O'REILLY®

Second Edition

Web Development with Node & Express

Leveraging the JavaScript Stack

Early Release

RAW & UNEDITED

Ethan Brown

1. Foreword
2. 1. Introducing Express
    a. The JavaScript Revolution
    b. Introducing Express
    c. Server-Side and Client-Side Applications
    d. A Brief History of Express
    e. Node: A New Kind of Web Server
    f. The Node Ecosystem
    g. Licensing
    h. Conclusion
3. 2. Getting Started with Node
    a. Getting Node
    b. Using the Terminal
    c. Editors
    d. npm
    e. A Simple Web Server with Node
        i. Hello World
        ii. Event-Driven Programming
        iii. Routing
        iv. Serving Static Resources
    f. Onward to Express
4. 3. Saving Time with Express

    a. Scaffolding
    b. The Meadowlark Travel Website
    c. Initial Steps
        i. Views and Layouts
        ii. Static Files and Views
        iii. Dynamic Content in Views
        iv. Conclusion
5. 4. Tidying Up
    a. Best Practices
    b. Version Control
    c. How to Use Git with This Book
        i. If You’re Following Along by Doing It Yourself
        ii. If You’re Following Along by Using the Official Repository
    d. npm Packages
    e. Project Metadata
    f. Node Modules
    g. Conclusion

Web Development with Node and Express

SECOND EDITION

Ethan Brown

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: Mary Treseler

Developmental Editors: Angela Rufino and Jennifer Pollock

Production Editor: Nan Barber

Copyeditor: Linley Dolby

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

Revision History for the Early Release

- 2019-03-05: First Early Release

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

978-1-492-05344-6

Foreword

The combination of JavaScript, Node, and Express is an ideal choice for web teams that want a powerful, quick-to-deploy technology stack that is widely respected in the development community and large enterprises alike.

Building great web applications and finding great web developers isn't easy. Great apps require great functionality, user experience, and business impact: delivered, deployed, and supported quickly and cost effectively. The lower total cost of ownership and faster time-to-market that Express provides is critical in the business world. If you are a web developer, you have to use at least some JavaScript. But you also have the option of using a *lot* of it. In this book, Ethan Brown shows you that you can use a lot of it, and it's not that hard thanks to Node and Express.

Node and Express are like machine guns that deliver upon the silver-bullet promise of JavaScript.

JavaScript is the most universally accepted language for client-side scripting. Unlike Flash, it's supported by all major web browsers. It's the fundamental technology behind many of the attractive animations and transitions you see on the Web. In fact, it's almost impossible not to utilize JavaScript if you want to achieve modern client-side functionality.

One problem with JavaScript is that it has always been vulnerable to sloppy programming. The Node ecosystem is changing that by providing frameworks, libraries, and tools that speed up development and encourage good coding habits. This helps us bring better apps to market faster.

We now have a great programming language that is supported by large enterprises, is easy-to-use, is designed for modern browsers, and is supplemented with great frameworks and libraries on both client-side and server-side. I call that revolutionary.

Steve Rosenbaum
President and CEO, Pop Art, Inc.

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 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 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 “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 very 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. Very 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 who were running high-performance, high-traffic websites, which created pressure on the Express team to to focus on performance, and Express now offers leading performance for hight-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

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 (Document Object Model) 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 word "web application" was officially introduced that year. I think of the period roughly between 1999 and 2012 as the "Web 2.0" era, where 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 2-3 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 1-3), and the user experiences a rich, multi-view 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 very 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. It is still relevant, and the conceptual difference between serving one HTML bundle or many is small. There is an SPA example in Chapter 14.

A Brief History of Express

Express's creator, 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 “plugin” 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.

> **NOTE**
>
> Express underwent a fairly substantial rewrite between 2.x and 3.0, then again between 3.x and 4.0. This book will focus 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 webservers is very minimal. Unlike IIS or Apache, which a person can spend many years mastering, Node is very 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 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 a later chapter. 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 frontrunners, and 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 frontrunners, 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.

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.

> **TIP**
>
> 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 very 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.

> **NOTE**
>
> Software is sometimes *dual licensed* (licensed under two different licenses). A very 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

Hopefully 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.

---

1 Often called "Just in Time" (JIT) compilation.

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."

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.

> **CAUTION**
>
> 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 only has 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 very 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 on 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.

> **CAUTION**
>
> 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 very easy to unthinkingly press, which leads to a very 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[1] (or an editor that has a vi mode). 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", which allow 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 very popular choice is Visual Studio Code (not to 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.

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.

> **NOTE**
>
> 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 version of Node, there is a project, nvm, that allows 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.

> **NOTE**
>
> *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*:

```
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....'))
```

> **NOTE**
>
> 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 up 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 plaintext 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 I’ll discuss in a later chapter.

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 on 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 plaintext 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.

> **NOTE**
>
> 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:

```
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).

> **TIP**
>
> 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.

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.jpg*. 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.jpg" alt="logo">`.

Now modify *helloWorld.js*:

```
const http = require('http'), 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.jpg':
      serveStaticFile(res, '/public/img/logo.jpg', 'image/jpeg')
      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....'))
```

> **NOTE**
>
> 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 function is simple: it calls `fs.readFile` to read the contents of the specified file. `fs.readFile` 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.

> **TIP**
>
> `__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

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.

---

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.

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 recreate 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 recreate 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 towards 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 a REST application, have no fear: the Meadowlark Travel website will expose REST services in addition to serving a functional website.

Initial Steps

Start by creating a new directory for your project: 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.

> **TIP**
>
> 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).

> **TIP**
>
> 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 --save express
```

Running `npm install` will install the named package(s) in the *node_modules* directory. If you specify the `--save` flag, it will 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":

```
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.`))
```

TIP

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 page indicating that the page doesn't exist.

> **NOTE**
>
> 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 if your `PORT` environment variable is set.

> **TIP**
>
> I highly recommend getting a browser plugin 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.

Let's add some routes for the home page and an About page. Before the 404 handler, we'll add two new routes:

```
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. For now, we're just returning plaintext with a status code of 200 (Express defaults to a status code of 200—you don't have to specify it explicitly).

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`, it is using `app.use`. `app.use` is the method by which Express adds *middleware*. We'll be covering middleware in more depth in Chapter 8, but for now, you can think of this as a catch-all handler for anything that didn't get matched by a route. This brings us to a very 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 sub-pages 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 Chapters 8 and 10.

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.

Where a view differs from a static resource (like an image or CSS file) is 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 very 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.

NOTE

> **NOTE**
>
> Pug was originally called “Jade,” and the name changed with the release of version 2 due to a trademark issue.

Pug is very 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.

> **NOTE**
>
> In the years following the original release of this book, React has taken the world like storm…which abstracts HTML away from fontend 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:

```
npm install --save express-handlebars
```

Then in *meadowlark.js*, modify the first few lines:

```
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. Not only does it become tedious to rewrite all that repetitive code for every page, it 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>
```

> **TIP**
>
> 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've got some views set up, we have to replace our old routes with new routes that use these views:

```
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 catch-all 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, called static content, is common and important, so we'll consider static content next.

Static Files and Views

Express relies on a *middleware* to handle static files and views. Middleware is a concept that will be covered in more detail in Chapter 8.

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 like 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:

```
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>
```

NOTE

> Remember that middleware is processed in order and static middleware—which is usually declared first or at least very early—it 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>

<p>Your fortune fo rthe day:</p>
<blockquote>{{fortune}}</blockquote>
```

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 5.

Conclusion

We've created a very 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 last 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.

> **TIP**
>
> 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!

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 only going to take 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

Hopefully 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 (DVCS). I narrowed my choices down to Git and Mercurial, and went with Git, due to 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 Git repository I am using for all of the examples and check out the associated tags 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've already got 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
```

> **NOTE**
>
> 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 only want to commit one or two files, or `git add -A` if you want to add all of your changes (including any files you might have deleted). Since we want to add all the work we've already done, we'll use:

```
git add -A
```

TIP

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
```

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. These directories represent the *end* of the chapter. That is, when you finish this chapter, you can see the finished code in the `ch04` directory. For chapters where there's a significant divergence, there may be additional directories, such as `ch08-jquery-file-upload`, which will be noted in the chapter text.

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 1.5.1. If you have an older version of this book, and you wish to see the latest changes and improvements to the code samples, you can check out the most current version: just be warned that the repository samples will be different than what you read here.

> **NOTE**
>
> 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 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.

NOTE

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 project. If you do go this route, and you 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.

The purpose of the *package.json* file is twofold: to describe your project and to list 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 --save`, 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.

> **NOTE**
>
> 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 recreate 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.

NOTE

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 you 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). 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*:

```
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.

> **NOTE**
>
> There are several ways to export functionality from a module. We will be covering different methods throughout the book and summarizing them later.

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 with 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:

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

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 very 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.

> **NOTE**
>
> 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", or ESM. If you've been writing JavaScript in React or another progressive front-end 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 front-end and back-end code. It's a good idea to be familiar with both.

About the Author

Ethan Brown is the Director of Technology at VMS, where he is responsible for the architecture and implementation of VMSPro®, cloud-based software for decision support, risk analysis, and creative ideation for large projects. He has over 20 years of programming experience, from embedded to the Web, and has embraced the JavaScript stack as the web platform of the future. He is author of the first edition of *Web Development with Node and Express* (O'Reilly).