler configuration is a big topic, but fortunately there is a lot of good documentation out there. The most popular bundlers available right now are as follows:

Webpack
: Webpack was one of the first bundlers to really take off, and it still maintains a huge following. It's very sophisticated, but that sophistication comes at a cost: the learning curve is steep. However, it's good to at least know the basics.

Parcel
: Parcel is the newcomer, and it has made a big splash. It's extremely well-documented, extremely fast, and, best of all, has the shortest learning curve. If you're looking to get the job done quickly, without a lot of fuss, start here.

Rollup
: Rollup sits somewhere between Webpack and Parcel. Like Webpack, it's very robust and has a lot of features. However, it is easier to get started with than Webpack, and not as simple as Parcel.

Conclusion

For what seems like such a simple thing, static resources can be a lot of trouble. However, they probably represent the bulk of the data actually being transferred to your visitors, so spending some time optimizing them will yield substantial payoff.

A viable solution to static assets not previously mentioned is to simply host your static resources on a CDN from the start, and always use the full URL to the resource in your views and CSS. This has the advantage of simplicity, but if you ever want to spend a weekend hackathon at that cabin in the woods without internet access, you'd be in trouble!

Elaborate bundling and minification is another area in which you can save time if the payoff isn't worth it for your application. In particular, if your site includes only one or two JavaScript files, and all of your CSS lives in a single file, you could probably skip bundling altogether, but real-world applications have a tendency to grow over time.

Whatever technique you choose to use to serve your static resources, I highly recommend hosting them separately, preferably on a CDN. If it sounds like a hassle to you, let me assure that it's not nearly as difficult as it sounds, especially if you spend a little time on your deployment system, so deploying static resources to one location and your application to another is automatic.

If you're concerned about the hosting costs of CDNs, I encourage you to take a look at what you're paying now for hosting. Most hosting providers essentially charge for bandwidth, even if you don't know it. However, if all of a sudden your site is mentioned on Slashdot, and you get "Slashdotted," you may find yourself with a hosting bill you didn't expect. CDN hosting is usually set up so that you pay for what you use. To give you an example, a website that I once managed for a medium-sized regional company, which used about 20 GB a month of bandwidth, paid only a few dollars per month to host static resources (and it was a very media-heavy site).

The performance gains you realize by hosting your static resources on a CDN are significant, and the cost and inconvenience of doing so is minimal, so I highly recommend going this route.

CHAPTER 18

Security

Most websites and applications these days have some kind of security requirement. If you are allowing people to log in, or if you're storing personally identifiable information (PII), you'll want to implement security for your site. In this chapter, we'll be discussing *HTTP Secure* (HTTPS), which establishes a foundation on which you can build a secure website, and authentication mechanisms, with a focus on third-party authentication.

Security is a big topic that could fill up an entire book. For that reason, our focus is going to be on leveraging existing authentication modules. Writing your own authentication system is certainly possible, but is a large and complicated undertaking. Furthermore, there are good reasons to prefer a third-party login approach, which we will discuss later in this chapter.

HTTPS

The first step in providing secure services is using HTTPS. The nature of the internet makes it possible for a third party to intercept packets being transmitted between clients and servers. HTTPS encrypts those packets, making it extremely difficult for an attacker to get access to the information being transmitted. (I say "very difficult," not "impossible," because there's no such thing as perfect security. However, HTTPS is considered sufficiently secure for banking, corporate security, and healthcare.)

You can think of HTTPS as sort of a foundation for securing your website. It does not provide authentication, but it lays the groundwork for authentication. For example, your authentication system probably involves transmitting a password; if that password is transmitted unencrypted, no amount of authentication sophistication will secure your system. Security is as strong as the weakest link, and the first link in that chain is the network protocol.

The HTTPS protocol is based on the server having a *public-key certificate*, sometimes called an SSL certificate. The current standard format for SSL certificates is called *X.509*. The idea behind certificates is that there are *certificate authorities* (CAs) that issue certificates. A certificate authority makes *trusted root certificates* available to browser vendors. Browsers include these trusted root certificates when you install a browser, and that's what establishes the chain of trust between the CA and the browser. For this chain to work, your server must use a certificate issued by a CA.

The upshot of this is that to provide HTTPS, you need a certificate from a CA, so how does one go about acquiring such a thing? Broadly speaking, you can generate your own, get one from a free CA, or purchase one from a commercial CA.

Generating Your Own Certificate

Generating your own certificate is easy, but generally suitable only for development and testing purposes (and possibly for intranet deployment). Because of the hierarchical nature established by certificate authorities, browsers will trust only certificates generated by a known CA (and that's probably not you). If your website uses a certificate from a CA that's not known to the browser, the browser will warn you in very alarming language that you're establishing a secure connection with an unknown (and therefore untrusted) entity. In development and testing, this is fine: you and your team know that you generated your own certificate, and you expect this behavior from browsers. If you were to deploy such a website to production for consumption by the public, they would turn away in droves.

> If you control the distribution and installation of browsers, you can automatically install your own root certificate when you install the browser. This will prevent people using that browser from being warned when they connect to your website. This is not trivial to set up, however, and applies only to environments in which you control the browser(s) being used. Unless you have a very solid reason to take this approach, it's generally more trouble than it's worth.

To generate your own certificate, you'll need an OpenSSL implementation. Table 18-1 shows how to acquire an implementation.

Table 18-1. Acquiring an implementation for different platforms

Platform	Instructions
macOS	`brew install openssl`
Ubuntu, Debian	`sudo apt-get install openssl`
Other Linux	Download from *http //www.openssl.org/source/;* extract tarball and follow instructions
Windows	Download from *http://gnuwin32.sourceforge.net/packages/openssl htm*

If you are a Windows user, you may need to specify the location of the OpenSSL configuration file, which can be tricky due to Windows pathnames. The surefire way is to locate the *openssl.cnf* file (usually in the *share* directory of the installation), and before you run the `openssl` command, set the `OPENSSL_CNF` environment variable: `SET OPENSSL_CONF=openssl.cnf`.

Once you've installed OpenSSL, you can generate a private key and a public certificate:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout meadowlark.pem
        -out meadowlark.crt
```

You will be asked for some details, such as your country code, city, and state, fully qualified domain name (*FQDN*, also called *common name* or *fully qualified hostname*), and email address. Since this certificate is for development/testing purposes, the values you provide are not particularly important (in fact, they're all optional, but leaving them out will result in a certificate that will be regarded with even more suspicion by a browser). The common name (FQDN) is what the browser uses to identify the domain. So if you're using *localhost*, you can use that for your FQDN, or you can use the IP address of the server, or the server name, if available. The encryption will still work if the common name and domain you use in the URL don't match, but your browser will give you an additional warning about the discrepancy.

If you're curious about the details of this command, you can read about them on the OpenSSL documentation page. It is worth pointing out that the `-nodes` option doesn't have anything to do with Node, or even the plural word "nodes": it actually means "no DES," meaning the private key is not DES-encrypted.

The result of this command is two files, *meadowlark.pem* and *meadowlark.crt*. The Privacy-Enhanced Electronic Mail (PEM) file is your private key, and should not be made available to the client. The CRT file is the self-signed certificate that will be sent to the browser to establish a secure connection.

Alternatively, there are websites that will provide free self-signed certificates, such as this one.

Using a Free Certificate Authority

HTTPS is based on trust, and it's an unfortunate reality that one of the easiest ways to gain trust on the internet is to buy it. And it's not all snake oil, either: establishing the security infrastructure, insuring certificates, and maintaining relationships with browser vendors is expensive.

Buying a certificate is not your only legitimate option for production-ready certificates: Let's Encrypt, a free, automated CA based on open source, has become a great option. As a matter of fact, unless you're already invested in an infrastructure that offers free or inexpensive certificates as its part of your hosting (AWS, for example), Let's Encrypt is a great option. The only downside to Let's Encrypt is that the maximum lifetime for their certificates is 90 days. This downside is offset by the fact that Let's Encrypt makes it very easy to automatically renew the certificates, and recommends setting up an automated process to do so every 60 days to ensure the certificates don't expire.

All of the major certificate vendors (such as Comodo and Symantec) offer free trial certificates that last anywhere from 30 to 90 days. This is a valid option if you want to test a commercial certificate, but you will need to purchase a certificate before the trial period is up if you want to ensure continuity of service.

Purchasing a Certificate

Currently, 90% of the approximately 50 root certificates distributed with every major browser are owned by four companies: Symantec (which purchased VeriSign), Comodo Group, Go Daddy, and GlobalSign. Purchasing directly from a CA can be quite expensive: it usually starts around $300 per year (though some offer certificates for less than $100 per year). A less expensive option is going through a reseller, from whom you can get an SSL certificate for as little as $10 per year or less.

It's important to understand exactly what it is you're paying for, and why you would pay $10, $150, or $300 (or more) for a certificate. The first important point to understand is that there is no difference whatsoever in the level of encryption offered between a $10 certificate and a $1,500 certificate. This is something that expensive certificate authorities would rather you not know: their marketing tries hard to obscure this fact.

If you choose to go with a commercial certificate vendor, I recommend the following three considerations in making your choice:

Customer support

If you ever have problems with your certificate, whether it be browser support (customers will let you know if your certificate is flagged by their browser as not trustworthy), installation issues, or renewal hassles, you will appreciate good customer support. This is one reason you might purchase a more expensive certificate. Often, your hosting provider will resell certificates, and in my experience, they provide a higher level of customer support, because they want to keep you as a hosting client as well.

Single-domain, multisubdomain, wildcard, and multidomain certificates

The most inexpensive certificates are usually *single domain*. That may not sound so bad, but remember that it means that if you purchase a certificate for *meadowlarktravel.com*, then the certificate will not work for *www.meadowlarktravel.com*, or vice versa. For this reason, I tend to avoid single-domain certificates, though it can be a good option for the extremely budget conscious (you can always set up redirects to funnel requests to the proper domain). *Multisubdomain certificates* are good in that you can purchase a single certificate that covers *meadowlarktravel.com*, *www.meadowlark.com*, *blog.meadowlarktravel.com*, *shop.meadowlarktravel.com*, etc. The downside is that you have to know in advance what subdomains you want to use.

If you see yourself adding or using different subdomains over the course of a year (that need to support HTTPS), you might be better off going with a *wildcard* certificate, which are generally more expensive. But they will work for *any* subdomain, and you never have to specify what the subdomains are.

Lastly, there are *multidomain certificates*, which, like wildcard certificates, tend to be more expensive. These certificates support whole multiple domains so, for example, you could have *meadowlarktravel.com*, *meadowlarktravel.us*, *meadowlarktravel.com*, and the *www* variants.

Domain, organization, and extended validation certificates

There are three kinds of certificates: domain, organization, and extended validation. *Domain certificates*, as the name indicates, simply provide confidence that you're doing business with the *domain* that you think you are. *Organization certificates*, on the other hand, provide some assurance about the actual organization you're dealing with. They're more difficult to get: there's usually paperwork involved, and you must provide things like state and/or federal business name records, physical addresses, etc. Different certificate vendors will require different documentation, so make sure to ask your certificate vendor what's required to get one of these certificates. Lastly are *extended validation certificates*, which are the Rolls Royce of SLL certificates. They are like organization certificates in that they verify the existence of the organization, but they require a higher standard of proof, and can even require expensive audits to establish your data security practices (though this seems to be increasingly rare). They can be had for as little as $150 for a single domain.

I recommend either the less expensive domain certificates or the extended validation certificates. Organization certificates, while they verify the existence of your organization, are not displayed any differently than browsers, so in my experience, unless the user actually examines the certificate (which is rare), there will be no apparent difference between this and a domain certificate. Extended validation certificates, on the other hand, usually display some clues to users that they are dealing with a legitimate business (such as the URL bar being displayed in green, and the organization name being displayed next to the SSL icon).

If you've dealt with SSL certificates before, you might be wondering why I didn't mention certificate insurance. I've omitted that price differentiator because essentially it's insurance against something that's almost impossible. The idea is that if someone suffers financial loss due to a transaction on your website, and they can *prove it was due to inadequate encryption*, the insurance is there to cover your damages. While it is certainly possible that, if your application involves financial transactions, someone may attempt to take legal action against you for financial loss, the likelihood of it being due to inadequate encryption is essentially zero. If I were to attempt to seek damages from a company due to financial loss linked to their online services, the absolute last approach I would take is to attempt to prove that the SSL encryption was broken. If you're faced with two certificates that differ only in price and insurance coverage, buy the cheaper certificate.

The process of purchasing a certificate starts with the creation of a private key (as we did previously for the self-signed certificate). You will then generate a *certificate signing request* (CSR) that will be uploaded during the certificate purchase process (the certificate issuer will provide instructions for doing this). Note that the certificate issuer never has access to your private key, nor is your private key transmitted over the internet, which protects the security of the private key. The issuer will then send you the certificate, which will have an extension of *.crt*, *.cer*, or *.der* (the certificate will be in a format called Distinguished Encoding Rules or DER, hence the less common *.der* extension). You will also receive any certificates in the certificate chain. It is safe to email this certificate because it won't work without the private key you generated.

Enabling HTTPS for Your Express App

You can modify your Express app to serve your website over HTTPS. In practice and in production, this is extremely uncommon, which we'll learn about in the next section. However, for advanced applications, testing, and your own understanding of HTTPS, it's useful to know how to serve HTTPS.

Once you have your private key and certificate, using them in your app is easy. Let's revisit how we've been creating our server:

```
app.listen(app.get('port'), () => {
  console.log(`Express started in ${app.get('env')} mode ` +
    `on port + ${app.get('port')}.`)
})
```

Switching over to HTTPS is simple. I recommend that you put your private key and SSL cert in a subdirectory called *ssl* (though it's quite common to keep it in your project root). Then you just use the `https` module instead of `http`, and pass an `options` object along to the `createServer` method:

```
const https = require('https')
const fs = require('fs')          // usually at top of file

// ...the rest of your application configuration
```

```
const options = {
  key: fs.readFileSync(__dirname + '/ssl/meadowlark.pem'),
  cert: fs.readFileSync(__dirname + '/ssl/meadowlark.crt'),
}

const port = process.env.PORT || 3000
https.createServer(options, app).listen(port, () => {
  console.log(`Express started in ${app.get('env')} mode ` +
    `on port + ${port}.`)
})
```

That's all there is to it. Assuming you're still running your server on port 3000, you can now connect to *https://localhost:3000*. If you try to connect to *http://localhost:3000*, it will simply time out.

A Note on Ports

Whether you know it or not, when you visit a website, you're *always* connecting to a specific port, even though it's not specified in the URL. If you don't specify a port, port 80 is assumed for HTTP. As a matter of fact, most browsers will simply not display the port number if you explicitly specify port 80. For example, navigate to *http://www.apple.com:80*; chances are, when the page loads, the browser will simply strip off the *:80*. It's still connecting on port 80; it's just implicit.

Similarly, there's a standard port for HTTPS, 443. Browser behavior is similar: if you connect to *https://www.google.com:443*, most browsers will simply not display the *:443*, but that's the port they're connecting over.

If you're not using port 80 for HTTP or port 443 for HTTPS, you'll have to explicitly specify the port *and* the protocol to connect correctly. There's no way to run HTTP and HTTPS on the same port (technically, it's possible, but there's no good reason to do it, and the implementation would be very complicated).

If you want to run your HTTP app on port 80, or your HTTPS app on port 443 so you don't have to specify the port explicitly, you have two things to consider. First is that many systems already have a default web server running on port 80.

The other thing to know is that on most operating systems, ports 1–1023 require elevated privileges to open. For example, on a Linux or macOS machine, if you attempt to start your app on port 80, it will probably fail with an `EACCES` error. To run on port 80 or 443 (or any port under 1024), you'll need to elevate your privileges by using the `sudo` command. If you don't have administrator rights, you will be unable to start the server directly on port 80 or 443.

Unless you're managing your own servers, you probably don't have root access to your hosted account: so what happens when you want to run on port 80 or 443? Generally, hosting providers have some kind of proxy service that runs with elevated privileges that will pass requests through to your app, which is running on a nonprivileged port. We'll learn more about this in the next section.

HTTPS and Proxies

As we've seen, it's very easy to use HTTPS with Express, and for development, it will work fine. However, when you want to scale your site out to handle more traffic, you will want to use a proxy server such as NGINX (see Chapter 12). If your site is running in a shared hosting environment, it is almost certain that there will be a proxy server that will route requests to your application.

If you're using a proxy server, then the client (the user's browser) will communicate with the *proxy server*, not your server. The proxy server, in turn, will most likely communicate with your app over regular HTTP (since your app and the proxy server will be running together on a trusted network). You will often hear people say that the HTTPS *terminates* at the proxy server, or that the proxy is performing "SSL termination."

For the most part, once you or your hosting provider has correctly configured the proxy server to handle HTTPS requests, you won't need to do any additional work. The exception to that rule is if your application needs to handle both secure and insecure requests.

There are three solutions to this problem. The first is simply to configure your proxy to redirect all HTTP traffic to HTTPS, in essence forcing all communication with your application to be over HTTPS. This approach is becoming much more common, and it's certainly an easy solution to the problem.

The second approach is to somehow communicate the protocol used in the client-proxy communication to the server. The usual way to communicate this is through the `X-Forwarded-Proto` header. For example, to set this header in NGINX:

```
proxy_set_header X-Forwarded-Proto $scheme;
```

Then, in your app, you could test to see if the protocol was HTTPS:

```
app.get('/', (req, res) => {
  // the following is essentially
  // equivalent to: if(req.secure)
  if(req.headers['x-forwarded-proto'] === 'https') {
    res.send('line is secure')
  } else {
    res.send('you are insecure!')
  }
})
```

In NGINX, there is a separate `server` configuration block for HTTP and HTTPS. If you fail to set the `X-Forwarded-Protocol` in the configuration block corresponding to HTTP, you open yourself up to the possibility of a client spoofing the header and thereby fooling your application into thinking that the connection is secure even though it isn't. If you take this approach, make sure you *always* set the `X-Forwarded-Protocol` header.

When you're using a proxy, Express provides some convenience properties that make the proxy more "transparent" (as if you weren't using one, without sacrificing the benefits). To take advantage of that, tell Express to trust the proxy by using `app.enable('trust proxy')`. Once you do, `req.protocol`, `req.secure`, and `req.ip` will refer to the client's connection to the proxy, not to your app.

Cross-Site Request Forgery

Cross-site request forgery (CSRF) attacks exploit the fact that users generally trust their browser and visit multiple sites in the same session. In a CSRF attack, script on a malicious site makes requests of another site: if you are logged in on the other site, the malicious site can successfully access secure data from another site.

To prevent CSRF attacks, you must have a way to make sure a request legitimately came from your website. The way we do this is to pass a unique token to the browser. When the browser then submits a form, the server checks to make sure the token matches. The `csurf` middleware will handle the token creation and verification for you; all you'll have to do is make sure the token is included in requests to the server. Install the `csurf` middleware (`npm install csurf`); then link it in and add a token to `res.locals`. Make sure you link in the `csurf` middleware after you link in `body-parser`, `cookie-parser`, and `express-session`:

```
// this must come after we link in body-parser,
// cookie-parser, and express-session
const csrf = require('csurf')

app.use(csrf({ cookie: true }))
app.use((req, res, next) => {
  res.locals._csrfToken = req.csrfToken()
  next()
})
```

The `csurf` middleware adds the `csrfToken` method to the request object. We don't have to assign it to `res.locals`; we could just pass `req.csrfToken()` explicitly to every view that needs it, but this is generally less work.

Note that the package itself is called csurf, but most of the variables and methods are csrf, without the "u." It's easy to get tripped up here, so mind your vowels!

Now on all of your forms (and AJAX calls), you'll have to provide a field called _csrf, which must match the generated token. Let's see how we would add this to one of our forms:

```
<form action="/newsletter" method="POST">
  <input type="hidden" name="_csrf" value="{{_csrfToken}}">
  Name: <input type="text" name="name"><br>
  Email: <input type="email" name="email"><br>
  <button type="submit">Submit</button>
</form>
```

The csurf middleware will handle the rest: if the body contains fields, but no valid _csrf field, it will raise an error (make sure you have an error route in your middleware!). Go ahead and remove the hidden field and see what happens.

If you have an API, you probably don't want the csurf middleware interfering with it. If you want to restrict access to your API from other websites, you should look into the "API key" functionality of an API library like connect-rest. To prevent csurf from interfering with your middleware, link it in before you link in csurf.

Authentication

Authentication is a big, complicated topic. Unfortunately, it's also a vital part of most nontrivial web applications. The most important piece of wisdom I can impart to you is *don't try to do it yourself*. If you look at your business card and it doesn't say "Security Expert," you probably aren't prepared for the complex considerations involved in designing a secure authentication system.

I'm not saying that you shouldn't try to understand the security systems in your application. I'm just recommending that you don't try to build it yourself. Feel free to study the open source code of the authentication techniques I'm going to recommend. It will certainly give you some insight as to why you might not want to take on this task unaided!

Authentication Versus Authorization

While the two terms are often used interchangeably, there is a difference. *Authentication* refers to verifying users' identities. That is, they are who they say they are. *Authorization* refers to determining what a user is authorized to access, modify, or view. For example, customers might be authorized to access their account information, whereas a Meadowlark Travel employee would be authorized to access another person's account information or sales notes.

Authentication is often abbreviated as *authN* and "authorization" as *authZ*.

Usually (but not always), authentication comes first, and then authorization is determined. Authorization can be very simple (authorized/not authorized), broad (user/administrator), or very fine-grained, specifying read, write, delete, and update privileges against different account types. The complexity of your authorization system is dependent on the type of application you're writing.

Because authorization is so dependent on the details of your application, I'll be giving only a rough outline in this book, using a very broad authentication scheme (customer/employee). I will often use the abbreviation "auth," but only when it is clear from the context whether it means "authentication" or "authorization," or when it doesn't matter.

The Problem with Passwords

The problem with passwords is that every security system is only as strong as its weakest link. And passwords require the user to invent a password—and there's your weakest link. Humans are notoriously bad at coming up with secure passwords. In an analysis of security breaches in 2018, the most popular password is "123456." "password" is second. Even in the security conscious year of 2018, people are still choosing abysmally bad passwords. Having password policies requiring, for example, a capital letter, a number, and a punctuation mark is just going to result in a password of "Password1!".

Even analyzing passwords against a list of common passwords doesn't do much to stop the problem. Then people start writing down their higher-quality passwords on notepads, leaving them in unencrypted files on their computers, or emailing them to themselves.

At the end of the day, it's a problem that you, the app designer, cannot do much to fix. However, there are things you can do that promote more secure passwords. One is to pass the buck and rely on a third party for authentication. The other is to make your login system friendly to password management services, like 1Password, Bitwarden, and LastPass.

Third-Party Authentication

Third-party authentication takes advantage of the fact that pretty much everyone on the internet has an account on at least one major service, such as Google, Facebook, Twitter, or LinkedIn. All of these services provide a mechanism to authenticate and identify your users through their service.

> Third-party authentication is often referred to as *federated authentication* or *delegated authentication*. The terms are largely interchangeable, though federated authentication is usually associated with Security Assertion Markup Language (SAML) and OpenID, and delegated authentication is often associated with OAuth.

Third-party authentication has three major advantages. First, your authentication burden is lowered. You do not have to worry about authenticating individual users, only interacting with a trusted third party. The second advantage is that it reduces *password fatigue*: the stress associated with having too many accounts. I use LastPass, and I just checked my password vault: I have almost 400 passwords. As a technology professional, I may have more than your average internet user, but it's not uncommon for even a casual internet user to have dozens or even hundreds of accounts. Lastly, third-party authentication is *frictionless*: it allows your users to start using your site more quickly, with credentials they already have. Often, if users see that they have to create yet *another* username and password, they will simply move on.

If you don't use a password manager, the chances are, you're using the same password for most of those sites (most people have a "secure" password they use for banking and the like, and an "insecure" password they use for everything else). The problem with this approach is that if even *one* of the sites you use that password for is breached, and your password becomes known, then hackers will try using that same password with other services. It's like putting all of your eggs in one basket.

Third-party authentication has its downsides. Hard as it is to believe, there *are* folks out there who don't have an account on Google, Facebook, Twitter, or LinkedIn. Then, among the people who *do* have such accounts, suspicion (or a desire for privacy) may make them unwilling to use those credentials to log onto your website. Many websites solve this particular problem by encouraging users to use an existing account, but those who don't have them (or are unwilling to use them to access your service) can create a new login for your service.

Storing Users in Your Database

Whether or not you rely on a third party to authenticate your users, you will want to store a record of users in your own database. For example, if you're using Facebook for authentication, that only verifies a user's identity. If you need to save settings specific to that user, you can't reasonably use Facebook for that: you have to store information about that user in your own database. Also, you probably want to associate an email address with your users, and they may not wish to use the same email address they use for Facebook (or whatever third-party authentication service you use). Lastly, storing user information in your database allows you to perform authentication yourself, should you wish to provide that option.

So let's create a model for our users, *models/user.js*:

```
const mongoose = require('mongoose')

const userSchema = mongoose.Schema({
  authId: String,
  name: String,
  email: String,
  role: String,
  created: Date,
})

const User = mongoose.model('User', userSchema)
module.exports = User
```

And modify *db.js* with the appropriate abstractions (if you're using PostgreSQL, I'll leave it as an exercise to hook up this abstraction):

```
const User = require('./models/user')

module.exports = {
  //...
  getUserById: async id => User.findById(id),
  getUserByAuthId: async authId => User.findOne({ authId }),
  addUser: async data => new User(data).save(),
}
```

Recall that every object in a MongoDB database has its own unique ID, stored in its `_id` property. However, that ID is controlled by MongoDB, and we need some way to map a user record to a third-party ID, so we have our own ID property, called `authId`. Since we'll be using multiple authentication strategies, that ID will be a combination of a strategy type and a third-party ID, to prevent collisions. For example, a Facebook user might have an `authId` of `facebook:525764102`, whereas a Twitter user would have an `authId` of `twitter:376841763`.

We will be using two roles in our example: "customer" and "employee."

Authentication Versus Registration and the User Experience

Authentication refers to verifying a user's identity, either with a trusted third party, or through credentials you've provided the user (such as a username and password). Registration is the process by which a user gets an account on your site (from our perspective, registration is when we create a user record in the database).

When users join your site for the first time, it should be clear to them that they're registering. Using a third-party authentication system, we could register them without their knowledge if they successfully authenticate through the third party. This is not generally considered a good practice, and it should be clear to users that they're registering for your site (whether they're authenticating through a third party or not), and provide a clear mechanism for canceling their membership.

One user experience situation to consider is "third-party confusion." If a user registers in January for your service using Facebook, then returns in July, and is confronted with a screen offering the choices of logging in with Facebook, Twitter, Google, or LinkedIn, the user may very well have forgotten what registration service was originally used. This is one of the pitfalls of third-party authentication, and there is precious little you can do about it. It's another good reason to ask the user to provide an email address: this way, you can give the user an option to look up their account by email, and send an email to that address specifying what service was used for authentication.

If you feel that you have a firm grasp on the social networks your users use, you can ease this problem by having a primary authentication service. For example, if you feel pretty confident that the majority of your users have a Facebook account, you could have a big button that says, "Log in with Facebook." Then, using smaller buttons or even just text links, say, "or log in with Google, Twitter, or LinkedIn." This approach can cut down on the instance of third-party confusion.

Passport

Passport is a very popular and robust authentication module for Node/Express. It is not tied to any one authentication mechanism; rather, it is based on the idea of pluggable authentication *strategies* (including a local strategy if you don't want to use third-party authentication). Understanding the flow of authentication information can be overwhelming, so we'll start with just one authentication mechanism and add more later.

The detail that's important to understand is that, with third-party authentication, your app *never receives a password*. That is handled entirely by the third party. This is a good thing: it's putting the burden of secure handling and storage of passwords on the third party.[1]

The whole process, then, relies on redirects (it must, if your application is never to receive the user's third-party password). At first, you might be confused about why you can pass *localhost* URLs to the third party and still successfully authenticate (after all, the third-party server handling your request doesn't know about *your localhost*). It works because the third party simply instructs *your browser* to redirect, and your browser is inside your network, and can therefore redirect to local addresses.

The basic flow is shown in Figure 18-1. This diagram shows the important flow of functionality, making it clear that the authentication actually occurs on the third-party website. Enjoy the simplicity of the diagram—things are about to get a lot more complicated.

When you use Passport, there are four steps that your app will be responsible for. Consider a more detailed view of the third-party authentication flow, as shown in Figure 18-2.

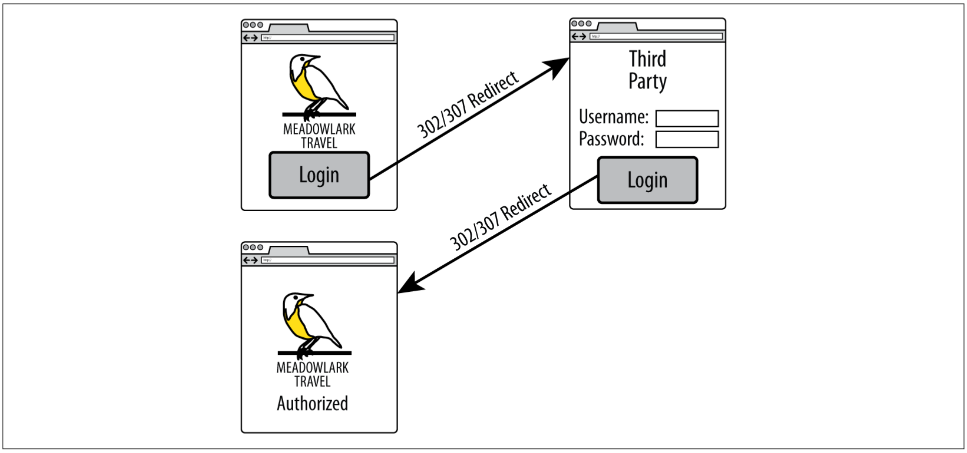


Figure 18-1. Third-party authentication flow

For simplicity, we are using Meadowlark Travel to represent your app, and Facebook for the third-party authentication mechanism. Figure 18-2 illustrates how the user goes from the login page to the secure Account Info page (the Account Info page is just used for illustration purposes: this could be any page on your website that requires authentication).

[1] It is unlikely that the third party is storing passwords either. A password can be verified by storing something called a *salted hash*, which is a one-way transformation of the password. That is, once you generate a hash from a password, you can't recover the password. *Salting* the hash provides additional protection against certain kinds of attacks.

This diagram shows detail you don't normally think about, but is important to understand in this context. In particular, when you visit a URL, *you* aren't making the request of the server: the browser is actually doing that. That said, the browser can do three things: make an HTTP request, display the response, and perform a redirect (which is essentially making another request and displaying another response… which in turn could be another redirect).

In the Meadowlark column, you can see the four steps your application is actually responsible for. Fortunately, we'll be leveraging Passport (and pluggable strategies) to perform the details of those steps; otherwise, this book would be much, much longer.

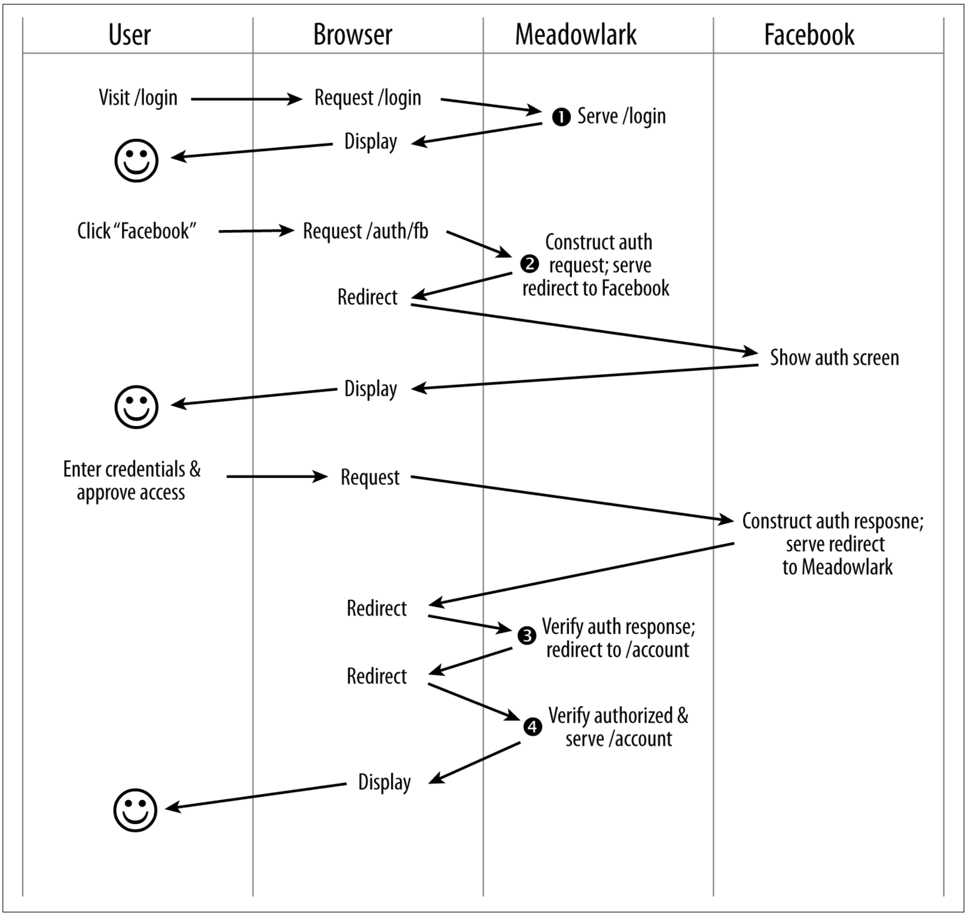

User
Browser
Meadowlark
Facebook
Visit /login
Request /login
❶ Serve /login
Display
Click "Facebook"
Request /auth/fb
❷ Construct auth request; serve redirect to Facebook
Redirect
Show auth screen
Display
Enter credentials & approve access
Request
Construct auth resposne; serve redirect to Meadowlark
Redirect
❸ Verify auth response; redirect to /account
Redirect
❹ Verify authorized & serve /account
Display

Figure 18-2. Detailed view of third-party authentication flow

Before we get into implementation details, let's consider each of the steps in a little more detail:

Login page

The login page is where the user can choose the login method. If you're using a third-party authentication, it's usually just a button or a link. If you're using local authentication, it will include username and password fields. If the user attempts to access a URL requiring authentication (such as `/account` in our example) without being logged in, this is probably the page you will want to redirect to (alternatively, you could redirect to a Not Authorized page with a link to the login page).

Construct authentication request

In this step, you'll be constructing a request to be sent to a third party (via a redirect). The details of this request are complicated and specific to the authentication strategy. Passport (and the strategy plugin) will be doing all the heavy lifting here. The auth request includes protection against man-in-the-middle attacks, as well as other vectors an attacker might exploit. Usually the auth request is short-lived, so you can't store it and expect to use it later: this helps prevent attacks by limiting the window in which an attacker has time to act. This is where you can request additional information from the third-party authorization mechanism. For example, it's common to request the user's name, and possibly email address. Keep in mind that the more information you request from users, the less likely they are to authorize your application.

Verify authentication response

Assuming the user authorized your application, you'll get back a valid auth response from the third party, which is proof of the user's identity. Once again, the details of this validation are complicated and will be handled by Passport (and the strategy plugin). If the auth response indicates that the user is not authorized (if invalid credentials were entered, or your application wasn't authorized by the user), you would then redirect to an appropriate page (either back to the login page, or to a Not Authorized or Unable to Authorize page). Included in the auth response will be an ID for the user that is unique *to that specific third party*, as well as any additional details you requested in step 2. To enable step 4, we must "remember" that the user is authorized. The usual way to do this is to set a session variable containing the user's ID, indicating that this session has been authorized (cookies can also be used, though I recommend using sessions).

Verify authorization

In step 3, we stored a user ID in the session. The presence of that user ID allows us to retrieve a user object from the database that contains information about what the user is authorized to do. In this manner, we don't have to authenticate with the third party for every request (which would result in a slow and painful user experience). This task is simple, and we no longer need Passport for this: we have our own user object that contains our own authentication rules. (If that object isn't available, it indicates the request isn't authorized, and we can redirect to the login or Not Authorized page.)

> Using Passport for authentication is a fair amount of work, as you'll see in this chapter. However, authentication is an important part of your application, and I feel that it is wise to invest some time in getting it right. There are projects such as LockIt that try to provide a more "off the shelf" solution. Another increasingly popular option is Auth0, which is very robust but isn't as easy to set up as LockIt. To make the most effective use of LockIt or Auth0 (or similar solutions), however, it behooves you to understand the details of authentication and authorization, which is what this chapter is designed to do. Also, if you ever need to customize an authentication solution, Passport is a great place to start.

Setting up Passport

To keep things simple, we'll start with a single authentication provider. Arbitrarily, we'll choose Facebook. Before we can set up Passport and the Facebook strategy, we'll need to do a little configuration in Facebook. For Facebook authentication, you'll need a *Facebook app*. If you already have a suitable Facebook app, you can use that, or you can create a new one specifically for authentication. If possible, you should use your organization's official Facebook account to create the app. That is, if you worked for Meadowlark Travel, you would use the Meadowlark Travel Facebook account to create the app (you can always add your personal Facebook account as an administrator of the app for ease of administration). For testing purposes, it's fine to use your own Facebook account, but using a personal account for production will appear unprofessional and suspicious to your users.

The details of Facebook app administration seem to change fairly frequently, so I am not going to explain the details here. Consult the Facebook developer documentation if you need details on creating and administering your app.

For development and testing purposes, you will need to associate the development/testing domain name with the app. Facebook allows you to use *localhost* (and port numbers), which is great for testing purposes. Alternatively, you can specify a local IP address, which can be helpful if you're using a virtualized server, or another server on your network for testing. The important thing is that the URL you enter into your browser to test the app (for example, *http://localhost:3000*) is associated with the Facebook app. Currently, you can associate only one domain with your app: if you need to be able to use multiple domains, you will have to create multiple apps (for example, you could have Meadowlark Dev, Meadowlark Test, and Meadowlark Staging; your production app can simply be called Meadowlark Travel).

Once you've configured your app, you will need its unique app ID, and its app secret, both of which can be found on the Facebook app management page for that app.

> One of the biggest frustrations you'll probably face is receiving a message from Facebook such as "Given URL is not allowed by the Application configuration." This indicates that the hostname and port in the callback URL do not match what you've configured in your app. If you look at the URL in your browser, you will see the encoded URL, which should give you a clue. For example, if I'm using 192.168.0.103:3443, and I get that message, I look at the URL. If I see *redirect_uri=https%3A%2F%2F192.68.0.103%3A3443%2Fauth%2Ffacebook%2Fcallback* in the querystring, I can quickly spot the mistake: I used 68 instead of 168 in my hostname.

Now let's install Passport and the Facebook authentication strategy:

```
npm install passport passport-facebook
```

Before we're done, there's going to be a lot of authentication code (especially if we're supporting multiple strategies), and we don't want to clutter up *meadowlark.js* with all that code. Instead, we'll create a module called *lib/auth.js*. This is going to be a large file, so we're going to take it piece by piece (see *ch18* in the companion repo for the finished example). We'll start with the imports and two methods that Passport requires, `serializeUser` and `deserializeUser`:

```
const passport = require('passport')
const FacebookStrategy = require('passport-facebook').Strategy

const db = require('../db')

passport.serializeUser((user, done) => done(null, user._id))

passport.deserializeUser((id, done) => {
  db.getUserById(id)
    .then(user => done(null, user))
    .catch(err => done(err, null))
})
```

Passport uses `serializeUser` and `deserializeUser` to map requests to the authenticated user, allowing you to use whatever storage method you want. In our case, we are only going to store our database ID (the `_id` property) in the session. The way we're using the ID here makes "serialize" and "deserialize" soft of into misnomers: we're actually just storing a user ID in the session. Then, when needed, we can get a user object by finding that ID in the database.

Once these two methods are implemented, as long as there is an active session, and the user has successfully authenticated, `req.session.passport.user` will be the corresponding user object as retrieved from the database.

Next, we're going to choose what to export. To enable Passport's functionality, we'll need to do two distinct activities: initialize Passport, and register routes that will handle authentication and the redirected callbacks from our third-party authentication services. We don't want to combine these two in one function because in our main application file, we may want to choose when Passport is linked into the middleware chain (remember that order is significant when adding middleware). So, instead of having our module export function that does either of these things, we're going to have it return a function that returns an object that has the methods we need. Why not just return an object to start with? Because we need to bake in some configuration values. Also, since we need to link the Passport middleware into our application, a function is an easy way to pass in the Express application object:

```
module.exports = (app, options) => {
  // if success and failure redirects aren't specified,
  // set some reasonable defaults
  if(!options.successRedirect) options.successRedirect = '/account'
  if(!options.failureRedirect) options.failureRedirect = '/login'
  return {
    init: function() { /* TODO */ },
    registerRoutes: function() { /* TODO */ },
  }
}
```

Before we get into the details of the `init` and `registerRoutes` methods, let's look at how we'll use this module (hopefully that will make this business of returning a function that returns an object a little more clear):

```
const createAuth = require('./lib/auth')

// ...other app configuration

const auth = createAuth(app, {
  // baseUrl is optional; it will default to localhost if you omit it;
  // it can be helpful to set this if you're not working on
  // your local machine.  For example, if you were using a staging server,
  // you might set the BASE_URL environment variable to
  // https://staging.meadowlark.com
  baseUrl: process.env.BASE_URL,
  providers: credentials.authProviders,
  successRedirect: '/account',
  failureRedirect: '/unauthorized',
})

// auth.init() links in Passport middleware:
auth.init()

// now we can specify our auth routes:
auth.registerRoutes()
```

Notice that, in addition to specifying the success and failure redirect paths, we also specify a property called `providers`, which we've externalized in the credentials file (see Chapter 13). We'll need to add the `authProviders` property to *.credentials.development.json*:

```
"authProviders": {
  "facebook": {
    "appId": "your_app_id",
    "appSecret": "your_app_secret"
  }
}
```

> Another reason to bundle our authentication code in a module like this is that we can reuse it for other projects; as a matter of fact, there are already some authentication packages that do essentially what we're doing here. However, it's important to understand the details of what's going on, so even if you end up using a module someone else wrote, this will help you understand everything that's going on in your authentication flow.

Now let's take care of our `init` method (previously a "TODO" in *auth.js*):

```
init: function() {
  var config = options.providers

  // configure Facebook strategy
  passport.use(new FacebookStrategy({
    clientID: config.facebook.appId,
    clientSecret: config.facebook.appSecret,
    callbackURL: (options.baseUrl || '') + '/auth/facebook/callback',
  }, (accessToken, refreshToken, profile, done) => {
    const authId = 'facebook:' + profile.id
    db.getUserByAuthId(authId)
      .then(user => {
        if(user) return done(null, user)
        db.addUser({
          authId: authId,
          name: profile.displayName,
          created: new Date(),
          role: 'customer',
        })
          .then(user => done(null, user))
          .catch(err => done(err, null))
      })
      .catch(err => {
        if(err) return done(err, null);
      })
  }))

  app.use(passport.initialize())
  app.use(passport.sessionp))
},
```

This is a pretty dense bit of code, but most of it is actually just Passport boilerplate. The important bit is inside the function that gets passed to the `FacebookStrategy` instance. When this function gets called (after the user has successfully authenticated), the `profile` parameter contains information about the Facebook user. Most important, it includes a Facebook ID: that's what we'll use to associate a Facebook account to our own user object. Note that we namespace our `authId` property by prefixing *`facebook:`*. Slight as the chance may be, this prevents the possibility of a Facebook ID colliding with a Twitter or Google ID (it also allows us to examine user models to see what authentication method a user is using, which could be useful). If the database already contains an entry for this namespaced ID, we simply return it (this is when `serializeUser` gets called, which will put our own user ID into the session). If no user record is returned, we create a new user object and save it to the database.

The last thing we have to do is create our `registerRoutes` method (don't worry, this one is much shorter):

```
registerRoutes: () => {
  app.get('/auth/facebook', (req, res, next) => {
    if(req.query.redirect) req.session.authRedirect = req.query.redirect
    passport.authenticate('facebook')(req, res, next)
  })
  app.get('/auth/facebook/callback', passport.authenticate('facebook',
    { failureRedirect: options.failureRedirect }),
    (req, res) => {
      // we only get here on successful authentication
      const redirect = req.session.authRedirect
      if(redirect) delete req.session.authRedirect
      res.redirect(303, redirect || options.successRedirect)
    }
  )
},
```

Now we have the path */auth/facebook*; visiting this path will automatically redirect the visitor to Facebook's authentication screen (this is done by `passport.authenticate('facebook’))`, step 2 in Figure 18-1. Note that we check to see if there's a querystring parameter `redirect`; if there is, we save it in the session. This is so we can automatically redirect to the intended destination after completing authentication. Once the user authorizes with Twitter, the browser will be redirected back to your site —specifically, to the */auth/facebook/callback* path (with the optional `redirect` querystring indicating where the user was originally).

Also on the querystring are authentication tokens that Passport will verify. If the verification fails, Passport will redirect the browser to `options.failureRedirect`. If the verification is successful, Passport will call `next`, which is where your application comes back in. Note how the middleware is chained in the handler for */auth/facebook/callback*: `passport.authenticate` is called first. If it calls `next`, control passes over to your function, which then redirects to either the original location or `options.successRedirect`, if the `redirect` querystring parameter wasn't specified.

> Omitting the `redirect` querystring parameter can simplify your authentication routes, which may be tempting if you have only one URL that requires authentication. However, having this functionality available will eventually come in handy and provide a better user experience. No doubt you've experienced this yourself before: you've found the page you want, and you're instructed to log in. You do, and you're redirected to a default page, and you have to navigate back to the original page. It's not a very satisfying user experience.

The "magic" that Passport is doing during this process is saving the user (in our case, just a database user ID) to the session. This is a good thing, because the browser is *redirecting*, which is a different HTTP request: without having that information in the session, we wouldn't have any way to know that the user had been authenticated! Once a user has been successfully authenticated, `req.session.passport.user` will be set, and that's how future requests will know that the user has been authenticated.

Let's look at our `/account` handler to see how it checks to make sure the user is authenticated (this route handler will be in our main application file, or in a separate routing module, not in */lib/auth.js*):

```
app.get('/account', (req, res) => {
  if(!req.user)
    return res.redirect(303, '/unauthorized')
  res.render('account', { username: req.user.name })
})
// we also need an 'unauthorized' page
app.get('/unauthorized', (req, res) => {
  res.status(403).render('unauthorized')
})
// and a way to logout
app.get('/logout', (req, res) => {
  req.logout()
  res.redirect('/')
})
```

Now only authenticated users will see the account page; everyone else will be redirected to a Not Authorized page.

Role-Based Authorization

So far, we're not technically doing any authorization (we're only differentiating between authorized and unauthorized users). However, let's say we want only customers to see their account views (employees might have an entirely different view where they can see user account information).

Remember that in a single route, you can have multiple functions, which get called in order. Let's create a function called `customerOnly` that will allow only customers:

```
const customerOnly = (req, res, next) => {
  if(req.user && req.user.role === 'customer') return next()
  // we want customer-only pages to know they need to logon
  res.redirect(303, '/unauthorized')
}
```

Let's also create am `employeeOnly` function that will operate a little differently. Let's say we have a path */sales* that we want to be available only to employees. Furthermore, we don't want nonemployees to even be aware of its existence, even if they stumble on it by accident. If a potential attacker went to the */sales* path, and saw a Not Authorized page, that is a little information that might make an attack easier (simply by knowing that the page is there). So, for a little added security, we want nonemployees to see a regular 404 page when they visit the */sales* page, giving potential attackers nothing to work with:

```
const employeeOnly = (req, res, next) => {
  if(req.user && req.user.role === 'employee') return next()
  // we want employee-only authorization failures to be "hidden", to
  // prevent potential hackers from even knowing that such a page exists
  next('route')
}
```

Calling `next('route’)` will not simply execute the next handler in the route: it will skip this route altogether. Assuming there's not a route further on down the line that will handle `/account`, this will eventually pass to the 404 handler, giving us the desired result.

Here's how easy it is to put these functions to use:

```
// customer routes

app.get('/account', customerOnly, (req, res) => {
  res.render('account', { username: req.user.name })
})
app.get('/account/order-history', customerOnly, (req, res) => {
  res.render('account/order-history')
})
app.get('/account/email-prefs', customerOnly, (req, res) => {
  res.render('account/email-prefs')
})
```

```
// employer routes

app.get('/sales', employeeOnly, (req, res) => {
        res.render('sales')
})
```

It should be clear that role-based authorization can be as simple or as complicated as you wish. For example, what if you want to allow multiple roles? You could use the following function and route:

```
const allow = roles => (req, res, next) => {
  if(req.user && roles.split(',').includes(req.user.role)) return next()
  res.redirect(303, '/unauthorized')
}
```

Hopefully that example gives you an idea of how creative you can be with role-based authorization. You could even authorize on other properties, such as the length of time a user has been a member or how many vacations that user has booked with you.

Adding Authentication Providers

Now that our framework is in place, adding more authentication providers is easy. Let's say we want to authenticate with Google. Before we start adding code, you'll have to set up a project on your Google account.

Go to your Google Developers Console and choose a project from the navigation bar (if you don't already have a project, click New Project and follow the instructions. Once you've selected a project, click "Enable APIs and Services" and enable Cloud Identity API. Click Credentials, and then Create Credentials, and choose "OAuth client ID," and then "Web application." Enter the appropriate URLs for your app: for testing you can use *http://localhost:3000* for the authorized origins, and *http://localhost:3000/auth/google/callback* for authorized redirect URIs.

Once you have got everything set up on the Google side, run `npm install passport-google-oauth20`, and add the following code to *lib/auth.js*:

```
// configure Google strategy
passport.use(new GoogleStrategy({
  clientID: config.google.clientID,
  clientSecret: config.google.clientSecret,
  callbackURL: (options.baseUrl || '') + '/auth/google/callback',
}, (token, tokenSecret, profile, done) => {
  const authId = 'google:' + profile.id
  db.getUserByAuthId(authId)
    .then(user => {
      if(user) return done(null, user)
      db.addUser({
        authId: authId,
    name: profile.displayName,
    created: new Date(),
    role: 'customer',
   })
    .then(user => done(null, user))
    .catch(err => done(err, null))
  })
  .catch(err => {
   console.log('whoops, there was an error: ', err.message)
   if(err) return done(err, null);
  })
}))
```

And the following to the `registerRoutes` method:

```
app.get('/auth/google', (req, res, next) => {
 if(req.query.redirect) req.session.authRedirect = req.query.redirect
 passport.authenticate('google', { scope: ['profile'] })(req, res, next)
})
app.get('/auth/google/callback', passport.authenticate('google',
 { failureRedirect: options.failureRedirect }),
 (req, res) => {
  // we only get here on successful authentication
  const redirect = req.session.authRedirect
  if(redirect) delete req.session.authRedirect
  res.redirect(303, req.query.redirect || options.successRedirect)
 }
)
```

Conclusion

Congratulations on making it through the most intricate chapter! It's unfortunate that such an important feature (authentication and authorization) is so complicated, but in a world rife with security threats, it's an unavoidable complexity. Fortunately, projects like Passport (and the excellent authentication schemes based on it) lessen our burden somewhat. Still, I encourage you not to give short shrift to this area of your application: exercising diligence in the area of security will make you a good internet citizen. Your users may never thank you for it, but woe be to the owners of an application who allow user data to be compromised because of poor security.

CHAPTER 19

Integrating with Third-Party APIs

Increasingly, successful websites are not completely standalone. To engage existing users and find new users, integration with social networking is a must. To provide store locators or other location-aware services, using geolocation and mapping services is essential. It doesn't stop there: more and more organizations are realizing that providing an API helps expand their service and makes it more useful.

In this chapter, we'll be discussing the two most common integration needs: social media and geolocation.

Social Media

Social media is a great way to promote your product or service: if that's your goal, the ability for your users to easily share your content on social media sites is essential. As I write this, the dominant social networking services are Facebook, Twitter, Instagram, and YouTube. Sites like Pinterest and Flickr have their place, but they are usually a little more audience specific (for example, if your website is about DIY crafting, you would absolutely want to support Pinterest). Laugh if you will, but I predict that MySpace will make a comeback. Its site redesign is inspired, and it's worth noting that MySpace is built on Node.

Social Media Plugins and Site Performance

Most social media integration is a frontend affair. You reference the appropriate JavaScript files in your page, and it enables both incoming content (the top three stories from your Facebook page, for example) and outgoing content (the ability to tweet about the page you're on, for example). While this often represents the easiest path to social media integration, it comes at a cost: I've seen page load times double or even triple thanks to the additional HTTP requests. If page performance is important to you (and it should be, especially for mobile users), you should carefully consider how you integrate social media.

That said, the code that enables a Facebook Like button or a Tweet button leverages in-browser cookies to post on the user's behalf. Moving this functionality to the backend would be difficult (and, in some instances, impossible). So if that is functionality you need, linking in the appropriate third-party library is your best option, even though it can affect your page performance.

Searching for Tweets

Let's say that we want to mention the top 10 most recent tweets that contain the hashtags #Oregon #travel. We could use a frontend component to do this, but it will involve additional HTTP requests. Furthermore, if we do it on the backend, we have the option of caching the tweets for performance. Also, if we do the searching on the backend, we can "blacklist" uncharitable tweets, which would be more difficult on the frontend.

Twitter, like Facebook, allows you to create *apps*. It's something of a misnomer: a Twitter app doesn't *do* anything (in the traditional sense). It's more like a set of credentials that you can use to create the actual app on your site. The easiest and most portable way to access the Twitter API is to create an app and use it to get access tokens.

Create a Twitter app by going to *http://dev.twitter.com*. Make sure you're logged on, and click your username in the navigation bar, and then Apps. Click "Create an app," and follow the instructions. Once you have an application, you'll see that you now have a *consumer API key* and an *API secret key*. The API secret key, as the name indicates, should be kept secret: do not ever include this in responses sent to the client. If a third party were to get access to this secret, they could make requests on behalf of your application, which could have unfortunate consequences for you if the use is malicious.

Now that we have a consumer API key and secret key, we can communicate with the Twitter REST API.

To keep our code tidy, we'll put our Twitter code in a module called *lib/twitter.js*:

```
const https = require('https')

module.exports = twitterOptions => {

 return {

  search: async (query, count) => {
    // TODO
  }
 }
}
```

This pattern should be starting to become familiar to you. Our module exports a function into which the caller passes a configuration object. What's returned is an object containing methods. In this way, we can add functionality to our module. Currently, we're only providing a `search` method. Here's how we will be using the library:

```
const twitter = require('./lib/twitter')({
  consumerApiKey: credentials.twitter.consumerApiKey,
  apiSecretKey: credentials.twitter.apiSecretKey,
})

const tweets = await twitter.search('#Oregon #travel', 10)
// tweets will be in result.statuses
```

(Don't forget to put a `twitter` property with `consumerApiKey` and `apiSecretKey` in your *.credentials.development.json* file.)

Before we implement the `search` method, we must provide some functionality to authenticate ourselves to Twitter. The process is simple: we use HTTPS to request an access token based on our consumer key and consumer secret. We only have to do this once: currently, Twitter does not expire access tokens (though you can invalidate them manually). Since we don't want to request an access token every time, we'll cache the access token so we can reuse it.

The way we've constructed our module allows us to create private functionality that's not available to the caller. Specifically, the only thing that's available to the caller is `module.exports`. Since we're returning a function, only that function is available to the caller. Calling that function results in an object, and only the properties of that object are available to the caller. So we're going to create a variable `accessToken`, which we'll use to cache our access token, and a `getAccessToken` function that will get the access token. The first time it's called, it will make a Twitter API request to get the access token. Subsequent calls will simply return the value of `accessToken`:

```
const https = require('https')

module.exports = function(twitterOptions) {

  // this variable will be invisible outside of this module
  let accessToken = null

  // this function will be invisible outside of this module
  const getAccessToken = async () => {
    if(accessToken) return accessToken
    // TODO: get access token
  }

  return {
    search: async (query, count) => {
      // TODO
    }
  }

}
```

We mark `getAccessToken` as async because we may have to make an HTTP request to the Twitter API (if there isn't a cached token). Now that we've established the basic structure, let's implement `getAccessToken`:

```
const getAccessToken = async () => {
  if(accessToken) return accessToken

  const bearerToken = Buffer(
    encodeURIComponent(twitterOptions.consumerApiKey) + ':' +
    encodeURIComponent(twitterOptions.apiSecretKey)
  ).toString('base64')

  const options = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'POST',
    path: '/oauth2/token?grant_type=client_credentials',
    headers: {
      'Authorization': 'Basic ' + bearerToken,
    },
  }

  return new Promise((resolve, reject) =>
    https.request(options, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => {
        const auth = JSON.parse(data)
        if(auth.token_type !== 'bearer')
          return reject(new Error('Twitter auth failed.'))
        accessToken = auth.access_token
        return resolve(accessToken)
      })
    }).end()
  )
}
```

The details of constructing this call are available on Twitter's developer documentation page for application-only authentication. Basically, we have to construct a bearer token that's a base64-encoded combination of the consumer key and consumer secret. Once we've constructed that token, we can call the `/oauth2/token` API with the `Authorization` header containing the bearer token to request an access token. Note that we must use HTTPS: if you attempt to make this call over HTTP, you are transmitting your secret key unencrypted, and the API will simply hang up on you.

Once we receive the full response from the API (we listen for the end event of the response stream), we can parse the JSON, make sure the token type is bearer, and be on our merry way. We cache the access token, and then invoke the callback.

Now that we have a mechanism for obtaining an access token, we can make API calls. So let's implement our search method:

```
search: async (query, count) => {
  const accessToken = await getAccessToken()
  const options = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'GET',
    path: '/1.1/search/tweets.json?q=' +
      encodeURIComponent(query) +
      '&count=' + (count || 10),
    headers: {
      'Authorization': 'Bearer ' + accessToken,
    },
  }
  return new Promise((resolve, reject) =>
    https.request(options, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => resolve(JSON.parse(data)))
    }).end()
  )
},
```

Rendering Tweets

Now we have the ability to search tweets…so how do we display them on our site? Largely, it's up to you, but there are some things to consider. Twitter has an interest in making sure its data is used in a manner consistent with the brand. To that end, it does have display requirements, which employ functional elements you must include to display a tweet.

There is some wiggle room in the requirements (for example, if you're displaying on a device that doesn't support images, you don't have to include the avatar image), but for the most part, you'll end up with something that looks very much like an embedded tweet. It's a lot of work, and there is a way around it…but it involves linking to Twitter's widget library, which is the very HTTP request we're trying to avoid.

If you need to display tweets, your best bet is to use the Twitter widget library, even though it incurs an extra HTTP request. For more complicated use of the API, you'll still have to access the REST API from the backend, so you will probably end up using the REST API in concert with frontend scripts.

Let's continue with our example: we want to display the top 10 tweets that mention the hashtags #Oregon #travel. We'll use the REST API to search for the tweets and the Twitter widget library to display them. Since we don't want to run up against usage limits (or slow down our server), we'll cache the tweets and the HTML to display them for 15 minutes.

We'll start by modifying our Twitter library to include a method `embed`, which gets the HTML to display a tweet. Note we're using an npm library `querystringify` to construct a querystring from an object, so don't forget to `npm install querystringify` and import it (`const qs = require( ‘querystringify ’)`), and then add the following function to the export of *lib/twitter.js*:

```
embed: async (url, options = {}) => {
  options.url = url
  const accessToken = await getAccessToken()
  const requestOptions = {
    hostname: 'api.twitter.com',
    port: 443,
    method: 'GET',
    path: '/1.1/statuses/oembed.json?' + qs.stringify(options),
    headers: {
      'Authorization': 'Bearer ' + accessToken,
    },
  }
  return new Promise((resolve, reject) =>
    https.request(requestOptions, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => resolve(JSON.parse(data)))
    }).end()
  )
},
```

Now we're ready to search for, and cache, tweets. In our main application file, create the following function `getTopTweets`:

```
const twitterClient = createTwitterClient(credentials.twitter)

const getTopTweets = ((twitterClient, search) => {
  const topTweets = {
    count: 10,
    lastRefreshed: 0,
    refreshInterval: 15 * 60 * 1000,
    tweets: [],
  }
  return async () => {
    if(Date.now() > topTweets.lastRefreshed + topTweets.refreshInterval) {
      const tweets =
       await twitterClient.search('#Oregon #travel', topTweets.count)
      const formattedTweets = await Promise.all(
        tweets.statuses.map(async ({ id_str, user }) => {
          const url = `https://twitter.com/${user.id_str}/statuses/${id_str}`
          const embeddedTweet =
           await twitterClient.embed(url, { omit_script: 1 })
          return embeddedTweet.html
        })
      )
      topTweets.lastRefreshed = Date.now()
      topTweets.tweets = formattedTweets
    }
    return topTweets.tweets
  }
})(twitterClient, '#Oregon #travel')
```

The essence of the getTopTweets function is to not just search for tweets with a specified hashtag, but to cache those tweets for some reasonable period of time. Note that we created an immediately invoked function expression, or IIFE: that's because we want the topTweets cache safely inside a closure so it can't be messed with. The asynchronous function that's returned from the IIFE refreshes the cache if necessary, and then returns the contents of the cache.

Lastly, let's create a view, views/social.handlebars as a home for our social media presence (which right now, includes only our selected tweets):

```
<h2>Oregon Travel in Social Media</h2>

<script id="twitter-wjs" type="text/javascript"
  async defer src="//platform.twitter.com/widgets.js"></script>

{{{tweets}}}
```

And a route to handle it:

```
app.get('/social', async (req, res) => {
  res.render('social', { tweets: await getTopTweets() })
})
```

Note that we reference an external script, Twitter's widgets.js. This is the script that will format and give functionality to the embedded tweets on your page. By default, the oembed API will include a reference to this script in the HTML, but since we're displaying 10 tweets, that would reference that script nine more times than necessary! So recall that, when we called the oembed API, we passed in the option { omit_script: 1 }. Since we did that, we have to provide it somewhere, which we did in the view. Go ahead and try removing the script from the view. You'll still see the tweets, but they won't have any formatting or functionality.

Now we have a nice social media feed! Let's turn our attention to another important application: displaying maps in our application.

Geocoding

Geocoding refers to the process of taking a street address or place name (Bletchley Park, Sherwood Drive, Bletchley, Milton Keynes MK3 6EB, UK) and converting it to geographic coordinates (latitude 51.9976597, longitude –0.7406863). If your application is going to be doing any kind of geographic calculation—distances or directions —or displaying a map, then you'll need geographic coordinates.

> You may be used to seeing geographic coordinates specified in degrees, minutes, and seconds (DMS). Geocoding APIs and mapping services use a single floating-point number for latitude and longitude. If you need to display DMS coordinates, see this wikipedia article.

Geocoding with Google

Both Google and Bing offer excellent REST services for geocoding. We'll be using Google for our example, but the Bing service is very similar.

Without attaching a billing account to your Google account, your geocoding requests will be limited to one a day, which will make for a very slow testing cycle indeed! Whenever possible in this book, I've tried to avoid recommending services you couldn't at least use in a development capacity for free, and I did try some free geocoding services, and found a significant enough gulf in usability that I continue to recommend Google geocoding. However, as I write this, the cost of development-volume geocoding with Google is free: you receive a $200 monthly credit with your account, and you would have to make 40,000 requests to exhaust that! If you want to follow along with this chapter, go to your Google console, choose Billing from the main menu, and enter your billing information.

Once you've set up billing, you'll need an API key for Google's geocoding API. Go to the console, select your project from the navigation bar, and then click on APIs. If the geocoding API isn't in your list of enabled APIs, locate it in the list of additional APIs and add it. Most of the Google APIs share the same API credentials, so click on the navigation menu in the upper left, and go back to your dashboard. Click on Credentials, and create a new API key if you don't have an appropriate one already. Note that API keys can be restricted to prevent abuse, so make sure your API key can be used from your application. If you need one for developing, you can restrict the key by IP address, and choose your IP address (if you don't know what it is, you can just ask Google, "What's my IP address?").

Once you have an API key, add it to *.credentials.development.json*:

```
"google": {
  "apiKey": "<YOUR API KEY>"
}
```

Then create a module *lib/geocode.js*:

```
const https = require('https')
const { credentials } = require('../config')

module.exports = async query => {

  const options = {
    hostname: 'maps.googleapis.com',
    path: '/maps/api/geocode/json?address=' +
      encodeURIComponent(query) + '&key=' +
      credentials.google.apiKey,
  }

  return new Promise((resolve, reject) =>
    https.request(options, res => {
      let data = ''
      res.on('data', chunk => data += chunk)
      res.on('end', () => {
        data = JSON.parse(data)
        if(!data.results.length)
          return reject(new Error(`no results for "${query}"`))
        resolve(data.results[0].geometry.location)
      })
    }).end()
  )

}
```

Now we have a function that will contact the Google API to geocode an address. If it can't find an address (or fails for any other reason), an error will be returned. The API can return multiple addresses. For example, if you search for "10 Main Street" without specifying a city, state, or postal code, it will return dozens of results. Our implementation simply picks the first one. The API returns a lot of information, but all we're currently interested in are the coordinates. You could easily modify this interface to return more information. See the Google geocoding API documentation for more information about the data the API returns.

Usage restrictions

The Google geocoding API currently has a monthly usage limit, but you pay $0.005 per geocoding request. So if you made a million requests in any given month, you'd have a $5,000 bill from Google…so there is a probably practical limit for you!

If you're worried about runaway charges—which could happen if you accidentally leave a service running, or if a bad actor gets access to your credentials—you can add a budget and configure alerts to notify you as you approach them. Go to your Google developer console, and choose "Budgets & alerts" from the Billing menu.

At the time of writing, Google limits you to 5,000 requests per 100 seconds to prevent abuse, which would be difficult to exceed. Google's API also requires that if you use a map on your website, you use Google Maps. That is, if you're using Google's service to geocode your data, you can't turn around and display that information on a Bing map without violating the terms of service. Generally, this is not an onerous restriction, as you probably wouldn't be doing geocoding unless you intended to display locations on a map. However, if you like Bing's maps better than Google's, or vice versa, you should be mindful of the terms of service and use the appropriate API.

Geocoding Your Data

We have a nice database of vacation packages around Oregon, and we might decide we want to display a map with pins showing where the various vacations are, and this is where geocoding comes in.

We already have vacation data in the database, and each vacation has a location search string that will work with geocoding, but we don't yet have coordinates.

The question now is when and how do we do the geocoding? Broadly speaking, we have three options:

- Geocode when we add new vacations to the database. This is probably a great option when we add an admin interface to the system that allows vendors to dynamically add vacations to the database. Since we're stopping short of this functionality, however, we'll discard this option.
- Geocode as necessary when retrieving vacations from the database. This approach would do a check every time we get vacations from the database: if any of them have missing coordinates, we would geocode them. This option sounds appealing, and is probably the easiest of the three, but it has some big disadvantages that make it unsuitable. The first is performance: if you add a thousand new vacations to the database, the first person to look at the vacations list is going to have to wait for all of those geocoding requests to succeed and get written to the database. Furthermore, one can imagine a situation where a load testing suite adds a thousand vacations to the database and then performs a thousand requests. Since they all run concurrently, each of those thousand requests results in a thousand geocoding requests because the data hasn't been written to the

database yet…resulting in a million geocoding requests and a $5,000 bill from Google! So let's cross this one off the list.

- Have a script to find vacations with missing coordinate date, and geocode those. This approach offers the best solution for our current situation. For development purposes, we're populating the vacation database once, and we don't yet have an admin interface for adding new vacations. Furthermore, if we do decide to add an admin interface later, this approach isn't incompatible with that: as a matter of fact, we could just run this process after adding a new vacation, and it would work.

First, we need to add a way to update an existing vacation in *db.js* (we'll also add a method to close the database connection, which will come in handy in scripts):

```
module.exports = {
  //...
  updateVacationBySku: async (sku, data) => Vacation.updateOne({ sku }, data),
  close: () => mongoose.connection.close(),
}
```

Then we can write a script *db-geocode.js*:

```
const db = require('./db')
const geocode = require('./lib/geocode')

const geocodeVacations = async () => {
  const vacations = await db.getVacations()
  const vacationsWithoutCoordinates = vacations.filter(({ location }) =>
    !location.coordinates || typeof location.coordinates.lat !== 'number')
  console.log(`geocoding ${vacationsWithoutCoordinates.length} ` +
    `of ${vacations.length} vacations:`)
  return Promise.all(vacationsWithoutCoordinates.map(async ({ sku, location }) => {
    const { search } = location
    if(typeof search !== 'string' || !/\w/.test(search))
      return console.log(`  SKU ${sku} FAILED: does not have location.search`)
    try {
      const coordinates = await geocode(search)
      await db.updateVacationBySku(sku, { location: { search, coordinates } })
      console.log(`  SKU ${sku} SUCCEEDED: ${coordinates.lat}, ${coordinates.lng}`)
    } catch(err) {
      return console.log(`  SKU {sku} FAILED: ${err.message}`)
    }
  }))
}

geocodeVacations()
  .then(() => {
    console.log('DONE')
    db.close()
  })
  .catch(err => {
    console.error('ERROR: ' + err.message)
    db.close()
  })
```

When you run the script (`node db-geocode.js`), you should see that all of your vacations have been successfully geocoded! Now that we have that information, let's learn how to display it on a map….

Displaying a Map

While displaying vacations on a map really falls under "frontend" work, it would be very disappointing to get this far and not see the fruits of our labor. So we're going to take a slight departure from the backend focus of this book, and see how to display our newly geocoded dealers on a map.

We already created a Google API key to do our geocoding, but we still need to enable the maps API. Go to your Google console, click on APIs, and find Maps JavaScript API and enable it if it isn't already.

Now we can create a view to display our vacations map, *views/vacations-map.handlebars*. We'll start with just displaying the map, and work on adding vacations next:

```
<div id="map" style="width: 100%; height: 60vh;"></div>
<script>
  let map = undefined
  async function initMap() {
    map = new google.maps.Map(document.getElementById('map'), {
      // approximate geographic center of oregon
      center: { lat: 44.0978126, lng: -120.0963654 },
      // this zoom level covers most of the state
      zoom: 7,
    })
  }
</script>
<script src="https://maps.googleapis.com/maps/api/js?key={{googleApiKey}}&callback=initMap"
    async defer></script>
```

Now it's time to put some pins on the map corresponding with our vacations. In Chapter 15, we created an API endpoint `/api/vacations`, which will now include geocoded data. We'll use that endpoint to get our vacations, and put pins on the map. Modify the `initMap` function in *views/vacations-map.handlebars.js*:

```
async function initMap() {
  map = new google.maps.Map(document.getElementById('map'), {
    // approximate geographic center of oregon
    center: { lat: 44.0978126, lng: -120.0963654 },
    // this zoom level covers most of the state
    zoom: 7,
  })
```

```
    const vacations = await fetch('/api/vacations').then(res => res.json())
    vacations.forEach(({ name, location }) => {
      const marker = new google.maps.Marker({
        position: location.coordinates,
        map,
        title: name,
      })
    })
  }
```

Now we have a map showing where all our vacations are! There are a lot of ways we could improve this page: probably the best place to start would be to linking the markers with the vacation detail page, so you could click on a marker and it would take you to the vacation info page. We could also implement custom markers or tooltips: the Google Maps API has a lot of features, and you can learn about them from the official Google documentation.

Weather Data

Remember our "current weather" widget from Chapter 7? Let's get that hooked up with some live data! We'll be using the US National Weather Service (NWS) API to get forecast information. As with our Twitter integration, and our use of geocoding, we'll be caching the forecast to prevent passing every hit to our website on to NWS (which might get us blacklisted if our website gets popular). Create a file called *lib/weather.js*:

```
const https = require('https')
const { URL } = require('url')

const _fetch = url => new Promise((resolve, reject) => {
  const { hostname, pathname, search } = new URL(url)
  const options = {
    hostname,
    path: pathname + search,
    headers: {
      'User-Agent': 'Meadowlark Travel'
    },
  }
  https.get(options, res => {
    let data = ''
    res.on('data', chunk => data += chunk)
    res.on('end', () => resolve(JSON.parse(data)))
  }).end()
})

module.exports = locations => {

  const cache = {
    refreshFrequency: 15 * 60 * 1000,
    lastRefreshed: 0,
    refreshing: false,
    forecasts: locations.map(location => ({ location })),
  }

  const updateForecast = async forecast => {
    if(!forecast.url) {
      const { lat, lng } = forecast.location.coordinates
      const path = `/points/${lat.toFixed(4)},${lng.toFixed(4)}`
      const points = await _fetch('https://api.weather.gov' + path)
      forecast.url = points.properties.forecast
    }
    const { properties: { periods } } = await _fetch(forecast.url)
    const currentPeriod = periods[0]
    Object.assign(forecast, {
      iconUrl: currentPeriod.icon,
      weather: currentPeriod.shortForecast,
      temp: currentPeriod.temperature + ' ' + currentPeriod.temperatureUnit,
    })
    return forecast
  }

  const getForecasts = async () => {
    if(Date.now() > cache.lastRefreshed + cache.refreshFrequency) {
      console.log('updating cache')
      cache.refreshing = true
      cache.forecasts = await Promise.all(cache.forecasts.map(updateForecast))
      cache.refreshing = false
    }
    return cache.forecasts
  }

  return getForecasts

}
```

You'll notice that we got tired of using Node's built-in `https` library directly, and instead created a utility function `_fetch` to make our weather functionality a little more readable. One thing that might jump out at you is that we're setting the `User-Agent` header to `Meadowlark Travel`. This is a quirk of the NWS weather API: it requires a string for the `User-Agent`. They state that they will eventually replace this with an API key, but for now we just need to provide a value here.

Getting weather data from the NWS API is a two-part affair here. There's an API endpoint called `points` that takes a latitude and longitude (with exactly four decimal digits) and returns information about that location…including the appropriate URL from which to get a forecast. Once we have that URL for any given set of coordinates, we don't need to fetch it again. All we need to do is call that URL to get the updated forecast.

Note that a lot more data is returned from the forecast than we're using; we could get a lot more sophisticated with this feature. In particular, the forecast URL returns an array of periods, with the first element being the current period (for example, "afternoon" or "evening"). It follows up with periods stretching into the next week. Feel free to look at the data in the `periods` array to see the kind of data that's available to you.

One detail worth pointing out is that we have a boolean property in our cache called `refreshing`. This is necessary since updating the cache takes a finite amount of time, and is done asynchronously. If multiple requests come in before the first cache refresh completes, they will all kick off the work of refreshing the cache. It won't hurt anything, exactly, but you will be making more API calls than are strictly necessary. This boolean variable is just a flag to any additional requests to say, "We're working on it."

We've designed this to be a drop-in replacement for the dummy function we created back in Chapter 7. All we have to do is open *lib/middleware/weather.js* and replace the `getWeatherData` function:

```
const weatherData = require('../weather')

const getWeatherData = weatherData([
  {
    name: 'Portland',
    coordinates: { lat: 45.5154586, lng: -122.6793461 },
  },
  {
    name: 'Bend',
    coordinates: { lat: 44.0581728, lng: -121.3153096 },
  },
  {
    name: 'Manzanita',
    coordinates: { lat: 45.7184398, lng: -123.9351354 },
  },
])
```

Now we have live weather data in our widget!

Conclusion

We've really only scratched the surface of what can be done with third-party API integration. Everywhere you look, new APIs are popping up, offering every kind of data imaginable (even the City of Portland is now making a lot of public data available through REST APIs). While it would be impossible to cover even a small percentage of the APIs available to you, this chapter has covered the fundamentals you'll need to know to use these APIs: `http.request`, `https.request`, and parsing JSON.

We now have a lot of knowledge under our belt. We've covered a lot of ground! What happens when things go wrong, though? In the next chapter, we'll be discussing debugging techniques to help us when things don't work out as we expect.

CHAPTER 20
Debugging

"Debugging" is perhaps an unfortunate term, what with its association with defects. The fact is, what we refer to as "debugging" is an activity you will find yourself doing all the time, whether you're implementing a new feature, learning how something works, or actually fixing a bug. A better term might be "exploring," but we'll stick with "debugging," since the activity it refers to is unambiguous, regardless of the motivation.

Debugging is an oft-neglected skill: it seems that most programmers are expected to be born knowing how to do it. Perhaps computer science professors and book authors see debugging as such an obvious skill that they overlook it.

The fact is, debugging is a skill that can be taught, and it is an important way by which programmers come to understand not just the framework they are working in, but also their own code and that of their team. In this chapter, we'll discuss some of the tools and techniques you can use for debugging Node and Express applications effectively.

The First Principle of Debugging

As the name implies, "debugging" often refers to the process of finding and eliminating defects. Before we talk about tools, let's consider some general debugging principles.

> How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?
>
> —Sir Arthur Conan Doyle

The first and most important principle of debugging is the process of *elimination*. Modern computer systems are incredibly complicated, and if you had to hold the *whole system* in your head, and pluck the source of a single problem out of that vast space, you probably wouldn't even know where to start. Whenever you're confronted with a problem that isn't immediately obvious, your *very first thought* should be, "What can I *eliminate* as the source of the problem?" The more you can eliminate, the fewer places you have to look.

Elimination can take many forms. Here are some common examples:

- Systematically commenting out or disabling blocks of code.
- Writing code that can be covered by unit tests; the unit tests themselves provide a framework for elimination.
- Analyzing network traffic to determine if the problem is on the client or server side.
- Testing a different part of the system that has similarities to the first.
- Using input that has worked before, and changing that input one piece at a time until the problem exhibits.
- Using version control to go back and forth in time until the problem disappears, and you can isolate it to a particular change (see `git bisect` for more information about this).
- "Mocking" functionality to eliminate complex subsystems.

Elimination is not a silver bullet, though. Often, problems are due to complex interactions between two or more components: eliminate (or mock) any one of the components, and the problem could go away, but the problem can't be isolated to any single component. Even in this situation, though, elimination can help narrow down the problem, even if it doesn't light up a neon sign over the exact location.

Elimination is most successful when it's careful and methodical. It's very easy to miss things when you just wantonly eliminate components without considering how those components affect the whole. Play a game with yourself: when you consider a component to eliminate, walk through how the removal of that component will affect the system. This will inform you about what to expect and whether or not removing the component tells you anything useful.

Take Advantage of REPL and the Console

Both Node and your browser offer you a *read-eval-print loop* (REPL); this is basically just a way to write JavaScript interactively. You type in some JavaScript, press Enter, and immediately see the output. It's a great way to play around, and is often the quickest and most intuitive way to locate an error in small bits of code.

In a browser, all you have to do is pull up your JavaScript console, and you have a REPL. In Node, all you have to do is type `node` without any arguments, and you enter REPL mode; you can require packages, create variables and functions, or do anything else you could normally do in your code (except create packages: there's no meaningful way to do that in the REPL).

Console logging is also your friend. It's a crude debugging technique, perhaps, but an easy one (both easy to understand and easy to implement). Calling `console.log` in Node will output the contents of an object in an easy-to-read format, so you can easily spot problems. Keep in mind that some objects are so large that logging them to the console will produce so much output that you'll have a hard time finding any useful information. For example, try `console.log(req)` in one of your path handlers.

Using Node's Built-in Debugger

Node has a built-in debugger that allows you to step through your application, as if you were going on a ride-along with the JavaScript interpreter. All you have to do to start debugging your app is use the `inspect` argument:

```
node inspect meadowlark.js
```

When you do, you'll immediately notice a couple of things. First, on your console you will see a URL; this is because the Node debugger works by creating its own web server, which allows you to control the execution of the application being debugged. This may not be impressive right now, but the usefulness of this approach will be clear when we discuss inspector clients.

When you're in the console debugger, you can type `help` to get a list of commands. The commands you will use most often are `n` (next), `s` (step in), and `o` (step out). `n` will step "over" the current line: it will execute it, but if that instruction calls other functions, they will be executed before control is returned to you. `s`, in contrast, will step *into* the current line: if that line invokes other functions, you will be able to step through them. `o` allows you to step out of the currently executing function. (Note that "stepping in" and "stepping out" refer only to *functions*; they do not step into or out of `if` or `for` blocks or other flow-control statements.)

The command-line debugger has more functionality, but chances are, you won't want to use it that often. The command line is great for many things, but debugging isn't one of them. It's good that it's available in a pinch (for example, if all you have is SSH access to the server, or if your server doesn't even have a GUI installed). More often, you'll want to use a graphical inspector client.

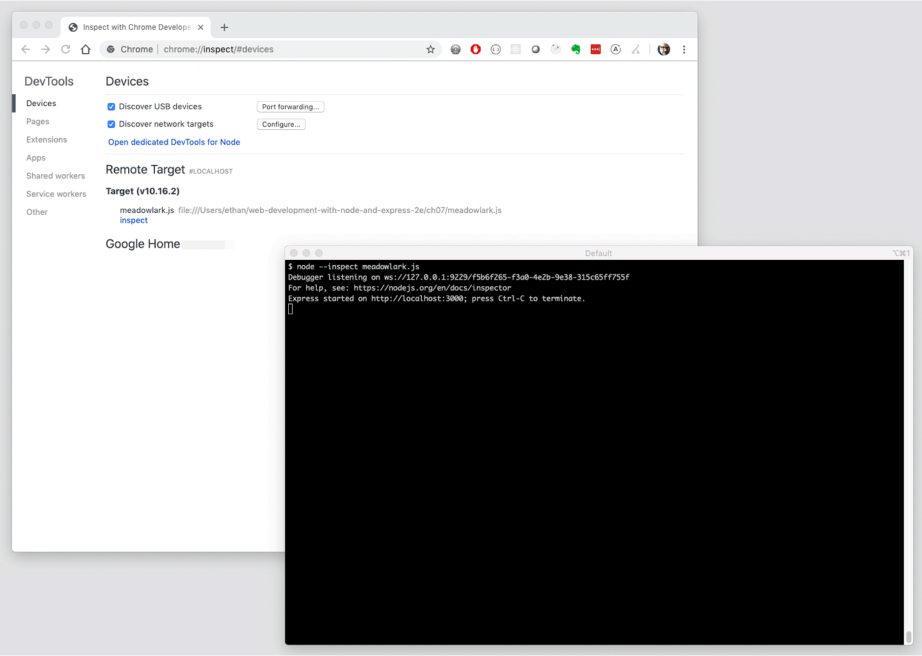
Node Inspector Clients

While you probably won't want to use the command-line debugger except in a pinch, the fact that Node exposes its debugging controls through a web service gives you other options.

The most straightforward debugger is to use Chrome, which uses the same debugging interface as it does for debugging frontend code. So if you've ever used that interface, you should feel right at home. Getting started is easy. Start your application with the `--inspect` option (which is distinct from the `inspect` argument mentioned previously):

```
node --inspect meadowlark.js
```

Now the fun begins: in your browser's URL bar, enter *`chrome://inspect`*. You'll see a DevTools page, and in the Devices section, click "Open dedicated DevTools for Node." This will open a new window, which is your debugger:



Devices

☑ Discover USB devices

☑ Discover network targets

Open dedicated DevTools for Node

Remote Target #LOCALHOST

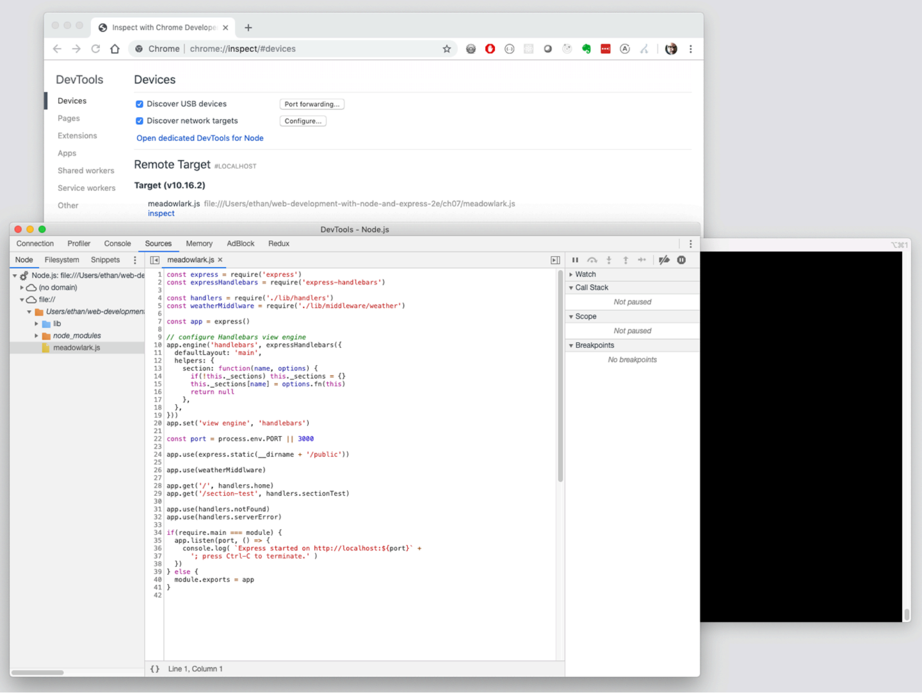
Target (v10.16.2)

meadowlark.js file:///Users/ethan/web-development-with-node-and-express-2e/ch07/meadowlark.js
inspect

Google Home

```
$ node --inspect meadowlark.js
Debugger listening on ws://127.0.0.1:9229/f5b6f265-f3a0-4e2b-9e38-315c65ff755f
For help, see: https://nodejs.org/en/docs/inspector
Express started on http://localhost:3000; press Ctrl-C to terminate.
```

Click the Sources tab, and then, in the leftmost pane, click Node.js to expand it, and then click "file://". You'll see the folder that your application is in; expand that, and you'll see all of your JavaScript source (you'll only see JavaScript and sometimes JSON files if you've required them somewhere). From here, you can click any file to see its source, and set breakpoints:



```
const express = require('express')
const expressHandlebars = require('express-handlebars')

const handlers = require('./lib/handlers')
const weatherMiddlware = require('./lib/middleware/weather')

const app = express()

// configure Handlebars view engine
app.engine('handlebars', expressHandlebars({
  defaultLayout: 'main',
  helpers: {
    section: function(name, options) {
      if(!this._sections) this._sections = {}
      this._sections[name] = options.fn(this)
      return null
    },
  },
}))
app.set('view engine', 'handlebars')

const port = process.env.PORT || 3000

app.use(express.static(__dirname + '/public'))

app.use(weatherMiddlware)

app.get('/', handlers.home)
app.get('/section-test', handlers.sectionTest)

app.use(handlers.notFound)
app.use(handlers.serverError)

if(require.main === module) {
  app.listen(port, () => {
    console.log( `Express started on http://localhost:${port}` +
      '; press Ctrl-C to terminate.' )
  })
} else {
  module.exports = app
}

```

Unlike our previous experience with the command-line debugger, your application is already running: all of the middleware has been linked in, and the app is listening. So how do we step through our code? The easiest way (and the method you'll probably use the most often) is to set a *breakpoint*. This just tells the debugger to stop execution on a specific line so you can step through the code.

All you have to do to set a breakpoint is to open a source file from the "file://" browser in the debugger, and click the line number (in the left column); a little blue arrow will appear, indicating there's a breakpoint on that line (click again to turn it off). Go ahead and set a breakpoint inside one of your route handlers. Then, in another browser window, visit that route. If you're using Chrome, the browser will automatically switch to the debugger window, while the original browser just spins (because the server has been paused and isn't responding to the request).

In the debugger window, you can step through the program in a much more visual manner than we did with the command-line debugger. You'll see that the line you set a breakpoint on is highlighted in blue. That means that's the current execution line (which is actually the next line that will execute). From here, you have access to the same commands as we did in the command-line debugger. Similar to the command-line debugger, we have the following actions available to us:

Resume script execution (F8)

This will simply "let it fly"; you will no longer be stepping through the code, unless you stop on another breakpoint. You usually use this when you've seen what you need to see, or you want to skip ahead to another breakpoint.

Step over next function call (F10)

If the current line invokes a function, the debugger will not descend into that function. That is, the function will be executed, and the debugger will advance to the next line after the function invocation. You'll use this when you're on a function call that you're not interested in the details of.

Step into next function call (F11)

This will descend into the function call, hiding nothing from you. If this is the only action you ever used, you would eventually see everything that gets executed —which sounds fun at first, but after you've been at it for an hour, you'll have a newfound respect for what Node and Express are doing for you!

Step out of current function (Shift-F11)

Will execute the rest of the function you're currently in and resume debugging on the next line of the *caller* of this function. Most commonly, you'll use this when you either accidentally step into a function or have seen as much as you need of the function.

In addition to all of the control actions, you have access to a console: that console is executing in the *current context of your application*. So you can inspect variables and even change them, or invoke functions. This can be incredibly handy for trying out really simple things, but it can quickly get confusing, so I don't encourage you to dynamically modify your running application too much in this manner; it's too easy to get lost.

On the right, you have some useful data. Starting at the top are *watch expressions*; these are JavaScript expressions you can define that will be updated in real time as you step through the application. For example, if there was a specific variable you wanted to keep track of, you could enter it here.

Below watch expressions is the *call stack*; this shows you how you got where you are. That is, the function you're in was called by some function, and that function was called by some function; the call stack lists all of those functions. In the highly asynchronous world of Node, the call stack can be very difficult to unravel and understand, especially when anonymous functions are involved. The topmost entry in that list is where you are now. The one right below it is the function that called the function that you're in now, and so on. If you click any entry in this list, you will be magically transported to that context: all of your watches and your console context will now be in that context.

Below the call stack are the scope variables. As the name implies, these are the variables that are currently in scope (which includes variables in the parent scope that are visible to us). This section can often provide you a lot of information about the key variables you're interested in at a glance. If you have a lot of variables, this list will become unwieldy, and you might be better off defining just the variables you're interested in as watch expressions.

Next, there is a list of all breakpoints, which is really just bookkeeping: it's handy to have if you're debugging a hairy problem and you have a lot of breakpoints set. Clicking one will take you directly there (but it won't change the context, like clicking something in the call stack; this makes sense because not every breakpoint will represent an active context, whereas everything in the call stack does).

Sometimes, what you need to debug is your application setup (when you're linking middleware into Express, for example). Running the debugger as we have been, that will all happen in the blink of an eye before we can even set a breakpoint. Fortunately, there's a way around that. All we have to do is specify `--inspect-brk` instead of simply `--inspect`:

```
node --inspect-brk meadowlark.js
```

The debugger will break on the very first line of your application, and then you can step through or set breakpoints as you see fit.

Chrome isn't your only option for an inspect client. In particular, if you use Visual Studio Code, its built-in debugger works very well. Instead of starting your application with the `--inspect` or `--inspect-brk` options, click the Debug icon in the Visual Studio Code side menu (a bug with a line through it). At the top of the sidebar, you'll see a little gear icon; click that, and that will open some debugging configuration settings. The only setting you need to worry about is "program"; make sure it's pointing to your entry point (*meadowlark.js*, for example).

> You may have to also set the current working directory, or `"cwd"`. For example, if you've opened Visual Studio Code in a parent directory of where *meadowlark.js* lives, you may need to set `"cwd"` (which is the same as having to `cd` into the right directory before running `node meadowlark.js`.

Once you're all set up, just click the green Play arrow in the debug bar, and your debugger is running. The interface is slightly different from Chrome's, but if you're using Visual Studio Code, you will probably feel right at home. For more information, see Debugging in Visual Studio Code.

Debugging Asynchronous Functions

One of the most common frustrations people have when being exposed to asynchronous programming for the first time is in debugging. Consider the following code, for example:

```
1 console.log('Baa, baa, black sheep,');
2 fs.readFile('yes_sir_yes_sir.txt', (err, data) => {
3         console.log('Have you any wool?');
4         console.log(data);
5 })
6 console.log('Three bags full.')
```

If you're new to asynchronous programming, you might expect to see the following:

```
Baa, baa, black sheep,
Have you any wool?
Yes, sir, yes, sir,
Three bags full.
```

But you won't; instead you'll see this:

```
Baa, baa, black sheep,
Three bags full.
Have you any wool?
Yes, sir, yes, sir,
```

If you're confused about this, debugging probably won't help. You'll start on line 1, then step over it, which puts you on line 2. You then step in, expecting to enter the function, ending up on line 3, but you actually end up on line 5! That's because `fs.readFile` executes the function only *when it's done reading the file*, which won't happen until your application is idle. So you step over line 5, and you land on line 6… you then keep trying to step, but never get to line 3 (you eventually will, but it could take a while).

If you want to debug lines 3 or 4, all you have to do is set a breakpoint on line 3, and then let the debugger run. When the file is read and the function is invoked, you'll break on that line, and hopefully all will be clear.

Debugging Express

If, like me, you've seen a lot of overengineered frameworks in your career, the idea of stepping through the framework source code might sound like madness (or torture) to you. And exploring the Express source code is not child's play, but it *is* well within the grasp of anyone with a good understanding of JavaScript and Node. And sometimes, when you are having problems with your code, debugging those problems can best be solved by stepping through the Express source code itself (or third-party middleware).

This section will be a brief tour of the Express source code so that you can be more effective in debugging your Express applications. For each part of the tour, I will give you the filename with respect to the Express root (which you can find in your *node_modules/express* directory), and the name of the function. I'm not using line numbers, because of course they may differ depending on what exact version of Express you're using:

Express app creation (lib/express.js, `function createApplication`*)*
: This is where your Express app begins its life. This is the function that's being invoked when you call `const app = express()` in your code.

Express app initialization (lib/application.js, `app.defaultConfiguration`*)*
: This is where Express gets initialized: it's a good place to see all the defaults Express starts out with. It's rarely necessary to set a breakpoint here, but it is useful to step through it at least once to get a feel for the default Express settings.

Add middleware (lib/application.js, `app.use`*)*
: Every time Express links middleware in (whether you do it explicitly, or it's explicitly done by Express or any third parties), this function gets called. It's deceptively simple, but really understanding it takes some effort. It's sometimes useful to put a breakpoint in here (you'll want to use `--debug-brk` when you run your app; otherwise, all the middleware will be added before you can set a breakpoint), but it can be overwhelming: you'll be surprised at how much middleware is linked in in a typical application.

Render view (lib/application.js, `app.render`*)*
: This is another pretty meaty function, but a useful one if you need to debug tricky view-related issues. If you step through this function, you'll see how the view engine is selected and invoked.

Request extensions (lib/request.js)
: You will probably be surprised at how sparse and easy to understand this file is. Most of the methods Express adds to the request objects are very simple convenience functions. It's rarely necessary to step through this code or set breakpoints because of the simplicity of the code. It is, however, often helpful to look at this code to understand how some of the Express convenience methods work.

Send response (lib/response.js, `res.send`*)*
: It almost doesn't matter how you construct a response—`.send`, `.render`, `.json`, or `.jsonp`—it will eventually get to this function (the exception is `.sendFile`). So this is a handy place to set a breakpoint, because it should be called for every response. You can then use the call stack to see how you got there, which can be helpful in figuring out where there might be a problem.

Response extensions (lib/response.js)
: While there is some meat in `res.send`, most of the other methods in the response object are pretty straightforward. It's occasionally useful to put breakpoints in these functions to see exactly how your app is responding to the request.

Static middleware (node_modules/serve-static/index.js, `function staticMiddleware`*)*
: Generally, if static files aren't being served as you expect, the problem is with your routing, not with the static middleware: routing takes precedence over the static middleware. So if you have a file *public/test.jpg*, and a route */test.jpg*, the static middleware will never even get called in deference to the route. However, if you need specifics about how headers are set differently for static files, it can be useful to step through the static middleware.

If you're scratching your head wondering where all the middleware is, that's because there is very little middleware in Express (the static middleware and the router being the notable exceptions).

Just as it's helpful to dive into the Express source code when you're trying to unravel a difficult problem, you may have to look into the source code of your middleware. There's really too much to go through, but there are three I want to mention as being pretty fundamental to understanding what's going on in an Express application:

Session middleware (node_modules/express-session/index.js, `function session`*)*
: A lot goes into making sessions work, but the code is pretty straightforward. You may want to set a breakpoint in this function if you're having issues that are related to sessions. Keep in mind that it is up to you to provide the storage engine for the session middleware.

Logger middleware (node_modules/morgan/index.js, `function logger`*)*
: The logger middleware is really there for you as a debugging aid, not to be debugged itself. However, there's some subtlety to the way logging works that you'll get only by stepping through the logger middleware once or twice. The first time I did it, I had a lot of "aha" moments, and found myself using logging more effectively in my applications, so I recommend taking a tour of this middleware at least once.

URL-encoded body parsing (node_modules/body-parser/lib/types/urlencoded.js, `function urlencoded`*)*
: The manner in which request bodies are parsed is often a mystery to people. It's not really that complicated, and stepping through this middleware will help you understand the way HTTP requests work. Aside from a learning experience, you won't find that you need to step into this middleware for debugging very often.

Conclusion

We've discussed a *lot* of middleware in this book. I can't reasonably list every landmark you might want to look at on your tour of Express internals, but hopefully these highlights take away some of the mystery of Express, and embolden you to explore the framework source code whenever needed. Middleware varies greatly not just in quality but in accessibility: some middleware is wickedly difficult to understand, while some is as clear as a pool of water. Whatever the case, don't be afraid to look: if it's too complicated, you can move on (unless you really need to understand it, of course), and if not, you might learn something.

CHAPTER 21
Going Live

The big day is here: you've spent weeks or months toiling over your labor of love, and now your website or service is ready to launch. It's not as easy as just "flipping a switch" and then your website is live…or is it?

In this chapter (which you should really read *weeks* before launch, not the day of!), you'll learn about some of the domain registration and hosting services available to you, techniques for moving from a staging environment to production, deployment techniques, and things to consider when picking production services.

Domain Registration and Hosting

People are often confused about the difference between *domain registration* and *hosting*. If you're reading this book, you probably aren't, but I bet you know people who are, like your clients or your manager.

Every website and service on the internet can be identified by an *Internet Protocol (IP) address* (or more than one). These numbers are not particularly friendly to humans (and that situation will only get worse as IPv6 adoption improves), but your computer ultimately needs these numbers to show you a web page. That's where *domain names* come in. They map a human-friendly name (like *google.com*) with an IP address (74.125.239.13 or 2601:1c2:1902:5b38:c256:27ff:fe70:47d1).

A real-world analogy would be the difference between a business name and a physical address. A domain name is like your business name (Apple), and an IP address is like your physical address (One Apple Park Way, Cupertino, CA 95014). If you need to actually get in your car and visit Apple's headquarters, you'll need to know the physical address. Fortunately, if you know the business name, you can probably get the physical address. The other reason this abstraction is helpful is that an organization can move (getting a new physical address), and people can still find it even though it's moved (as a matter of fact, Apple *did* move its physical headquarters between the first and second editions of this book).

Hosting, on the other hand, describes the computers that run your website. To continue the physical analogy, hosting could be compared to the buildings you see once you reach the physical address. What is often confusing to people is that domain registration has very little to do with hosting, and you do not always purchase your domain from the same entity that you pay for hosting (in the same way that you usually buy land from one person and pay another person to build and maintain buildings for you).

While it's certainly possible to host your website without a domain name, it's quite unfriendly: IP addresses aren't very marketable! Usually, when you purchase hosting, you're automatically assigned a subdomain (which we'll cover in a moment), which can be thought of as something between a marketing-friendly domain name and an IP address (for example, *ec2-54-201-235-192.us-west-2.compute.amazonaws.com*).

Once you have a domain, and you go live, you could reach your website with multiple URLs. For example:

- *http://meadowlarktravel.com/*
- *http://www.meadowlarktravel.com/*
- *http://ec2-54-201-235-192.us-west-2.compute.amazonaws.com/*
- *http://54.201.235.192/*

Thanks to domain mapping, all of these addresses point to the same website. Once the requests reach your website, it is possible to take action based on the URL that was used. For example, if someone gets to your website from the IP address, you could automatically redirect to the domain name, though that is not very common as there is little point to it (it is more common to redirect from *http://meadowlarktravel.com/* to *http://www.meadowlarktravel.com/*).

Most domain registrars offer hosting services (or partner with companies that do). Aside from AWS, I've never found registrar hosting options to be particularly attractive, and it's okay to separate domain registration and hosting.

Domain Name System

The *Domain Name System* (DNS) is what's responsible for mapping domain names to IP addresses. The system is fairly intricate, but there are some things about DNS that you should know as a website owner.

Security

You should always keep in mind that *domain names are valuable*. If a hacker were to completely compromise your hosting service and take control of your hosting, but you retained control of your domain, you could get new hosting and redirect the domain. If, on the other hand, your *domain* were compromised, you could be in real trouble. Your reputation is tied to your domain, and good domain names are carefully guarded. People who have lost control of domains have found that it can be devastating, and there are those in the world who will actively try to compromise your domain (especially if it's a particularly short or memorable one) so they can sell it off, ruin your reputation, or blackmail you. The upshot is that *you should take domain security very seriously*, perhaps even more seriously than your data (depending on how valuable your data is). I've seen people spend inordinate amounts of time and money on hosting security while getting the cheapest, sketchiest domain registration they can find. Don't make that mistake. (Fortunately, quality domain registration is not particularly expensive.)

Given the importance of protecting ownership of your domain, you should employ good security practices with respect to your domain registration. At the very least, you should use strong, unique passwords, and employ proper password hygiene (no keeping it on a sticky note attached to your monitor). Preferably, you should use a registrar that offers two-factor authentication. Don't be afraid to ask your registrar pointed questions about what is required to authorize changes to your account. The registrars I recommend are AWS Route 53, Name.com and Namecheap.com. All three offer two-factor authentication, and I have found their support to be good and their online control panels to be easy and robust.

When you register a domain, you must provide a third-party email address that's associated with that domain (i.e., if you're registering *meadowlarktravel.com*, you shouldn't use *admin@meadowlarktravel.com* as your registrant email). Since any security system is as strong as its weakest link, you should use an email address with good security. It's quite common to use a Gmail or Outlook account, and if you do, you should employ the same security standards as you do with your domain registrar account (good password hygiene and two-factor authentication).

Top-Level Domains

What your domain ends with (such as *.com* or *.net*) is called a *top-level-domain* (TLD). Generally speaking, there are two types of TLD: country code TLDs and general TLDs. Country code TLDs (such as *.us*, *.es*, and *.uk*) are designed to provide a geographic categorization. However, there are few restrictions on who can acquire these TLDs (the internet is truly a global network, after all), so they are often used for "clever" domains, such as *placehold.it* and *goo.gl*.

General TLDs (gTLDs) include the familiar *.com*, *.net*, *.gov*, *.fed*, *.mil*, and *.edu*. While anyone can acquire an available *.com* or *.net* domain, there are restrictions in place for the others mentioned. For more information, see Table 21-1.

Table 21-1. Restricted gTLDs

TLD	More information
.gov, *.fed*	*https://www.dotgov.gov*
.edu	*https://net.educause.edu/*
.mil	Military personnel and contractors should contact their IT department, or the Department of Defense Unified Registration System

The Internet Corporation for Assigned Names and Numbers (ICANN) is ultimately responsible for management of TLDs, though it delegates much of the actual administration to other organizations. Recently, the ICANN has authorized many new gTLDs, such as *.agency*, *.florist*, *.recipes*, and even *.ninja*. For the foreseeable future, *.com* will probably remain the "premium" TLD, and the hardest one to get real estate in. People who were lucky (or shrewd) enough to purchase *.com* domains in the internet's formative years received massive payouts for prime domains (for example, Facebook purchased *fb.com* in 2010 for a whopping $8.5 million dollars).

Given the scarcity of *.com* domains, people are turning to alternative TLDs, or using *.com.us* to try to get a domain that accurately reflects their organization. When picking a domain, you should consider how it's going to be used. If you plan on marketing primarily electronically (where people are more likely to click a link than type in a domain), then you should probably focus more on getting a catchy or meaningful domain than a short one. If you're focusing on print advertising, or you have reason to believe people will be entering your URL manually into their devices, you might consider alternative TLDs so you can get a shorter domain name. It's also common practice to have two domains: a short, easy-to-type one, and a longer one more suitable for marketing.

Subdomains

Whereas a TLD goes after your domain, a subdomain goes before it. By far, the most common subdomain is *www*. I've never particularly cared for this subdomain. After all, you're at a computer, *using* the World Wide Web; I'm pretty sure you're not going to be confused if there isn't a *www* to remind you of what you're doing. For this reason, I recommend using no subdomain for your primary domain: *http://meadowlarktravel.com/* instead of *http://www.meadowlarktravel.com/*. It's shorter and less busy, and thanks to redirects, there's no danger of losing visits from people who automatically start everything with *www*.

Subdomains are used for other purposes too. I commonly see things like *blogs.meadowlarktravel.com*, *api.meadowlarktravel.com*, and *m.meadowlarktravel.com* (for a mobile site). Often this is done for technical reasons: it can be easier to use a subdomain if, for example, your blog uses a completely different server than the rest of your site. A good proxy, though, can redirect traffic appropriately based on either subdomain or path, so the choice of whether to use a subdomain or a path should be more content-focused than technology-focused (remember what Tim Berners-Lee said about URLs expressing your information architecture, not your technical architecture).

I recommend that subdomains be used to compartmentalize significantly different parts of your website or service. For example, I think it's a good use of subdomains to make your API available at *api.meadowlarktravel.com*. Microsites (sites that have a different appearance than the rest of your site, usually highlighting a single product or subject) are also good candidates for subdomains. Another sensible use for subdomains is to separate admin interfaces from public interfaces (*admin.meadowlarktravel.com*, for employees only).

Your domain registrar, unless you specify otherwise, will redirect all traffic to your server regardless of subdomain. It is up to your server (or proxy), then, to take appropriate action based on the subdomain.

Nameservers

The "glue" that makes domains work are nameservers, and this is what you'll be asked to provide when you establish hosting for your website. Usually, this is pretty straightforward, as your hosting service will do most of the work for you. For example, let's say we choose to host *meadowlarktravel.com* at DigitalOcean. When you set up your hosting account with DigitalOcean, you'll be given the names of the DigitalOcean nameservers (there are multiple ones for redundancy). DigitalOcean, like most hosting providers, calls their nameservers *ns1.digitalocean.com*, *ns1.digitalocean.com*, and so on. Go to your domain registrar and set the nameservers for the domain you want to host, and you're all set.

The way the mapping works in this case is as follows:

1. Website visitor navigates to *http://meadowlarktravel.com/*.
2. The browser sends the request to the computer's network system.
3. The computer's network system, which has been given an internet IP address and a DNS server by the internet provider, asks the DNS resolver to resolve *meadowlarktravel.com*.

4. The DNS resolver is aware that *meadowlarktravel.com* is handled by *ns1.digitalocean.com*, so it asks *ns1.digitalocean.com* to give it an IP address for *meadowlarktravel.com*.
5. The server at *ns1.digitalocean.com* receives the request and recognizes that *meadowlarktravel.com* is indeed an active account, and returns the associated IP address.

While this is the most common case, it's not the only way to configure your domain mapping. Since the server (or proxy) that actually serves your website has an IP address, we can cut out the middleman by registering that IP address with the DNS resolvers (this effectively cuts out the middleman of the nameserver *ns1.digitalocean.com* in the previous example). For this approach to work, your hosting service must assign you a *static* IP address. Commonly, hosting providers will give your server(s) a *dynamic* IP address, which means it may change without notice, which would render this scheme ineffective. It can sometimes cost extra to get a static IP address instead of a dynamic one: check with your hosting provider.

If you want to map your domain to your website directly (skipping your host's nameservers), you will either be adding an A record or a CNAME record. An *A record* maps a domain name directly to an IP address, whereas a *CNAME* maps one domain name to another. CNAME records are usually a little less flexible, so A records are generally preferred.

If you're using AWS for your nameservers, in addition to A and CNAME records, it also has a record called an *alias* that offers a lot of advantages if you're pointing it to a service hosted on AWS. For more information, see the AWS documentation.

Whatever technique you use, domain mapping is usually aggressively cached, meaning that when you change your domain records, it can take up to 48 hours for your domain to be attached to the new server. Keep in mind that this is also subject to geography: if you see your domain working in Los Angeles, your client in New York may see the domain attached to the previous server. In my experience, 24 hours is usually sufficient for domains to resolve correctly in the continental US, with international resolution taking up to 48 hours.

If you need something to go live precisely at a certain time, you should not rely on DNS changes. Rather, modify your server to redirect to the "coming soon" site or page, and make the DNS changes in advance of the actual switchover. At the appointed moment, then, you can have your server switch over to the live site, and your visitors will see the change immediately, regardless of where they are in the world.

Hosting

Choosing a hosting service can seem overwhelming at first. Node has taken off in a big way, and everyone's clamoring to offer Node hosting to meet the demand. How you select a hosting provider depends very much on your needs. If you have reason to believe your site will be the next Amazon or Twitter, you'll have a very different set of concerns than you would if you were building a website for your local stamp collector's club.

Traditional hosting or cloud hosting?

The term "cloud" is one of the most nebulous tech terms to crop up in recent years. Really, it's just a fancy way to say "the internet," or "part of the internet." The term is not entirely useless, though. While not part of the technical definition of the term, hosting in the cloud usually implies a certain commoditizing of computing resources. That is to say, we no longer think about a "server" as a distinct, physical entity: it's simply a homogeneous resource somewhere in the cloud, and one is as good as another. I'm oversimplifying, of course: computing resources are distinguished (and priced) according to their memory, number of CPUs, etc. The difference is between knowing (and caring) what actual server your app is hosted on, and knowing it's hosted on *some* server in the cloud, and it could just as easily be moved over to a different one without you knowing (or caring).

Cloud hosting is also highly *virtualized*. That is, the server(s) your app is running on are not usually physical machines, but virtual machines running on physical servers. This idea was not introduced by cloud hosting, but it has become synonymous with it.

While cloud hosting had humble origins, it means a lot more than "homogenous servers" now. The major cloud providers offer many infrastructure services that (in theory) reduce your maintenance burden and offer a high degree of scalability. These services include database storage, file storage, networking queues, authentication, video processing, telecommunications services, artificial intelligence engines, and much more.

Cloud hosting can be a little disconcerting at first, not knowing anything about the actual physical machine your server is running on, trusting that your servers aren't going to be affected by the other servers running on the same computer. Really, though, nothing has changed: when your hosting bill comes, you're still paying for essentially the same thing: someone taking care of the physical hardware and networking that enables your web applications. All that's changed is that you're more removed from the hardware.

I believe that “traditional” hosting (for lack of a better term) will eventually disappear altogether. That’s not to say hosting companies will go out of business (though some inevitably will); they will just start to offer cloud hosting themselves.

XaaS

When considering cloud hosting, you will come across the acronyms SaaS, PaaS, IaaS, and FaaS:

Software as a Service (SaaS)
SaaS generally describes software (websites, apps) that are provided to you: you just use them. An example would be Google Documents or Dropbox.

Platform as a Service (PaaS)
PaaS provides all of the infrastructure for you (operating systems, networking—all of that is handled). All you have to do is write your applications. While there is often a blurry line between PaaS and IaaS (and you will often find yourself straddling that line as a developer), this is generally the service model we’re discussing in this book. If you’re running a website or web service, PaaS is probably what you’re looking for.

Infrastructure as a Service (IaaS)
IaaS gives you the most flexibility, but at cost. All you get are virtual machines and a basic network connecting them. You are then responsible for installing and maintaining operating systems, databases, and network policies. Unless you need this level of control over your environment, you will generally want to stick with PaaS. (Note that PaaS does allow you to have control over the *choice* of operating systems and network configuration: you just don’t have to do it yourself.)

Functions as a Service (FaaS)
FaaS describes offerings such as AWS Lambda, Google Functions, and Azure Functions, which provide a way to run individual functions in the cloud without having to configure the runtime environment yourself. It’s at the core of what is commonly being called “serverless” architecture.

The behemoths

The companies that essentially run the internet (or, at least, are heavily invested in the running of the internet) have realized that with the commoditization of computing resources, they have another viable product to sell. Amazon, Microsoft, and Google all offer cloud computing services, and their services are quite good.

All of these services are priced similarly: if your hosting needs are modest, there will be minimal price difference among the three. If you have very high bandwidth or storage needs, you will have to evaluate the services more carefully, as the cost difference could be greater, depending on your needs.

While Microsoft does not normally leap to mind when we consider open source platforms, I would not overlook Azure. Not only is the platform established and robust, but Microsoft has bent over backward to make it friendly to not just Node, but the open source community. Microsoft offers a one-month Azure trial, which is a great way to determine if the service meets your needs; if you're considering one of the big three, I definitely recommend the free trial to evaluate Azure. Microsoft offers Node APIs for all of its major services, including its cloud storage service. In addition to excellent Node hosting, Azure offers an excellent cloud storage system (with a JavaScript API), as well as good support for MongoDB.

Amazon offers the most comprehensive set of resources, including SMS (text message), cloud storage, email services, payment services (ecommerce), DNS, and more. In addition, Amazon offers a free usage tier, making it very easy to evaluate.

Google's cloud platform has come a long way and now offers robust Node hosting and, as you might expect, excellent integration with its own services (mapping, authentication, and search being particularly attractive).

In addition to the "big three," it is worth considering Heroku, which has been catering to people wanting to host fast and nimble Node applications for some time now. I've also had great luck with DigitalOcean, which focuses more on providing containers and a limited number of services in a very user-friendly manner.

Boutique hosting

Smaller hosting services, which I'm going to call "boutique" hosting services (for lack of a better word), may not have the infrastructure or resources of Microsoft, Amazon, or Google, but that doesn't mean they don't offer something valuable.

Because boutique hosting services can't compete in terms of infrastructure, they usually focus on customer service and support. If you need a lot of support, you might want to consider a boutique hosting service. If you have a hosting provider you've been happy with, don't hesitate to ask if it offers (or plans on offering) Node hosting.

Deployment

It still surprises me that, in 2019, people are still using FTP to deploy their applications. If you are, *please stop*. FTP is in no way secure. Not only are all your files transmitted unencrypted, but your *username and password* are also. If your hosting provider doesn't give you an option, find a new hosting provider. If you really have no choice, make sure you use a unique password that you're not using for anything else.

At minimum, you should be using SFTP or FTPS (not to be confused), but you should really be considering a *continuous delivery* (CD) service.

The idea behind CD is that you're never very far away from a version that can be released (weeks or even days). CD is usually used in the same breath as *continuous integration* (CI), which refers to automated processes for integrating the work of developers and testing them.

In general, the more you can automate your processes, the easier your development will be. Imagine merging in changes, and automatically getting notified that unit tests pass, then integration tests pass, and then seeing your changes online…in a matter of minutes! It's a great goal, but you have to invest some work up front to get it set up, and there will be some maintenance over time.

Although the steps themselves are similar (run unit tests, run integration tests, deploy to staging servers, deploy to production servers), the process of setting up CI/CD pipelines (a word you'll hear a lot when discussing CI/CD) varies substantially.

You should look at some of the options available for CI/CD and choose one that meets your needs:

AWS CodePipeline
: If you're hosting on AWS, CodePipeline should be first on your list, as it will be the easiest path to CI/CD for you. It's very robust, but I've found it to be a little less user-friendly than some of the other options.

Microsoft Azure Web Apps
: If you're hosting on Azure, Web Apps is your best bet (are you noticing a trend here?). I haven't had much experience with this service, but it seems to be well loved in the community.

Travis CI
: Travis CI has been around for a long time now, and has a large, loyal user base and good documentation.

Semaphore
: Semaphore is easy to set up and configure, but it doesn't offer many features, and its basic (low-cost) plans are slow.

Google Cloud Build
: I haven't tried Google Cloud Build yet, but it looks robust and, like CodePipeline and Azure Web Apps, it's likely that is the best choice if you're hosting on Google Cloud.]

CircleCI
: CircleCI is another CI that's been around for some time, and is well loved.

Jenkins

Jenkins is another incumbent with a large community. My experience is that it hasn't kept up with modern deployment practices as well as some of the other options here, but it did just release a new version that looks promising.

At the end of the day, CI/CD services are automating the activities that *you* create. You still have to write the code, determine your versioning scheme, write high-quality unit and integration tests and a way to run them, and understand your deployment infrastructure. The examples in this book could be automated simply enough: most everything could be deployed to a single server running a Node instance. However, as you start to grow your infrastructure, so too will your CI/CD pipeline grow in complexity.

Git's role in deployment

Git's greatest strength (and greatest weakness) is its flexibility. It can be adapted to almost any workflow imaginable. For the sake of deployment, I recommend creating one or more branches *specifically for deployment*. For example, you might have a `production` branch and a `staging` branch. How you use those branches is very much up to your individual workflow.

One popular approach is to flow from `master` to `staging` to `production`. So once some changes on `master` are ready to go live, you could merge them into `staging`. Once they have been approved on the staging server, you could then merge `staging` into `production`. While this makes logical sense, I dislike the clutter it creates (merges, merges everywhere). Also, if you have lots of features that need to be staged and pushed to production in different orders, this can get messy quickly.

I feel a better approach is to merge `master` into `staging` and, when you're ready to go live with changes, then merge `master` into `production`. In this way, `staging` and `production` become less associated: you can even have multiple staging branches to experiment with different features before going live (and you can merge things other than `master` into them). Only when something has been approved for production do you merge it into `production`.

What happens when you need to roll back changes? This is where things can get complicated. There are multiple techniques for undoing changes, such as applying the inverse of a commit to undo prior commits (`git revert`), these techniques not only are complicated, but also can can cause problems down the line. The typical way of handling this is to create tags (for example, `git tag v1.2.0` on your `production` branch) every time you make a deployment. If you need to roll back to a specific version, you always have that tag available.

In the end, it is up to you and your team to decide on a Git workflow. More important than the workflow you pick is the consistency with which you use it, and the training and communication surrounding it.

> We've already discussed the value of keeping your binary assets (multimedia and documents) separate from your code repository. Git-based deployment offers another incentive for this approach. If you have 4 GB of multimedia data in your repository, they're going to take forever to clone, and you have an unnecessary copy of all of your data for every production server.

Manual Git-based deployment

If you're not ready yet to take the step of setting up CI/CD, you could start with a manual Git-based deployment. The advantage of this approach is that you'll get comfortable with the steps and challenges involved in deployment, which will serve you well when you take the step of automation.

For each server you want to deploy to, you will have to clone the repository, check out the `production` branch, and then set up the infrastructure necessary to start/restart your app (which will be dependent on your choice of platform). When you update the `production` branch, you will have to go to each server, run `git pull --ff-only`, run `npm install --production`, and then restart the app. If your deployments aren't often, and you don't have very many servers, this may not represent a terrible hardship, but if you're updating more often, this will get old fast, and you'll want to find some way to automate the system.

> The `--ff-only` argument to `git pull` allows only fast-forward pulls, preventing automatic merging or rebasing. If you know the pull is fast-forward only, you may safely omit it, but if you get in the habit of doing it, you will never accidentally invoke a merge or rebase!

In essence, what you are doing here is replicating the way you work in development, except you're doing it on a remote server. Manual processes always run the risk of human error, and I recommend this approach only as a stepping stone toward more automated development.

Conclusion

Deploying your website (especially for the first time) should be an exciting occasion. There should be champagne and cheering, but all too often, there is sweating, cursing, and late nights. I've seen far too many websites launched at three in the morning by an irritable, exhausted team. Fortunately, that's changing, partly thanks to cloud deployment.

No matter what deployment strategy you choose, the most important thing you can do is to start production deployments early, before the site is ready to go live. You don't have to hook up the domain, so the public doesn't need to know. If you've already deployed the site to production servers half a dozen times before the day of launch, your chances of a successful launch will be much higher. Ideally, your functioning website will already be running on the production server long before launch: all you have to do is flip the switch from the old site to the new site.

CHAPTER 22

Maintenance

You launched the site! Congratulations, now you never have to think about it again. What's that? You *do* have to keep thinking about it? Well, in that case, keep reading.

Although it has happened a couple of times in my career, it has been the exception to the rule that you finish a site and then never have to touch it again (and when it does happen, it's usually because someone else is doing the work, not that work doesn't need to be done). I distinctly remember one website launch "postmortem." I piped up and said, "Shouldn't we really call it a *postpartum*?"[1] Launching a website really is more of a birth than a death. Once it launches, you're glued to the analytics, anxiously awaiting the client's reaction, waking up at three in the morning to check to see if the site is still up. It's your baby.

Scoping a website, designing a website, building a website: these are all activities that can be planned to death. But what usually receives short shrift is *planning the maintenance* of a website. This chapter will give you some advice on navigating those waters.

The Principles of Maintenance

Have a Longevity Plan

It always surprises me when a client agrees on a price to build a website, but it's never discussed how long the site is expected to last. My experience is that if you do good work, clients are happy to pay for it. What clients do *not* appreciate is the unexpected: being told after three years that their site has to be rebuilt when they had an unspoken expectation that it would last five.

[1] As it happened, the term *postpartum* was a little too visceral. We now call them *retrospectives*.

The internet moves fast. If you built a website with the absolute best and newest technology you could find, it might feel like a creaky relic in two short years. Or it could truck along for seven, aging, but doing so gracefully (this is a lot less common!).

Setting expectations about website longevity is part art, part salesmanship, and part science. The science of it involves something that all scientists, but very few web developers, do: keep records. Imagine if you had a record of every website your team had ever launched, the history of maintenance requests and failures, the technologies used, and how long before each site was rebuilt. There are many variables, obviously, from the team members involved, to the economy, to the shifting winds of technology, but that doesn't mean that meaningful trends can't be discovered in the data. You may find that certain development approaches work better for your team, or certain platforms or technologies. What I almost guarantee you will find is a correlation between "procrastination" and defects: the longer you put off an infrastructure update or platform upgrade that's causing pain, the worse it will be. Having a good issue-tracking system and keeping meticulous records will allow you to give your client a much better (and more realistic) picture of what the life cycle of their project is going to be.

The salesmanship of it boils down to money, of course. If a client can afford to have their website completely rebuilt every three years, then they won't be very likely to suffer from aging infrastructure (they will have other problems, though). On the flip side, there will be clients who need their dollar to stretch as far as possible, wanting a website that will last for five or even seven years. (I've known websites that have dragged on for even longer than that, but I feel that seven years is the maximum realistic life expectancy for websites that have any hope of continuing to be useful.) You have a responsibility to both of these clients, and both come with their own challenges. With the clients who have a lot of money, don't just take their money because they have it: use that extra money to give them something extraordinary. With the clients on a tight budget, you will have to find creative ways to design their website for greater longevity in the face of constantly changing technology. Both of these extremes have their own challenges, but ones that can be solved. What's important, though is that you *know* what the expectations are.

Lastly, there's the art of the matter. This is what ties it all together: understanding what the client can afford, and where you can honestly convince the client to spend more money so they get value where they need it. It is also the art of understanding technology futures, and being able to predict what technologies will be painfully obsolete in five years and which will be going strong.

There's no way to predict anything with absolute certainty, of course. You could bet wrong on technologies, personnel shifts can completely change the technical culture of your organization, and technology vendors can go out of business (though this is usually less of a problem in the open source world). The technology that you thought would be solid for the lifetime of your product may turn out to be a fad, and you'll find yourself facing the decision to rebuild sooner than you expected. On the flip side, sometimes the exactly right team comes together at the exact right time with the exact right technology, and something is created that far outlives any reasonable expectations. None of this uncertainty should deter you from having a plan, however: better to have a plan that goes awry than to always be rudderless.

It should be clear to you by now that I feel that JavaScript and Node are technologies that are going to be around for a while. The Node community is vibrant and enthusiastic, and wisely based on a language that has clearly *won*. Most important, perhaps, is that JavaScript is a multiparadigm language: object-oriented, functional, procedural, synchronous, asynchronous—it's all there. This makes JavaScript an inviting platform for developers from many different backgrounds, and is in large part responsible for the pace of innovation in the JavaScript ecosystem.

Use Source Control

This probably seems obvious to you, but it's not just about *using* source control, it's about using it *well*. Why are you using source control? Understand the reasons, and make sure the tools are supporting those reasons. There are many reasons to use source control, but the one that always seems to me to have the biggest payoff is attribution: knowing exactly what change was made when and who did it, so I can ask for more information if necessary. Version control is one of our greatest tools for understanding the history of our projects and how we work together as a team.

Use an Issue Tracker

Issue trackers go back to the science of development. Without a systematic way to record the history of a project, no insight is possible. You've probably heard it said that the definition of insanity is "doing the same thing over and over again and expecting different results" (often dubiously attributed to Albert Einstein). It does seem crazy to repeat your mistakes over and over again, but how can you avoid it if you don't know what mistakes you're making?

Record everything: every defect the client reports; every defect you find before the client sees it; every complaint, every question, every bit of praise. Record how long it took, who fixed it, what Git commits were involved, and who approved the fix. The art here is finding tools that don't make this overly time-consuming or onerous. A bad issue-tracking system will languish, unused, and it will be worse than useless. A good issue-tracking system will yield vital insights into your business, your team, and your clients.

Exercise Good Hygiene

I'm not talking about brushing your teeth—though you should do that too—I'm talking about version control, testing, code reviews, and issue tracking. The tools you use are useful only if you use them, and use them correctly. Code reviews are a great way to encourage hygiene because *everything* can be touched on, from discussing the use of the issue-tracking system in which the request originated to the tests that had to be added to verify the fix to the version control commit comments.

The data you collect from your issue-tracking system should be reviewed on a periodic basis and discussed with the team. From this data, you can gain insights about what's working and what's not. You might be surprised by what you find.

Don't Procrastinate

Institutional procrastination can be one of the hardest things to combat. Usually it's something that doesn't seem so bad: you notice that your team is routinely eating up a lot of hours on a weekly update that could be drastically improved by a little refactoring. Every week that you delay refactoring is another week you're paying the inefficiency cost.[2] Worse, some costs may increase over time.

A great example of this is failing to update software dependencies. As the software ages, and team members change, it's harder to find people who remember (or ever understood) the creaky old software. The support community starts to evaporate, and before long, the technology is deprecated and you can't get any kind of support for it. You often hear this described as *technical debt*, and it's a very real thing. While you should avoid procrastinating, understanding the website longevity can factor into these decisions: if you're just about to redesign the whole website, there's little value in eliminating technical debt that's been building up.

Do Routine QA Checks

For each of your websites, you should have a *documented* routine QA check. That check should include a link checker, HTML and CSS validation, and running your tests. The key here is *documented*: if the items that compose the QA check aren't documented, you will inevitably miss things. A documented QA checklist for each site not only helps prevent overlooked checks, but also allows new team members to be effective immediately. Ideally, the QA checklist can be executed by a nontechnical team member. This will give your (possibly) nontechnical manager confidence in your team and will allow you to spread QA responsibilities around if you don't have a dedicated QA department. Depending on your relationship with your client, you may

[2] Mike Wilson of Fuel has this rule of thumb: "The third time you do something, take the time to automate it."

also want to share your QA checklist (or part of it) with the client; it's a good way to remind them what they're paying for, and that you are looking out for their best interests.

As part of your routine QA check, I recommend using Google Webmaster Tools and Bing Webmaster Tools. They are easy to set up, and they give you a very important view of your site: how the major search engines see it. It will alert you to any problems with your *robots.txt* file, HTML issues that are interfering with good search results, security issues, and more.

Monitor Analytics

If you're not running analytics on your website, you need to start now: it provides vital insight into not just the popularity of your website, but also how your users are using it. Google Analytics (GA) is excellent (and free!), and even if you supplement it with additional analytics services, there's little reason not to include GA on your site.

Often, you will be able to spot subtle UX issues by keeping an eye on your analytics. Are there certain pages that are not getting the traffic that you expect? That could indicate a problem with your navigation or promotions, or an SEO issue. Are your bounce rates high? That could indicate the content on your pages needs some tailoring (people are getting to your site by searching, but when they arrive on your site, they realize it's not what they're looking for). You should have an analytics checklist to go along with your QA checklist (it could even be part of your QA checklist). That checklist should be a "living document," because over the lifetime of your website, you or your client may have shifting priorities about what content is most important.

Optimize Performance

Study after study has shown the dramatic effect of performance on website traffic. It's a fast-paced world, and people expect their content delivered quickly, especially on mobile platforms. The number one principle in performance tuning is to *profile first, then optimize*. "Profiling" means finding out what actually is slowing your site down. If you spend days speeding up your content rendering when the problem is your social media plugins, you're wasting precious time and money.

Google PageSpeed Insights is a great way to measure the performance of your website (and now PageSpeed data is recorded in Google Analytics so you can monitor performance trends). I will not only give you an overall score for mobile and desktop performance, but also make prioritized suggestions about how to improve performance.

Unless you currently have performance issues, it's probably not necessary to do periodic performance checks (monitoring Google Analytics for significant changes in performance scores should be sufficient). However, it is gratifying to watch your boost in traffic when you improve performance.

Prioritize Lead Tracking

In the internet world, the strongest signal your visitors can give you to indicate interest in your product or service is contact information. You should treat this information with the utmost care. Any form that collects an email or phone number should be tested routinely as part of your QA checklist, and there should *always* be redundancy when you collect that information. The worst thing you can do to a potential customer is collect contact information and then lose it.

Because lead tracking is so critical to the success of your website, I recommend these five principles for collecting information:

Have a fallback in case JavaScript fails
: Collecting customer information via Ajax is fine—it often results in a better user experience. However, if JavaScript should fail for any reason (the user could disable it, or a script on your website could have an error, preventing your Ajax from functioning correctly), the form submission should work anyway. A great way to test this is to disable JavaScript and use your form. It's okay if the user experience is not ideal: the point is that user data is not lost. To implement this, *always* have a valid and working `action` parameter in your `<form>` tag, even if you normally use Ajax.

If you use Ajax, get the URL from the form's `action` *parameter*
: While not strictly necessary, this helps prevent you from accidentally forgetting the `action` parameter on your `<form>` tags. If you tie your Ajax to successful no-JavaScript submission, it's much harder to lose customer data. For example, your form tag could be `<form action="/submit/email" method="POST">`; then in your Ajax code, you would get the `action` for the form from the DOM, and use that in your Ajax submission code.

Provide at least one level of redundancy
: You'll probably want to save leads to a database or an external service such as Campaign Monitor. But what if your database fails, or Campaign Monitor goes down, or there's a network issue? You still don't want to lose that lead. A common way to provide redundancy is to send an email in addition to storing the lead. If you take this approach, you should not use a person's email address, but a shared email address (such as *dev@meadowlarktravel.com*): the redundancy does no good if you send it to a person and that person leaves the organization. You could also store the lead in a backup database, or even a CSV file. However, *whenever* your primary storage fails, there should be some mechanism to alert you of the failure. Collecting a redundant backup is the first half of the battle; being aware of failures and taking appropriate action is the second half.

In case of total storage failure, inform the user

Let's say you have three levels of redundancy: your primary storage is Campaign Monitor, and if that fails, you back up to a CSV file and send an email to *dev@meadowlarktravel.com*. If *all* of these channels fail, the user should receive a message that says something like, "We're sorry, we're experiencing technical difficulties. Please try again later, or contact *support@meadowlarktravel.com*."

Check for positive confirmation, not absence of an error

It's quite common to have your Ajax handler return an object with an `err` property in the case of failure; the client code then has something that looks like this: `if(data.err){ /* inform user of failure */ } else { /* thank user for successful submission */ }`. Avoid this approach. There's nothing wrong with setting an `err` property, but if there's an error in your Ajax handler, leading the server to respond with a 500 response code or a response that isn't valid JSON, *this approach could fail silently*. The user's lead will disappear into the void, and they will be none the wiser. Instead, provide a `success` property for successful submission (even if the primary storage failed: if the user's information was recorded by *something*, you may return `success`). Then your client-side code becomes `if(data.success){ /* thank user for successful submission */ } else { /* inform user of failure \*/ }`.

Prevent "Invisible" Failures

I see it all the time: because developers are in a hurry, they record errors in ways that never get checked. Whether it is a logfile, a table in a database, a client-side console log, or an email that goes to a dead address, the end result is the same: *your website has quality problems that are going unnoticed.*

The number one defense you can have against this problem is to *provide an easy, standard method for logging errors*. Document it. Don't make it difficult. Don't make it obscure. Make sure every developer that touches your project is aware of it. It can be as simple as exposing a `meadowlarkLog` function (`log` is often used by other packages). It doesn't matter if the function is recording to a database, flat file, email, or some combination thereof: the important thing is that it is standard. It also allows you to improve your logging mechanism (for example, flat files are less useful when you scale out your server, so you would modify your `meadowlarkLog` function to record to a database instead). Once you have the logging mechanism in place, documented, and everyone on your team knows about it, add "check logs" to your QA checklist, and have instructions on how to do that.

Code Reuse and Refactoring

One tragedy I see all the time is the reinvention of the wheel, over and over and over again. Usually it's just small things: tidbits that feel easier to just rewrite than to dig up in some project that you did months ago. All of those little rewritten snippets add up. Worse, it flies in the face of good QA: you're probably not going to go to the trouble to write tests for all these little snippets (and if you do, you're doubling the time that you're wasting by not reusing existing code). Each snippet—doing the same thing—can have different bugs. It's a bad habit.

Development in Node and Express offers some great ways to combat this problem. Node brought namespacing (via modules) and packages (via npm), and Express brings the concept of middleware. With these tools at your disposal, developing reusable code is a lot easier.

Private npm Registry

npm registries are a great place to store shared code; it's what npm was designed for, after all. In addition to simple storage, you get versioning, and a convenient way to include those packages in other projects.

There's a fly in the ointment, though: unless you're working in a completely open source organization, you may not want to create npm packages for all of your reusable code. (There can be other reasons than intellectual property protection, too: your packages could be so organization- or project-specific that it doesn't make sense to make them available on a public registry.)

One way to handle this is *private npm registries*. npm now offers Orgs, which allows you to publish private packages and give your developers paid logins, allowing them to access those private packages. See npm for more information about npm Orgs and private packages.

Middleware

As we've seen throughout this book, writing middleware is not some big, scary, complicated thing: we've done it a dozen times in this book and, after a while, you will do it without even thinking about it. The next step, then, is to put reusable middleware in a package and put it in an npm registry.

If you find that your middleware is too project-specific to put in a reusable package, you should consider refactoring the middleware to be configured for more general use. Remember that you can pass configuration objects into middleware to make them useful in a whole range of situations. Here is an overview of the most common ways to expose middleware in a Node module. All of the following assume that you're using these modules as a package, and that package is called `meadowlark-stuff`.

Module exposes middleware function directly

Use this method if your middleware doesn't need a configuration object:

```
module.exports = (req, res, next) => {
  // your middleware goes here...remember to call next()
  // or next('route') unless this middleware is expected
  // to be an endpoint
  next()
}
```

To use this middleware:

```
const stuff = require('meadowlark-stuff')

app.use(stuff)
```

Module exposes a function that returns middleware

Use this method if your middleware needs a configuration object or other information:

```
module.exports = config => {
  // it's common to create the config object
  // if it wasn't passed in:
  if(!config) config = {}

        return (req, res, next) => {
    // your middleware goes here...remember to call next()
    // or next('route') unless this middleware is expected
    // to be an endpoint
    next()
        }
}
```

To use this middleware:

```
const stuff = require('meadowlark-stuff')({ option: 'my choice' })

app.use(stuff)
```

Module exposes an object that contains middleware

Use this option if you want to expose multiple related middleware:

```
module.exports = config => {
  // it's common to create the config object
  // if it wasn't passed in:
  if(!config) config = {}

        return {
    m1: (req, res, next) => {
      // your middleware goes here...remember to call next()
      // or next('route') unless this middleware is expected
```

```
          // to be an endpoint
          next()
                    },
        m2: (req, res, next) => {
          next()
        },
      }
    }
```

To use this middleware:

```
    const stuff = require('meadowlark-stuff')({ option: 'my choice' })

    app.use(stuff.m1)
    app.use(stuff.m2)
```

Conclusion

When you're building a website, the focus is often on the launch, and for good reason: a lot of excitement surrounds a launch. However, a client that is delighted by a newly launched website will quickly become a dissatisfied customer if care isn't taken in maintaining the website. Approaching your maintenance plan with the same care with which you launch websites will provide the kind of experience that keeps clients coming back.

CHAPTER 23

Additional Resources

In this book, I have given you a comprehensive overview of building websites with Express. And we have covered a remarkable amount of ground, but we've still only scratched the surface of the packages, techniques, and frameworks that are available to you. In this chapter, we'll discuss where you can go for additional resources.

Online Documentation

For JavaScript, CSS, and HTML documentation, the Mozilla Developer Network (MDN) is without equal. If I need JavaScript documentation, I either search directly on MDN or append "mdn" to my search query. Otherwise, inevitably, w3schools appears in the search. Whoever is managing SEO for w3schools is a genius, but I recommend avoiding this site: I find the documentation is often severely lacking.

Where MDN is a great HTML reference, if you're new to HTML5 (or even if you're not), you should read Mark Pilgrim's *Dive Into HTML5*. WHATWG maintains an excellent "living standard" HTML5 specification; it is usually where I turn to first for really hard-to-answer HTML questions. Finally, the official specifications for HTML and CSS are located on the W3C website; they are dry, difficult-to-read documents, but sometimes it's your only recourse for the very hardest problems.

JavaScript adheres to the ECMA-262 ECMAScript language specification. To track the availability of JavaScript features in Node (and various browsers), see the excellent guide maintained by @kangax.

The Node documentation is very good, and comprehensive, and it should be your first choice for authoritative documentation about Node modules (such as `http`, `https`, and `fs`). The Express documentation is quite good, but not as comprehensive as one might like. The npm documentation is comprehensive and useful.

Periodicals

There are three free periodicals you should absolutely subscribe to and read dutifully every week:

- JavaScript Weekly
- Node Weekly
- HTML5 Weekly

These three periodicals will keep you informed of the latest news, services, blogs, and tutorials as they become available.

Stack Overflow

Chances are good that you've already used Stack Overflow (SO): since its inception in 2008, it has become the dominant online Q&A site, and is your best resource to get your JavaScript, Node, and Express questions answered (and any other technology covered in this book). Stack Overflow is a community-maintained, reputation-based Q&A site. The reputation model is what's responsible for the quality of the site and its continued success. Users can gain reputation by having their questions or answers "upvoted" or having an accepted answer. You don't have to have any reputation to ask a question, and registration is free. However, there are things you can do to increase the chances of getting your question answered in a useful manner, which we'll discuss in this section.

Reputation is the currency of Stack Overflow, and while there are people out there who genuinely want to help you, it's the chance to gain reputation that's the icing on the cake that motivates good answers. There are a lot of really smart people on SO, and they're all competing to provide the first and/or best correct answer to your question (there's a strong disincentive to provide a quick but bad answer, thankfully). Here are things you can do to increase the chances of getting a good answer for your question:

Be an informed SO user
: Take the SO tour, and then read "How do I ask a good question?" If you're so inclined, you can go on to read all of the help documentation—you'll earn a badge if you read it all!

Don't ask questions that have already been answered
: Do your due diligence, and try to find out if someone has already asked your question. If you ask a question that has an easily found answer already on SO, your question will quickly be closed as a duplicate, and people will often downvote you for this, negatively affecting your reputation.

Don't ask people to write your code for you

You will quickly find your question downvoted and closed if you simply ask, "How do I do *X*?" The SO community expects you to make an effort to solve your own problem before resorting to SO. Describe in your question what you've tried and why it isn't working.

Ask one question at a time

Questions that are asking five things—"How do I do this, then that, then the other things, and what's the best way to do this?"—are difficult to answer and are discouraged.

Craft a minimal example of your issue

I answer a lot of SO questions, and the ones I almost automatically skip over are those where I see three pages of code (or more!). Just taking your 5,000-line file and pasting into an SO question is not a great way to get your question answered (but people do it all the time). It's a lazy approach that isn't often rewarded. Not only are you less likely to get a useful answer, but the very process of eliminating things that *aren't* causing the problem can lead you to solving the problem yourself (then you don't even need to ask a question on SO). Crafting a minimal example is good for your debugging skills and for your critical thinking ability, and makes you a good SO citizen.

Learn Markdown

Stack Overflow uses Markdown for formatting questions and answers. A well-formatted question has a better chance of being answered, so you should invest the time to learn this useful and increasingly ubiquitous markup language.

Accept and upvote answers

If someone answers your question satisfactorily, you should upvote and accept it; it boosts the reputation of the answerer, and reputation is what drives SO. If multiple people provide acceptable answers, you should pick the one you think is best and accept that, and upvote anyone else you feel offered a useful answer.

If you figure out your own problem before someone else does, answer your own question

SO is a community resource: if you have a problem, chances are, someone else has it too. If you've figured it out, go ahead and answer your own question for the benefit of others.

If you enjoy helping the community, consider answering questions yourself: it's fun and rewarding, and it can lead to benefits that are more tangible than an arbitrary reputation score. If you have a question for which you've received no useful answers for two days, you can start a *bounty* on the question, using your own reputation. The reputation is withdrawn from your account immediately, and it is nonrefundable. If someone answers the question to your satisfaction, and you accept their answer, they will receive the bounty. The catch is, of course, you have to have reputation to start a bounty: the minimum bounty is 50 reputation points. While you can get reputation from asking quality questions, it's usually quicker to get reputation by providing quality answers.

Answering people's questions also has the benefit of being a great way to learn. I generally feel that I learn more from answering other people's questions than I do from having my questions answered. If you want to really thoroughly learn a technology, learn the basics and then start trying to tackle people's questions on SO. At first you might be consistently beat out by people who are already experts, but before long, you'll find that you *are* one of the experts.

Lastly, you shouldn't hesitate to use your reputation to further your career. A good reputation is absolutely worth putting on a résumé. It's worked for me and, now that I'm in the position of interviewing developers myself, I'm always impressed to see a good SO reputation (I consider a "good" SO reputation anything over 3,000; five-digit reputations are *great*). A good SO reputation tells me that someone is not just competent in their field, but they are clear communicators and generally helpful.

Contributing to Express

Express and Connect are open source projects, so anyone can submit *pull requests* (GitHub lingo for changes you've made that you would like included in the project). This is not easy to do: the developers working on these projects are pros and the ultimate authority on their own projects. I'm not discouraging you from contributing, but I am saying you have to dedicate some significant effort to be a successful contributor, and you cannot take submissions lightly.

The actual process of contributing is well-documented on the Express home page. The mechanics involve forking the project in your own GitHub account, cloning that fork, making your changes, pushing them back to GitHub, and creating a pull request (PR), which will be reviewed by one or more people on the project. If your submissions are small or are bug fixes, you may have luck simply submitting the pull request. If you're trying to do something major, you should communicate with one of the main developers and discuss your contribution. You don't want to waste hours or days on a complicated feature only to find that it doesn't fit with the maintainer's vision, or it's already being worked on by someone else.

The other way to contribute (indirectly) to the development of Express and Connect is to publish npm packages—specifically, middleware. Publishing your own middleware requires approval from no one, but that doesn't mean you should carelessly clutter the npm registry with low-quality middleware. Plan, test, implement, and document, and your middleware will enjoy more success.

If you do publish your own packages, here are the minimum things you should have:

Package name

While package naming is up to you, you obviously have to pick something that isn't already taken, which can sometimes be a challenge. npm packages now support namespacing by account, so you're not competing globally for names. If you're writing middleware, it's customary to prefix your package name with `connect-` or `express-`. Catchy package names that don't have any particular relation to what it does are fine, but even better is a package name that hints at what it does (a great example of a catchy but appropriate package name is `zombie`, for headless browser emulation).

Package description

Your package description should be short, concise, and descriptive. This is one of the primary fields that is indexed when people search for packages, so it's best to be descriptive, not clever (there's room for some cleverness and humor in your documentation, don't worry).

Author/contributors

Take some credit. Go on.

License(s)

This is often neglected, and there is nothing more frustrating than encountering a package without a license (leaving you unsure of whether you can use it in your project). Don't be that person. The MIT license is an easy choice if you don't want any restrictions on how your code is used. If you want it to be open source (and stay open source), another popular choice is the GPL license. It's also wise to include license files in the root directory of your project (they should start with *LICENSE*). For maximum coverage, dual-license with MIT and GPL. For an example of this in *package.json* and in *LICENSE* files, see my `connect-bundle` package.

Version

For the versioning system to work, you need to version your packages. Note that npm versioning is separate from commit numbers in your repository: you can update your repository all you like, but it won't change what people get when they use npm to install your package. You need to increment your version number and republish for changes to be reflected in the npm registry.

Dependencies

You should make an effort to be conservative about dependencies in your packages. I'm not suggesting constantly reinventing the wheel, but dependencies increase the size and licensing complexity of your package. At a minimum, you should make sure you aren't listing dependencies that you don't need.

Keywords
Along with description, keywords are the other major metadata used for people trying to find your package, so choose appropriate keywords.

Repository
You should have one. GitHub is the most common, but others are welcome.

README.md
The standard documentation format for both GitHub and npm is Markdown. It's an easy, wiki-like syntax that you can quickly learn. Quality documentation is vitally important if you want your package to be used. If I land on an npm page and there's no documentation, I usually just skip it without further investigation. At a minimum, you should describe basic usage (with examples). Even better is to have all options documented. Describing how to run tests goes the extra mile.

When you're ready to publish your own package, the process is quite easy. Register for a free npm account and then follow these steps:

1. Type `npm adduser`, and log in with your npm credentials.
2. Type `npm publish` to publish your package.

That's it! You'll probably want to create a project from scratch, and test your package by using `npm install`.

Conclusion

It is my sincere hope that this book has given you all the tools you need to get started with this exciting technology stack. At no time in my career have I felt so invigorated by a new technology (despite the odd main character that is JavaScript), and I hope I have managed to convey some of the elegance and promise of this stack. Though I have been building websites professionally for many years, I feel that, thanks to Node and Express, I understand the way the internet works at a deeper level than I ever have before. I believe that it's a technology that truly enhances understanding, instead of trying to hide the details from you, all while still providing a framework for quickly and efficiently building websites.

Whether you are a newcomer to web development, or just to Node and Express, I welcome you to the ranks of JavaScript developers. I look forward to seeing you at user groups and conferences, and most important, seeing what you will build.

Index

About the Author

Ethan Brown is director of technology at VMS, where he's responsible for the architecture and implementation of VMSPro, cloud-based software for decision support, risk analysis, and creative ideation for large projects. With over 20 years of programming experience, from embedded to the web, Ethan has embraced the JavaScript stack as the web platform of the future.

Colophon

The animals on the cover of *Web Development with Node and Express* are a black lark (*Melanocorypha yeltoniensis*) and a white-winged lark (*Melanocorypha leucopter*). Both birds are partially migratory and have been known to range far afield of their most suitable habitat in the steppes of Kazakhstan and central Russia. In addition to breeding there, male black larks will also winter in the Kazakh steppes, while females migrate southward. White-winged larks, on the other hand, fly farther west and north beyond the Black Sea during the winter months. The global range of these birds extends still farther: Europe constitutes a quarter to one-half of the global range of the white-winged lark and only five percent to a quarter of the global range of the black lark.

Black larks are so named for the black coloring that covers nearly the entire body of males of the species. Females, by contrast, resemble the coloring of the male in only their black legs and the black feathers of their underwings. A combination of dark and pale grays covers the rest of the female.

White-winged larks possess a distinctive pattern of black, white, and chestnut wing feathers. Gray streaks down the white-winged lark's back complement a pale white lower body. Males differ in appearance from females of the species only in the males' chestnut crowns.

Both black and white-winged larks evince the distinctively melodious call that has endeared larks of all variations to the imaginations of writers and musicians for centuries. Both birds eat insects and seeds as adults, and both birds make nests on the ground. Black larks have been observed carrying dung to their nests to build walls or lay a kind of pavement, though the cause for this behavior has not been identified.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery based on a black and white engraving from Lydekker's *The Royal Natural History*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.