MANNING

Express IN ACTION

Node applications with Express and its companion tools

Evan M. Hahn

MEAP

MEAP Edition

Manning Early Access Program

Express in Action

Node applications with Express and its companion tools

Version 10

welcome

Thank you so much for purchasing the MEAP for *Express in Action*! The book is really coming along, and I'm honored that you'll join me as I continue to work on it. This is an intermediate-level book aimed at intermediate-level JavaScript developers who want to start building web applications with Express and Node.js.

I've tried to make the content approachable to Node.js beginners. I want to emphasize how Express **works**, not just how to use it; I find this thinking invaluable when debugging and when deciding how to architect my software. I also hope to show how to extend Express with other tools to build rich web applications.

In this MEAP, we've got seven chapters of the planned eleven and an appendix.

The first chapter outlines Express at a high level, and is aptly-named "What is Express?". It explains what the Express framework is and isn't, its strengths and weaknesses, and its major conceptual hurdles.

Chapter 2 aims to give you a solid understanding of Express's foundations. It starts by focusing on low-level Node.js, and then transitions to showing how to do the same things with Express at a higher level. By the end of the chapter, you'll have a strong understanding of the framework which we'll build on for the rest of the book.

Chapter 4 digs into routing, which is one of Express's major features. Routing allows Express to appropriately sort requests coming into your website. The chapter starts with basic examples but gets in-depth. If any of the basic web workings confuse you, you can take a look at the appendix that describes HTTP in detail.

Chapter 5 talks about dynamically-generated HTML templates. We'll try both EJS and Jade, two popular templating engines for Node.

Chapter 6 is a chapter about databases. There are very few web applications that can function without **some** kind of data storage, and this chapter tackles that with the MongoDB database, a popular choice for Express developers.

Chapter 8 will have you setting up automated testing for your Express applications. After you write automated tests, you'll be much more confident that your code is robust and performing as it should. We'll look at some of the helpful tools out there.

Chapter 10 takes your applications to the real world—you'll deploy your applications to the Heroku cloud platform. You'll be able to write Express applications and show them to the world. This chapter also discusses compilation of assets, which includes the packaging of JavaScript and CSS to speed up your client-side resources.

Future chapters will discuss middleware in detail, how to secure Express applications, techniques for building APIs, and how to integrate Express with databases and real-time features.

I really hope you'll continue to give feedback in the Author Online forum. I'll be reading and responding to all of your comments, and your feedback is very helpful as I work on this book!

Thanks again! Honored that you'd join me on this journey.

—Evan Hahn

brief contents

Appendixes:

1 What is Express?

Before we talk about Express, we need to talk about Node.js.

For most of its life, the JavaScript programming language has lived inside of web browsers. It started as a simple scripting language for modifying small details of webpages, but grew into a complex language, with loads of applications and libraries. Many browser vendors like Mozilla and Google began to pump lots of resources into fast JavaScript runtimes, and Google Chrome and Mozilla Firefox got much faster JavaScript engines as a result.

In 2009, Node.js came around. Node took V8, Google Chrome's powerful JavaScript engine, out of the browser and made it able to run on servers. In the browser, developers had no choice but to choose JavaScript. In addition to Ruby, Python, C#, Java, or other languages, developers could now choose JavaScript when developing server-side applications.

JavaScript might not be the perfect language for everyone, but Node.js had some real benefits. For one, the V8 JavaScript engine is fast, and Node encourages an asynchronous coding style, making for faster code while avoiding multi-threaded nightmares. JavaScript also had a bevy of useful libraries because of its popularity. But the biggest benefit of Node.js is the ability to share code between browser and server. Developers don't have to do any kind of context switch when switching between client and server. Now they can use the same code and the same coding paradigms between two different JavaScript runtimes; the browser and the server.

Node.js caught on—people thought it was *pretty cool*.

Like browser-based JavaScript, Node.js provides a bevy of low-level features you'd need to build an application. But like browser-based JavaScript, its low-level offerins can be a bit verbose and difficult.

Enter Express.js. Express is a framework that acts as light layer on top of the Node.js web server, making it more pleasant to develop Node.js web applications.

Express.js is philosophically similar to jQuery. People want to add dynamic content to their webpages, but the “vanilla” browser APIs can be verbose, confusing, and limited in features. Developers often have to write a lot of boilerplate code. jQuery exists to cut down on this boilerplate code by simplifying the APIs of the browser and adding helpful new features. That's basically it.

Express is exactly the same. People want to make web applications with Node.js, but the “vanilla” Node.js APIs can be verbose, confusing, and limited in features. Developers often have to write a lot of boilerplate code. Express exists to cut down on this boilerplate code by simplifying the APIs of Node.js and adding helpful new features. That's basically it!

Like jQuery, Express aims to be extensible. It's hands-off about most parts of your applications' decisions and is easily extended with third-party libraries. Throughout this book and your Express career, you'll have to make decisions about your applications' architectures and you'll extend Express with a bevy of powerful third-party modules.

You probably didn't pick up this book for the “in short” definition, though. The rest of this chapter (and book, really) will discuss Express in much more depth.

> **NOTE** This book assumes that you are proficient in JavaScript, but not Node.js.

1.1 What is this Node.js business?

Node.js is no child's play.

When I first started using Node.js, I was confused. What *is* it?

Node.js (often shortened to "Node") is just a *JavaScript platform*—a way to run JavaScript. Most of the time, JavaScript is run in web browsers. But there's nothing about the JavaScript language that requires it to be run in a browser. It's a programming language just like Ruby or Python or C++ or PHP or Java. Sure, there are JavaScript runtimes bundled with all popular web browsers, but that doesn't mean that it has to be run there. If you were running a Python file called myfile.py, you would run `python myfile.py`. But you could write your own Python interpreter, call it SnakeWoman, and run `snakewoman myfile.py`. They did the same with Node; instead of typing `javascript myfile.js`, you type `node myfile.js`.

Running JavaScript outside of the browser lets us do a lot—anything a "regular" programming language could do, really—but it's mostly used for web development.

Okay, so we can run JavaScript on the server—why would we do this?

A lot of developers will tell you that Node.js is fast, and that's true. Node isn't the fastest thing on the market by any means, but it's fast for two reasons.

The first is pretty simple: the JavaScript engine is fast. It's based on the engine used in Google Chrome, which has a famously quick JavaScript engine. It can execute JavaScript like there's no tomorrow, processing thousands of instructions a second.

The second reason for its speed is in its ability to handle concurrency, and it's a bit less straightforward. Its performance comes from its asynchronous workings.

The best real-world analogy I can come up with is baking. Let's say I'm making some muffins. I have to prepare some batter. While I'm preparing the batter, I can't really do other things. I can't sit down and read a book, I can't cook something else, et cetera. But once I put the muffins in the oven, I don't just stand there looking at the oven until it's done—I go do something else. Maybe I start preparing more batter. Maybe I read a book. In any case, I don't have to wait for the muffins to be finished cooking for me to be able to do something.

In Node.js, a browser might request something from your server. You begin responding to this request and *another* request comes in. Let's say both requests have to talk to an external database. You can ask the external database about the first request, and *while* that external database is thinking, you can begin to respond to the second request. You're not doing two things at once, but when someone else is working on something, you're not held up waiting.

Other runtimes don't have this luxury built-in by default. Ruby on Rails, for example, can really only process one request at a time. To process more than one at a time, you effectively have to buy more servers. (There are, of course, many asterisks to this claim.)

Figure 1.1 demonstrates what this might look like:

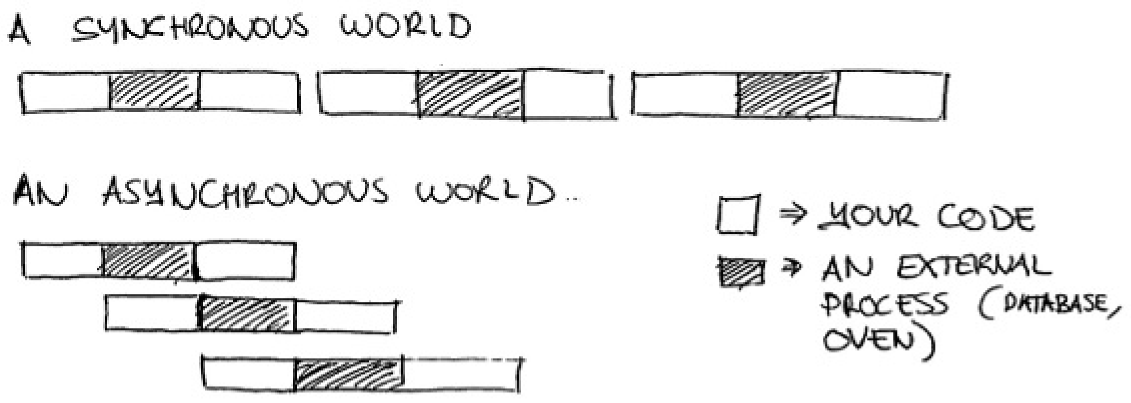


Figure 1.1 Comparing asynchronous code (like Node) to synchronous code. Note that asynchronous code can complete much faster, even though you're never executing your code in parallel.

I don't mean to tell you that Node.js is the fastest in the world because of its asynchronous capabilities. Node.js can really squeeze a lot of performance out of one CPU core, but it doesn't excel with multiple cores. Other programming languages truly allow you to actively do two things at once. To reuse the baking example: other programming languages let you buy more ovens so that you can bake more muffins simultaneously. Node is beginning to support this functionality but it is not as "first-class" in Node as it is in other programming languages.

Personally, I don't believe that performance is the biggest reason to choose Node. While it's faster than other scripting languages like Ruby or Python, I think the biggest reason is the fact that it's all one programming language.

Often, when you're writing a web application, you'll be using JavaScript. But before Node, you'd have to code everything in two different programming languages. You'd have to learn two completely different technologies, paradigms, and libraries. With Node, a backend developer can jump into front-end code and vice-versa. Personally, I think this is the most powerful feature of the runtime.

Other people seem to agree with me: people have created the "MEAN stack", which is an all-JavaScript web application stack consisting of MongoDB (a database controlled by JavaScript), Express, Angular.js (a front-end JavaScript framework), and Node.js. The "JavaScript everywhere" mentality is a huge benefit of Node.

Large companies are even getting behind Node; the list includes Walmart, the BBC, LinkedIn, and PayPal. It's no child's play.

1.2 What is Express?

Express is a relatively small framework that sits on top of Node.js's web server functionality to simplify its APIs and add helpful new features. It makes it easier to organize your application’s functionality with middleware and routing; it adds helpful utilities to Node.js’s HTTP objects; it facilitates the rendering of dynamic HTML views; it defines an easily-implemented extensibility standard. This book explores those features in a lot more depth, so all of that lingo will be demystified soon.

1.2.1 The functionality in Node.js

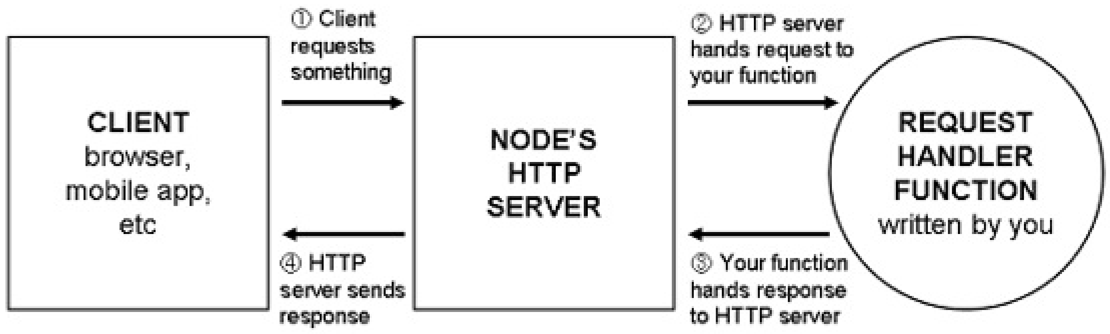
When you're creating a web application (or to be more precise, a web server) in Node.js, you write a single JavaScript function for your entire application. This function listens to a web browser’s requests, or the requests from a mobile application consuming your API, or any other client talking to your server. When a request comes in, this function will look at the request and determine how to respond. For example, if you visit the homepage in a web browser, this function could determine that you want the homepage and it will send back some HTML. If you send a message to an API endpoint, this function could determine what you want and respond with JSON (for example).

Imagine we're writing a web application that tells users the time and time zone on the server. It will work like this:

- If the client requests the homepage, our application will return an HTML page showing the time.

- If the client requests anything else, our application will return an HTTP 404 “Not Found” error and some accompanying text.

If you were building your application on top of Node.js without Express, a client hitting your server might look like Figure 1.2.



CLIENT browser, mobile app, etc
① Client requests something
NODE'S HTTP SERVER
② HTTP server hands request to your function
REQUEST HANDLER FUNCTION written by you
③ Your function hands response to HTTP server
④ HTTP server sends response

Figure 1.2 The flow of a request through a Node.js web application. Circles are written by you as the developer; squares are out of your domain.

The JavaScript function that processes browser requests in your application is called a **request handler**. There’s nothing too special about this; it is just a JavaScript function that takes the request, figures out what to do, and responds; that's it! Node's HTTP server handles the connection between the client and your JavaScript function so that you don't have to handle tricky network protocols.

In code, it's a function that takes two arguments: an object that represents the request and an object that represents the response. In our time/timezone application, the request handler function might check for the URL that the client is requesting. If they’re requesting the homepage, the request handler function should respond with the current time in an HTML page. Otherwise, it should respond with a 404. Every Node.js application is just like this: it's a single request handler function responding to requests. Conceptually, it's pretty simple!

The problem is that the Node APIs can get complex. Want to send a single JPEG file? That’ll be 45 lines of code. Want to create reusable HTML templates? Figure out how to do it yourself. Node.js's HTTP server is powerful, but it's missing a lot of features that you might want if you were building a real application.

Express was born to make it easier to write web applications with Node.js.

1.2.2 What Express adds to Node

In broad strokes, Express adds two big features to the Node.js HTTP server.

1. Express adds a number of helpful conveniences to Node.js's HTTP server, abstracting away a lot of its complexity. For example, where sending a single JPEG file is fairly complex in raw Node.js (especially if you have performance in mind), Express reduces it to just one line.
2. Express lets you refactor one monolithic request handler function into many smaller request handlers that only handle specific bits and pieces. This is more maintainable and more modular.

In contrast to Figure 1.2, Figure 1.3 shows how a request would flow through an Express application.

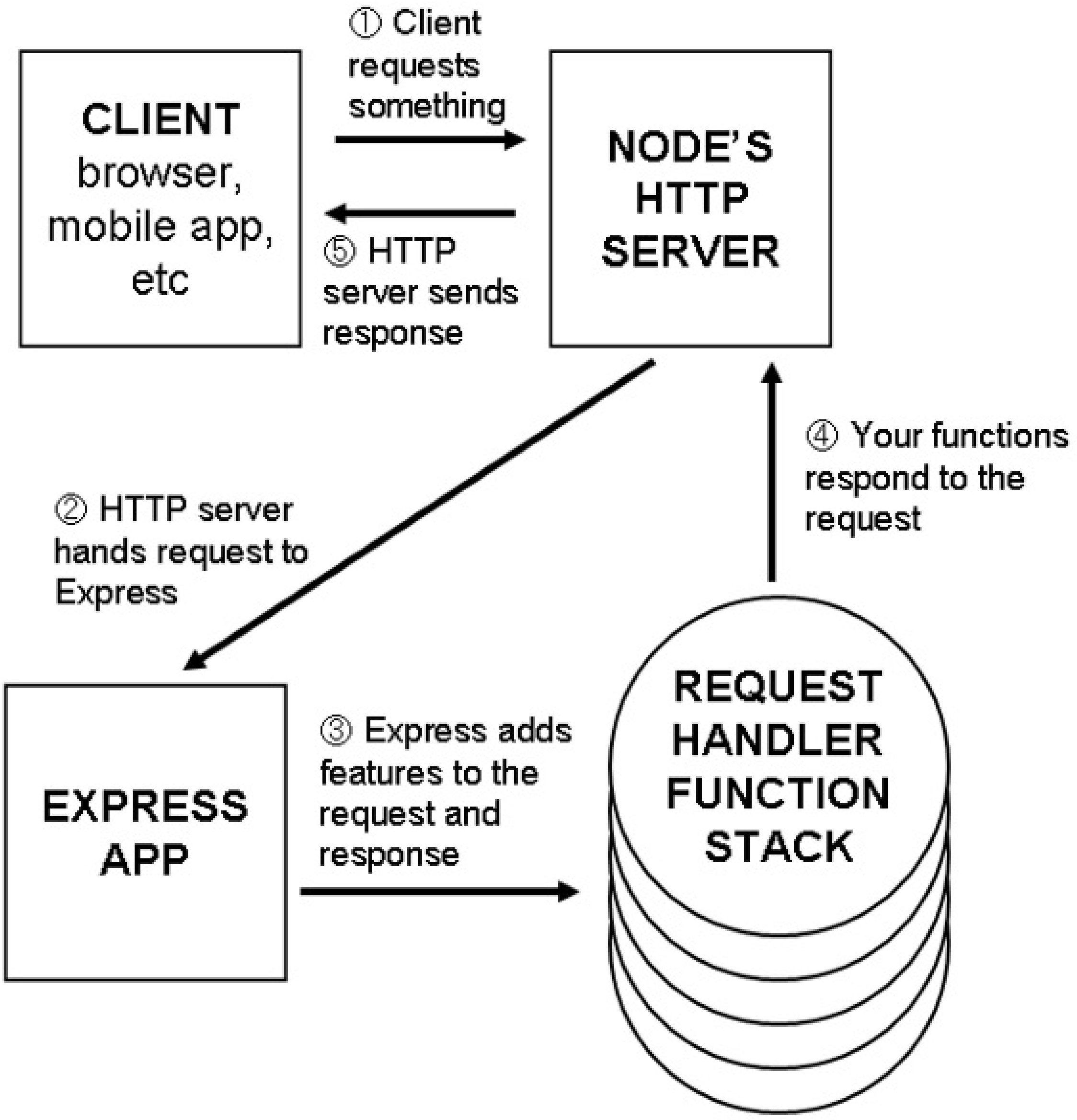

① Client requests something
CLIENT browser, mobile app, etc
NODE'S HTTP SERVER
⑤ HTTP server sends response
④ Your functions respond to the request
② HTTP server hands request to Express
REQUEST HANDLER FUNCTION STACK
EXPRESS APP
③ Express adds features to the request and response

Figure 1.3 The flow of a request through an Express. Once again, circles are code you write and squares are out of your domain.

This figure might look a little more complicated, but it's *much* simpler for you as the developer. There are essentially two things going on here:

1. Rather than one large request handler function, Express has you writing many smaller functions (many of which can be third-party and not written by you). Some functions are executed for every request (like a function that logs all requests, for example) and other functions are only executed sometimes (like a function that only handles the homepage or the 404 page, for example). Express has many utilities for partitioning these smaller request handler functions.
2. Request handler functions take two arguments: one is the request and

the other is the response. Node’s HTTP server provides you with some functionality; for example, Node’s HTTP server lets you extract the browser’s user-agent in one of its variables. Express augments this by adding extra features like easy access to the incoming request’s IP address and improved parsing of URLs. The response object also gets beefed up; Express adds things like the `sendFile`method, a one-line command that translates to about 45 lines of complicated file code. This makes it easier to write these request handler functions.

Instead of managing one monolithic request handler function with verbose Node.js APIs, you write multiple small request handler functions that are made more pleasant by Express and its easier APIs.

1.3 Express's minimal philosophy

Express is a framework, which means you'll have to build your app "the Express way". But "the Express way" isn't too opinionated; it doesn't give you a very rigid structure. That means you can build many different kinds of applications, from video chat applications to blogs to APIs.

It's very rare to build an Express app of any size that only uses Express. Express by itself probably doesn't do everything you need, and you'll probably find yourself with a large number of other libraries that you integrate into your Express applications. (We'll look at many of these libraries throughout the book.) You can have *exactly* what you need without any extra cruft, and it enables you to confidently understand every part of your application. In this way, it lends itself well to the "do one thing well" philosophy from the Unix world.

But this minimalism is a double-edged sword. On one hand, it's flexible and your apps are free of unused cruft. On the other hand, it does very little for you in comparison to other frameworks. This means that you make mistakes, you have to make far more decisions about your application's architecture, and you have to spend more time hunting for the right third-party modules. You get less out of the box.

While some might like a flexible framework, others might want more rigidity. For example, PayPal likes Express, but built a framework on top of it that more strictly enforces conventions for their many developers. Express doesn't care how you structure your apps, so two developers might make completely different decisions.

Because you're given the reins to steer your app in any direction, you might make an unwise decision that'll bite you later down the line. Sometimes, I look back on my still-learning-Express applications and thought, "Why did I do things this way?"

In order to write less code yourself, you wind up hunting for the right third-party packages to use. Sometimes, it's easy; there's one module that everyone loves and you love it too and it's a match made in heaven. Other times, it's harder to choose, because there are a lot of okay-ish ones or a small number. A bigger framework can save you that time and headache, and you'll simply use what you're given.

There's no right answer to this, and this book isn't going to try to debate the ultimate winner of the fight between big and small frameworks. But the fact of the matter is that Express is a minimalist framework, for better or for worse!

1.4 The core parts of Express

Alright, so Express is minimal, and it sugarcoats Node.js to make it easier to use. How does it do that?

When you get right down to it, Express has just four major features. There's a lot of conceptual stuff in the next few sections, but it's not just hand-waving; we'll get to the nitty-gritty details in the following chapters.

1.4.1 Middleware

As we saw above, raw Node.js gives us one request handler function to work with. The request comes into our function and the response goes out of our function.

Middleware is poorly-named, but it's a term that's not Express-specific and has been around for a while. The idea is pretty simple: rather than *one* monolithic request handler function, we call *several* request handler functions that each deal with a small chunk of the work. These smaller request handler functions are called middleware functions, or sometimes just "middleware".

Middleware can handle a variety of tasks, from logging requests to sending static files to setting HTTP headers. For example, the first middleware function we might use in an application is a logger—log every request that comes into our server. When the logger is all done logging, it will continue onto the next middleware in the chain. This next middleware function might authenticate users. If they're visiting a forbidden URL, respond with a "not authorized" page. If they're allowed to visit it, continue to the next function in the chain. The next function might send the homepage and be done. An illustration of two possible options is shown in Figure 1.4.

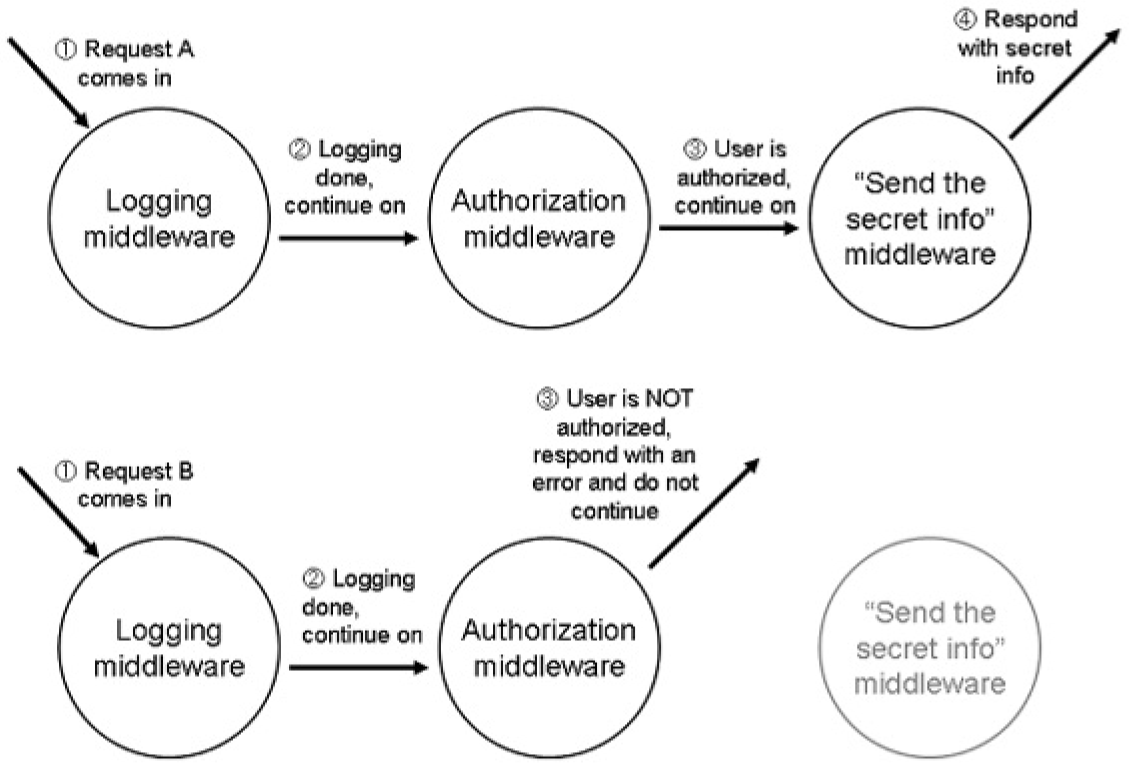

① Request A comes in
Logging middleware
② Logging done, continue on
Authorization middleware
③ User is authorized, continue on
"Send the secret info" middleware
④ Respond with secret info
① Request B comes in
Logging middleware
② Logging done, continue on
Authorization middleware
③ User is NOT authorized, respond with an error and do not continue
"Send the secret info" middleware

Figure 1.4 Two requests flowing through middleware functions. See that middleware sometimes continues on, but sometimes responds to requests.

In Figure 1.4, the logging middleware is first in the chain and is always called, so something will always noted in the log file. Next, the logging middleware continues to the next one in the chain, the authorization middleware. This middleware decides, by some decree, whether the user is authorized to keep going. If they are, it continues on to the next middleware in the chain.

Otherwise, send a “you’re not authorized!” message to the user and halt the chain. (This message could be an HTML page or a JSON response or anything else, depending on the application.) Finally, the last middleware, if it’s called, will send some secret information and not continue to any further middleware in the chain. (Once again, this last middleware can send any kind of response, from HTML to JSON to an image file.)

One of the biggest features of middleware is that it's relatively standardized, which means that *lots* of people have developed middleware for Express (including folks on the Express team). That means that if you can dream up the middleware, someone has probably made it. There's middleware to compile static assets like LESS and SCSS; there’s middleware for security and user authentication; there’s middleware to parse cookies and sessions.

1.4.2 Routing

Routing is better named than middleware. Like middleware, it breaks the one monolithic request handler function into smaller pieces. Unlike middleware, however, these request handlers are executed conditionally, depending on what URL and HTTP method a client sends.

For example, we might build a webpage with a homepage and a guestbook. When the user sends an HTTP GET to the homepage URL, Express should send the homepage. But when they visit the guestbook URL, it should send them the HTML for the guestbook, not for the homepage! And if they post a comment in the guestbook (with an HTTP POST to a particular URL), this should update the guestbook. Routing allows you to partition your application's behavior by route.

The behavior of these routes is, like middleware, defined in request handler functions. When the user visits the homepage, it will call a request handler function, written by you. When the user visits the guestbook URL, it will call another request handler function, also written by you.

Express applications have middleware *and* routes; they complement one another. For example, you might want to log all of the requests, but you'll also want to serve the homepage when the user asks for it.

1.4.3 Sub-applications

Express applications can often be pretty small, even fitting in just one file. As your applications get larger, though, you'll start to want to break things up into multiple folders and files. Express is unopinionated about how you scale your app, but it provides one important feature that's super helpful: sub-applications. In Express lingo, these mini-applications are called **routers**.

Express allows you to define routers that can be used in larger applications. Writing these sub-applications is almost exactly like writing "normal-sized" ones, but it allows you to further compartmentalize your app into smaller pieces. For example, you might have an administration panel in your app, and that can function pretty differently from the rest of your app. You could put the admin panel code side-by-side with the rest of your middleware and routes, but you can also create a sub-application for your admin panel. Figure 1.5 shows how an Express application might be broken up with routers.

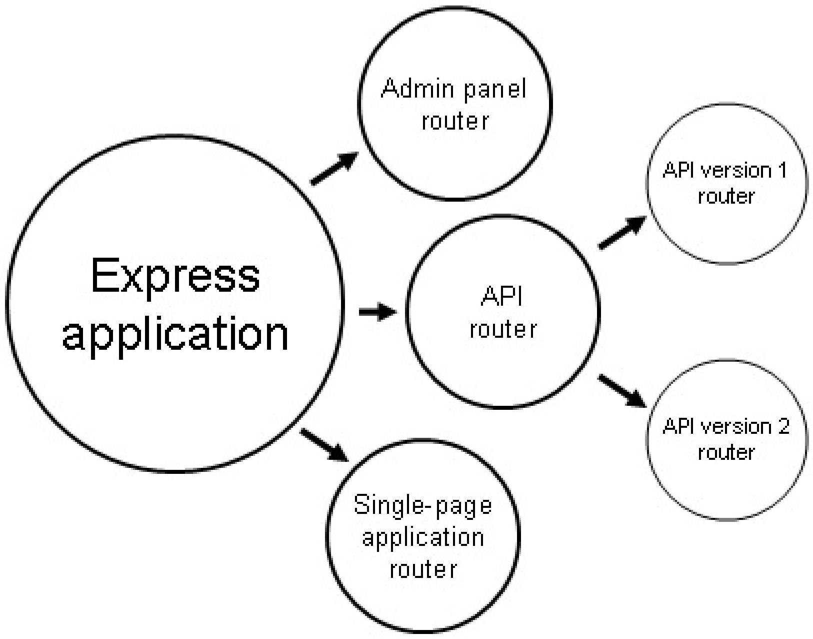


Figure 1.5 An example diagram showing how a large application could be broken up into routers.

This feature doesn't really shine until your applications get large, but when they do, it's hugely helpful.

1.4.4 Conveniences

Express applications are made up of middleware and routes. Both of them have you writing request handler functions, so you'll be doing that a lot!

To make these request handler functions easier to write, Express has added a bunch of niceties. In raw Node.js, if you want to write a request handler function that sends a JPEG file from a folder, that's a fair bit of code. In Express, that's just one call to the `sendFile` method. Express has a bunch of functionality for rendering HTML more easily, where Node.js keeps mum. It also comes with a bunch of functions that make it easier to parse requests as they come in, like grabbing the client's IP address.

Unlike the features above, these conveniences don't conceptually change how you organize your app, but they can be super helpful.

1.5 The ecosystem surrounding Express

Express, like any tool, doesn't exist in a vacuum.

It lives in the Node.js ecosystem, so you have a bevy of third-party modules that can help you, such as interfaces with databases. Because Express is extensible, lots of developers have made third-party modules that work well with Express (rather than general Node.js), such as specialized middleware or ways to render dynamic HTML.

1.5.1 Express versus other web application frameworks

Express is hardly the first web application framework, nor will it be the last.

Express isn't the only framework in the Node.js world. Perhaps its biggest "competitor" is called Hapi.js. Like Express, it's an unopinionated, relatively small framework that has routing and middleware-like functionality. It's different from Express in that it doesn't aim to smooth out Node.js's built-in HTTP server module, but to build a rather different architecture. It's a pretty mature framework developed by the folks at Walmart, and is used by Mozilla, OpenTable, and even the npm registry! While I doubt there's much animosity between Express developers and Hapi developers, Hapi is the biggest "competitor" to Express.

There are larger frameworks in the Node.js world as well, perhaps the most popular of which is the full-stack Meteor. While Express is unopinionated about how you build your applications, Meteor has a strict structure. While Express only deals with the HTTP server layer, Meteor is full-stack, running code on both client and server. This are simply design choices—one is not inherently better than the other.

Like Express piles features atop Node.js, some folks have decided to pile features atop Express. Some folks at PayPal created Kraken; while Kraken is technically just Express middleware, it sets up a *lot* of your application, from security defaults to bundled middleware. Sails.js is another up-and-coming framework built atop Express that adds databases, WebSocket integration, API generators, an asset pipeline, and more. Both of these frameworks are more opinionated than Express by design.

Express has several features, just one of which is middleware. Connect is a web application framework for Node.js that's *just* the middleware layer. Connect doesn't have routing or conveniences; it's just middleware. Express used to use Connect for its middleware layer, and while it now does middleware without Connect, Express middleware is completely compatible with Connect middleware. That means that any middleware that works in Connect also works in Express, which adds a huge number of helpful third-party modules to your arsenal.

This is JavaScript, so there are *countless* other Node.js web application frameworks out there, and I'm sure I've offended someone by not mentioning theirs.

Outside of the Node.js world, there are comparable frameworks.

Express was very much inspired by Sinatra, a minimal web application framework from the Ruby world. Sinatra, like Express, has routing and middleware-like functionality. Sinatra has inspired many clones and reinterpretations many other programming languages, so if you've ever used Sinatra or a Sinatra-like framework, Express will be familiar. Express is also like Bottle and Flask from the Python world.

Express isn't as much like Python's Django or Ruby on Rails or ASP.NET or Java's Play; those are larger, more opinionated frameworks with lots of features. Express is also unlike PHP; while it *is* code running on the server, it’s not as tightly coupled with HTML as “vanilla” PHP is.

This book should tell you that Express is better than all of these other frameworks, but it can’t—Express is simply one of the many ways to build a server-side web application. It has some real strengths that other frameworks don’t have, like Node.js’s performance and the ubiquitous JavaScript, but it does less for you than a larger framework might do, and some people don’t think JavaScript is the finest language out there. We could argue about which is best forever and never come to an answer, but it’s important to see where Express fits into the picture.

1.5.2 What Express is used for

In theory, Express could be used to build any web application. It can process incoming requests and respond to them, so it can do things that you can do in most of the other frameworks mentioned above. Why would you choose Express over something else?

One of the benefits of writing code in Node.js is the ability to share JavaScript code between the browser and the server. This is helpful from a code perspective because you can literally run the same code on client and server. It's also very helpful from a mental perspective; you don't have to get your mind in "server mode" and then switch your mind into "client mode"—it's all the same thing at some level. That means that a frontend developer can write backend code without having to learn a whole new language and its paradigms, and vice-versa. There is some learning to do—this book wouldn't exist otherwise!—but a lot of it is familiar to front-end web developers.

Express helps you do this, and people have come up with a fancy name for one arrangement of an all-JavaScript stack: the MEAN stack. Like the "LAMP" stack stands for Linux, Apache, MySQL, and PHP, "MEAN" stands for MongoDB (a JavaScript-friendly database), Express, Angular (a frontend JavaScript framework), and Node.js. People like the MEAN stack because it's full-stack JavaScript and you get all of the aforementioned benefits.

Express is often used to power single-page applications, or SPAs. SPAs are very JavaScript-heavy on the frontend, and they usually require a server component. The server is usually required to simply serve the HTML, CSS, and JavaScript, but there's often a REST API, too. Express can do both of these things quite well; it's great at serving HTML and other files, and it's great at building APIs. Because the learning curve is relatively low for frontend developers, they can whip up a simple SPA server without too much new learning.

When you write applications with Express, you can't get away from using Node.js, so you're going to have the "E" and the "N" parts of the MEAN stack, but the other two parts (MongoDB and Angular) are up to you because Express is unopinionated. Want to replace Angular with Backbone.js on the frontend? Now it's the MEBN stack. Want to use SQL instead of MongoDB? Now it's the SEAN stack. While MEAN is a common bit of lingo thrown around and a popular configuration, you can choose whichever you want. In this book, we'll cover the MongoDB database, so we'll get the "MEN" stack.

Express also fits in side-by-side with a lot of real-time features. While other programming environments can support real-time features like WebSockets and WebRTC, Node.js seems to get more of that than other languages and frameworks. That means that you can Because Node gets it, Express gets it too.

1.5.3 Third-party modules for Node and Express

The first few chapters of this book talk about "core" Express—that is, things that are baked into the framework. In very broad strokes, this is routes and middleware. But more than half of the book covers how to integrate Express with third-party modules.

There are *loads* of third-party modules for Express. Some are made specifically for Express and are compatible with its routing and middleware features. Others aren't made for Express specifically and work well in Node.js, so they also work well with Express.

In this book, we'll pick a number of third-party integrations and show some examples. But because Express is unopinonated, none of the contents of this book are the only options. If I cover Third-Party Tool X in this book, but you prefer alternative Third-Party Tool Y, you can swap them out.

Express has some small features for rendering HTML. If you’ve ever used “vanilla” PHP or a templating language like ERB, Jinja2, HAML, or Razor, you’ve dealt with rendering HTML on the server. Express doesn’t come with any templating languages built in, but it plays nicely with almost every Node-based templating engine, as we’ll see. Some popular templating languages come with Express support, while others need a simple helper library. In this book, we’ll look at two options: EJS (which looks a lot like HTML) and Jade (which tries to fix HTML with a radical new syntax).

Express doesn’t have any notion of a database. You can persist your application’s data however you choose; in files, in a relational SQL database, or in another kind of data storage mechanism. In this book, we’ll cover the popular MongoDB database for data storage. As we talked about above, you should never feel “boxed in” with Express—if you want to use another data store, Express will let you.

Users often want their applications to be secure. There are a number of helpful libraries and modules (some for “raw” Node and some for Express) that can tighten the belt of your Express applications. We’ll explore all of this in the chapter about security (which is one of my favorite chapters, personally). We’ll also talk about testing our Express code to make sure that the code powering our apps is robust.

An important thing to note: there’s no such thing as an “Express module”—only a Node module. A Node module can be *compatible* with Express and work well with its API, but they’re all just JavaScript served from the npm registry and you install them in just the same way. Just like in other environments, some modules integrate with other modules, where others can sit alongside. At the end of the day, Express is just a Node module just like any other.

Getting help when you need it

I really hope this book is helpful and chock-full of knowledge, but there's only so much wisdom one author can jam into a book. At some point, you're going to need to spread your wings and find answers. Let me do my best to guide

you:
For API documentation and simple guides, the official http://expressjs.com/ is the place to go. You can also find example applications all throughout the Express repository, at https://github.com/strongloop/express/tree/master/examples . I found these examples helpful when trying to find the "right" way to do things. There are loads of examples in there; check them out!
For Node modules, you'll be using Node's built-in npm tool and installing things from the registry at https://www.npmjs.org/ . If you need help finding good modules, I'd give Substack's "finding modules" a read at http://substack.net/finding_modules. It's a great summary of how to find quality Node packages.
Express used to be built on another package called Connect, and is still largely compatible with Connect-made modules. If you can't find a module for Express, you might have luck searching for Connect. This also applies if you're searching for answers to questions.
And as always, use your favorite search engine.

1.6 The obligatory hello world

Every introduction to a new code thing needs a “hello world”, right?

Let's take a look at one of the simplest Express applications we can build: the Hello World. We'll delve into this in much greater detail throughout the book, so don't worry if not all of this makes sense right now.

Here's Hello World, in Express:

Listing 1.1 “Hello World” in Express

```
var express = require("express");  #A

var app = express();  #B

app.get("/", function(request, response) {  #C
  response.send("Hello world!");            #C
});                                          #C

app.listen(3000, function() {                        #D
  console.log("Express app started on port 3000."); #D
});                                                  #D
```

#A Require Express and put it in a variable.
#B Call express() to make a new Express application, and put it inside of a variable called "app".
#C When someone sends a request to the root of your site (at "/"), they will be sent "Hello world!".
#D Start the Express server on port 3000 and log that the server has started.

Once again: if not all of this makes sense to you, don't worry! But you might be able to see that we're creating an Express application, defining a route that responds with "Hello world!", and starting our app on port 3000. There are a few steps you'll need to do to run this—all of that will become clear in the next couple of chapters.

We'll learn all of Express's secrets soon.

1.7 Summary

In this chapter, you learned that:

- Node.js is a powerful tool for writing web applications, but it can be cumbersome to do so. Express was made to smooth out that process.
- Express is a minimal, unopinionated framework that's flexible.
- Express has a few key features:
- Middleware, a way to break your app into smaller bits of behavior. Generally, middleware is called one by one, in a sequence.
- Routing similarly breaks your app up into smaller functions that are executed when the user visits a particular resource; for example, showing the homepage when the user requests the homepage URL.
- Routers can further break up large applications into smaller, composable sub-applications.
- Most of your Express code involves writing request handler functions, and Express adds a number of conveniences when writing these.

2 The Basics of Node.js

In the first chapter, we talked about what Node.js is. We discussed that it's JavaScript, it is asynchronous in nature, and that it has a rich set of third-party modules. If you're like me, you didn't totally understand these things when you first started with Node. This chapter aims to give the intro to Node that I wish I had: short and sweet.

We'll talk about

- Installing Node
- How to use its module system
- How to install third-party packages
- Some examples of its fancy "evented I/O".
- Some tricks for running your Node code nicely.

> **NOTE** I'm going to assume that you know a fair bit of JavaScript and that you don't want an extremely thorough knowledge of Node from this chapter. I'm also going to assume that you have a working understanding of how to use the command line. If this whirlwind introduction to Node is a little *too* whirlwind, check out Node.js in Action at http://www.manning.com/cantelon/ for more.

Let's get started.

2.1 Installing Node

A theme of the JavaScript world is an overwhelming number of choices, and Node's installation is no exception; there are a lot of different ways to get Node running on your system.

The official downloads page at http://nodejs.org/download/ has a number of download links for pretty much every platform—Windows, Mac, and Linux. The choice of platform should be obvious—choose the one for your operating system. If you're not sure if your system is 32-bit or 64-bit, search the web to try to answer it because you'll get a lot of performance benefits from choosing 64-bit if it's available. Mac and Windows users have the option to download a binary or an installer, and I'd recommend the latter.

If you have a package manager on your system, you can use that instead. Node.js is available on various package mangers, including apt-get, Homebrew, and Chocolatey. You can check out the official "Installing Node.js via package manager" guide at https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager.

If you're on Mac or Linux, I highly recommend the **Node Version Manager**, or NVM, found at https://github.com/creationix/nvm. If you're on Windows, NVMW at https://github.com/hakobera/nvmw is a port for Windows users. These programs allow you to easily switch between Node versions, which is great if you want to have the stable version of Node and the exciting experimental pre-release versions. It also allows you to easily upgrade Node when new versions are released. NVM has a couple of other benefits that I like, too: it's trivial to uninstall, and it doesn't need administrator (root) access to install it on your system.

NVM is a one-line install that you can copy-paste and run from the instructions at https://github.com/creationix/nvm (or https://github.com/hakobera/nvmw fo the Windows version).

In any case, get Node installed!

2.1.1 Running your first Node script

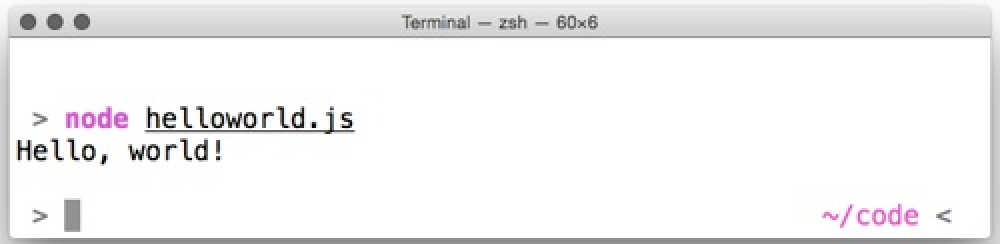
How ever you chose to install Node, it's time to run something! Let's built the classic "hello world". Create a file called `helloworld.js` and put the following inside:

Listing 2.1 helloworld.js

```
console.log("Hello, world!");
```

We call the `console.log` function with the argument we want to print: the string "Hello, world!". If you've ever used the console when writing browser-based JavaScript, this should look familiar.

To run this, type `node helloworld.js`. (You may have to `cd` into the directory where `helloworld.js` lives.) If everything works well, you should see the text appear on the screen! The output will look something like Figure 2.1.



```
> node helloworld.js
Hello, world!

>                                                  ~/code <
```

Figure 2.1 The result of running our "hello world" code.

2.2 Using modules

Most programming languages have a way of including File A from File B so that you can split your code into multiple files. C and C++ have `#include`; Python has `import`; Ruby and PHP have `require`. Some languages like C# do this kind of cross-file communication implicitly at compile time.

For most of its life, the JavaScript language didn't have an official way to do this. To solve this problem, people built things that concatenated JavaScript files together into one file, or built dependency loaders like RequireJS. A lot of web developers simply fill their webpages with `<script>` tags.

Node wanted to solve this problem elegantly and they implemented a standard module system called *CommonJS*. At its core, CommonJS lets you include code from one file in another.

There are three major components to this module system: requiring built-in modules, requiring third-party modules, and making your own modules. Let's see how they work.

2.2.1 Requiring built-in modules

Node has a number of built-in modules, ranging from file system access in a module called "fs" to utility functions in a built-in module called "util".

A common task when building web applications with Node is parsing of the URL. When a browser sends a request to your server, they'll ask for a specific URL. Perhaps they'll ask for the homepage; perhaps they'll ask for the about page; perhaps they'll ask for something else. These URLs come in as strings but we often want to parse them to get more information about them. Node has a built-in URL parser module; let's use it to see how to require packages.

Node's built-in `url` module exposes a few functions, but the "big kahuna" is a function called `parse`. It takes a URL string and extracts useful information, like the domain or the path.

We'll use Node's `require` function to use the `url` module. `require` is similar to keywords like `import` or `include` in other languages. `require` takes the name of a package as a string argument and returns a package. There's nothing special about the object that's returned—it's often an object, but it could be a function or a string or a number. Here's how we might use the URL module:

Listing 2.2 Requiring Node's URL module

```
var url = require("url");   #A
var parsedURL = url.parse("http://www.example.com/   #B
                      [CA]profile?name=barry");  #B

console.log(parsedURL.protocol);  // "http:"
console.log(parsedURL.host);      // "www.example.com"
console.log(parsedURL.query);     // "name=barry"
```

#A This requires a module called "url" and puts it in a variable called "url". It's just a convention that these are the same.
#B This uses url.parse. This would throw an undefined error if we didn't have the first line requiring the module.

In the above example, `require("url")` returns an object that has the parse function attached. Then we use it as we would any object!

If you save this as `url-test.js`, you can run it with `node url-test.js`. It will print the protocol, host, and query of our example URL.

Most of the time, when you are requiring a module, you'll put in a variable that has the same name as the module itself. The above example puts the url module in a variable of the same name: url.

But you don't have to do that! We could have put it in a variable with a completely different name, if we wanted. The following example illustrates that:

Listing 2.3 Requiring things into different variable names

```
var theURLModule = require("url");

var parsedURL = theURLModule.parse("http://example.com");
// ...
```

It's a loose convention to give the variables the same name as what you're requiring to prevent confusion, but there's nothing enforcing that in code.

2.2.2 Requiring third-party modules with package.json and npm

Node has several built-in modules, but they're rarely enough; third-party packages are indispensible when making applications. And this is a book about a third-party module, after all, so you should definitely know how to use them!

The first thing we need to talk about is `package.json`. Every Node project sits in a folder, and at the root of every Node project, there's a file called `package.json`. (When I say "every Node project", I mean every single one, from third-party packages to applications. You'll likely never build a Node project without one.)

"package dot json" is a pretty simple JSON file that defines project metadata like the name of the project, its version, and its authors. It also defines the project's dependencies.

Let's make a simple app. Make a new folder and save this to `package.json`:

Listing 2.4 A simple package.json file

```
{
  "name": "my-fun-project",    #A
  "author": "Evan Hahn",       #B
  "private": true,             #C
  "version": "0.2.0",          #D
  "dependencies": {}           #E
}
```

#A Define the name of your project.
#B Define the author. This can be an array of authors if you have many, and it probably isn't "Evan Hahn".
#C This says "this is a private project; don't let me be published to the package registry for anyone to use."
#D Define the version of the package.
#E Notice that this project has no dependencies yet. We'll install some soon!

Now that we've defined our package, we can install its dependencies.

When you install Node, you actually get *two* programs: Node (as you might expect) and something called *npm* (deliberately lowercase). npm is an official helper for Node that helps you with your Node projects.

npm is often called the "Node Package Manager", but its unabbreviated name has never been explicitly stated—its website randomly shows names like "Never Poke Monkeys" or "Nine Putrid Mangos". It may evade the "package manager" moniker because it does much more than that, but package management is perhaps its biggest feature, which we'll use now.

Let's say we want to use Mustache (see https://mustache.github.io/), a standard little templating system. It lets you turn template strings into "real" strings. An example explains it best:

Listing 2.5 An example of the Mustache templating system

```
// Returns "Hello, Nicholas Cage!"
Mustache.render("Hello, {{first}} {{last}}!", {
  first: "Nicholas",
  last: "Cage"
});

// Returns "Hello, Sheryl Sandberg!"
Mustache.render("Hello, {{first}} {{last}}!", {
  first: "Sheryl",
  last: "Sandberg"
});
```

Let's say that we want to write a simple Node application that greets Nicholas Cage with the Mustache module.

From the root of this directory, run `npm install mustache --save`. (You must run this command from the root of this directory so that `npm` knows where to put things.) This command will create a new folder in this directory called`node_modules`. Then it downloads the latest version of the Mustache package and puts it into this new `node_modules` folder (go look inside to check it out!). Finally, the `--save` flag will add it to your `package.json`. Your package.json file should look similar, but it will now have the latest version of the Mustache package:

Listing 2.6 A simple package.json file

```
{
  "name": "my-fun-project",
  "author": "Evan Hahn",
  "private": true,
  "version": "0.2.0",
  "dependencies": {
    "mustache": "^2.0.0"   #A
  }
}
```

#A Notice this new line. Your dependency version may be newer than the one here.

If you left off the `--save` flag, you'd see the new `node_modules` folder and it'd have Mustache inside, but nothing would be present in your `package.json`. The reason you want dependencies listed in your `package.json` is so that someone else can install the dependencies later if you gave them the project—they just have to run `npm install` with no arguments. Node projects typically have dependencies listed in their `package.json` but they don't come with the actual dependency files (they don't include the `node_modules` folder).

Now that we've installed it, we can use the Mustache module from our code!

Listing 2.7 Using the Mustache module

```
var Mustache = require("mustache");  #A
var result = Mustache.render("Hi, {{first}} {{last}}!", {
  first: "Nicolas",
  last: "Cage"
});
console.log(result);
```

#A Notice how we require Mustache—just like a built-in module.

Save the code above to `mustache-test.js` and run it with `node mustache-test.js`. You should see the text "Hi, Nicholas Cage!" appear.

And that's it! Once it's installed into `node_modules`, you can use Mustache just like you would a built-in module. Node knows how to require modules inside the `node_modules` folder.

When you're adding dependencies, you can also manually edit `package.json` and then run `npm install`. You can also install specific versions of dependencies or install them from places other than the official npm registry; see more at the `npm install` documentation (https://docs.npmjs.com/cli/install).

npm init

npm does much more than just install dependencies. For example, it allows you to autogenerate your package.json file. You can create package.json by hand yourself, but npm can do it for you.

In your new project directory, you can type npm init. It will ask you a bunch of questions about your project—project name, author, version—and when it's all done, it will save a new package.json. There's nothing sacred about this generated file; you can change it all you want. But it can save you a bit of time when creating these package.json files.

2.2.3 Defining your own modules

We've been using other peoples' modules for this whole chapter—now let's learn how to define our own.

Let's say we want a function that returns a random integer between 0 and 100. Without any module magic, that function might look like this:

Listing 2.8 A function that returns a random integer between 0 and 100

```
var MAX = 100;

function randomInteger() {
   return Math.floor((Math.random() * MAX));
}
```

This shouldn't be too earth-shattering; this might be how you'd write that function in a browser context. But in Node, we can't just save this into a file and call it a day; we need to choose a variable to export, so that when other files`require` this one, they know what to grab. In this case, we'll be exporting `randomInteger`.

Try saving this into a file called `random-integer.js`:

Listing 2.9 random-integer.js

```
var MAX = 100;

function randomInteger() {
   return Math.floor((Math.random() * MAX));
}

module.exports = randomInteger; #A
```

#A This line does the actual "exporting" of the module for other files.

The last line is the only thing that might be foreign to someone new to Node. You can only export one variable, and you'll choose that variable by setting `module.exports` to it. In this case, the variable we're exporting is a function. In this module, `MAX` is not exported, so that variable won't be available to anyone who requires this file. Nobody will be able to require it--it'll stay private to the module.

> **REMEMBER** module.exports can be anything you want. Anything to which you can assign a variable can be assigned to `module.exports`. It's a function in this example, but is often an object. It could even be a string or a number or an array if you'd like!

Now, let's say we wanted to use our new module. In the same directory as `random-integer.js`, save a new file. It doesn't matter what you call it (so long as it's not `random-integer.js`), but let's call it `print-three-random-integers.js`.

Listing 2.10 Using our module from another file

```
var randomInt = require("./random-integer");  #A
console.log(randomInt());  // 12
console.log(randomInt());  // 77
console.log(randomInt());  // 8
```

#A Note that this is a relative path.

We can now require it just like any other module, but we have to specify the path using the dot syntax. Other than that, it's exactly the same! You can use it as you would another module.

You can run this code just like any other, by running `node print-three-random-integers.js`. If you did everything correctly, it'll print three random numbers between 0 and 100!

You might try running `node random-integer.js`, and you'll notice that it doesn't appear do anything. It exports a module, but defining a function doesn't mean the function will run and print anything to the screen!

> **NOTE** This book only covers making local modules within a project. If you're interested in publishing open source packages for everyone to use, check out the guide on my website at http://evanhahn.com/make-an-npm-baby .

That's a quick intro to Node's module system!

2.3 Node: an asynchronous world

In Chapter 1, we discussed the asynchronous nature of Node. I used a "let's bake muffins" analogy. While I'm preparing the batter for my muffins, I can't do other substantive things; I can't read a book; I can't prepare more batter, et cetera. But once I put the muffins in the oven, I can do other things. I don't just stand there staring at the oven until it beeps—I could go for a jog. When the oven beeps, I'm back on muffin duty and I'm occupied again.

A key point here is that *I'm* never doing two things at once. Even if multiple things are happening at once (I could be jogging while the muffins are baking), *I'm* only doing one thing at a time. This is because the oven isn't *me*—it's an external resource.

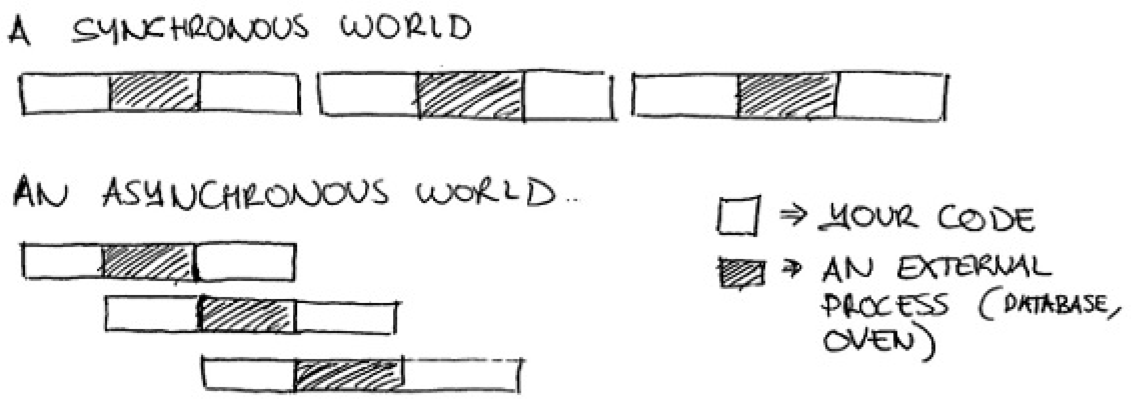


Figure 2.2 Comparing an asynchronous world (like Node) to a synchronous one.

Node's asynchronous model works similarly. A browser might request a 100 megabyte cat picture from your Node-powered web server. You begin to load this big photo off of the hard disk. As far as we're concerned, the hard disk is an external resource, so we ask it for the file and then we can move onto other things while we wait for it to load.

While you're loading that file, a second request comes in. You don't have to wait for the first request to finish completely—while you're waiting for the hard disk to finish what it was working on, you can start parsing the second request. Once again: Node is never really doing two things at once, but when an external resource is working on something, you're not held up waiting.

The two most common external resources you'll deal with in Express are:

1. Anything involving the file system—like reading and writing files from your hard drive
2. Anything involving a network—like receiving requests, sending responses, or sending your own requests over the Internet

Conceptually, that's about it!

In code, these asynchronous things are handled by callbacks. You've probably done something like this if you've ever done an AJAX request on a webpage; you send a request and pass a callback. When the browser has finished your request, it'll call your callback. Node works in exactly the same way.

For example, let's say you're reading a file called `myfile.txt` from disk. When you've finished reading the whole file, you want to print the number of times the letter X appears in the file. Here's how that might work:

Listing 2.11 Reading a file from disk

```
var fs = require("fs");  #A

var options = { encoding: "utf-8" };                       #B
fs.readFile("myfile.txt", options, function(err, data) {   #B
  if (err) {                                  #C
    console.error("Error reading file!");     #C
    return;                                   #C
  }                                           #C

  console.log(data.match(/x/gi).length + " letter X's");    #D
});
```

#A Require Node's file system module like we've seen before.

#B Read myfile.txt (and interpret the bytes as UTF-8).
#C Handle any errors encountered when reading the file.
#D Print the number of X's by using a regular expression.

Let's step through this code.

First, we require Node's built-in file system module. This has tons of functions for various tasks on the file system, most commonly reading and writing files. In this example, we'll use its `readFile` method.

Next, we set some options that we'll pass into `fs.readFile`. We call it with the filename (myfile.txt), the options we just created, and a callback. When the file has been read off of disk, Node will jump into your callback.

Most callbacks in Node are called with an error as their first argument. If all goes well, the `err` argument will be `null`. But if things don't go so well (maybe the file didn't exist or was corrupted), the `err` argument will have some value. It's a best practice to handle those errors. Sometimes the errors don't completely halt your program and you can continue on, but you often handle the error and then break out of the callback by throwing an error or returning.

This is a common Node practice and you'll see it almost everywhere you see a callback.

Finally, once we know we don't have any errors, we print out the number of X's in the file! We use a little regular expression trick to do this.

Okay, pop quiz: what happens if we added a `console.log` statement at the very end of this file, like this?

Listing 2.12 Adding a console.log after the asynchronous operations

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

vra options = { encoding: "utf-8" };
fs.readFile("myfile.txt", options, function(err, data) {
  // ...
});

console.log("Hello world!");  #A
```

#A Note the added line here.

Because this file reading operation is asynchronous, we'll see "Hello world" before we see any results from the file. This is because the external resource—the file system—hasn't gotten back to us yet.

This is how Node's asynchronous model can be super helpful. While an external resource is handling something, we can continue onto other code. In the context of web applications, that means that we can parse many more requests at once.

> **NOTE** There's a fantastic video on how callbacks and the event loop work in JavaScript (both in Node and in the browsers). If you're interested in understanding the nitty-gritty details, I cannot recommend Philip Roberts's "What the heck is the event loop anyway?" at https://www.youtube.com/watch?v=8aGhZQkoFbQ enough.

2.4 Building a web server with Node: the HTTP Module

Understanding the big concepts in Node will help you understand the built-in module that's most important to Express: its HTTP module. It's the module that makes it possible to develop web servers with Node, and it's what Express is built on.

Node's `http` module has various features (making requests to other servers, for instance) but we'll use its HTTP server component: a function called `http.createServer`. This function takes a callback that's called every time a request comes into your server, and it returns a server object. Here's a very simple server that sends "hello world" with every request (which you can save into `myserver.js` if you'd like to run it):

Listing 2.13 A simple "hello world" web server with Node

```
var http = require("http");            #A

function requestHandler(request, response) {                #B
  console.log("In comes a request to: " + request.url);   #B
  response.end("Hello, world!");                          #B
}                                                         #B

var server = http.createServer(requestHandler);  #C

server.listen(3000);  #D
```

#A Require Node's built-in HTTP module.
#B Define a function that'll handle incoming HTTP requests.
#C Create a server that uses your function to handle requests.
#D Start the server listening on port 3000.

This code is split up into four chunks above.

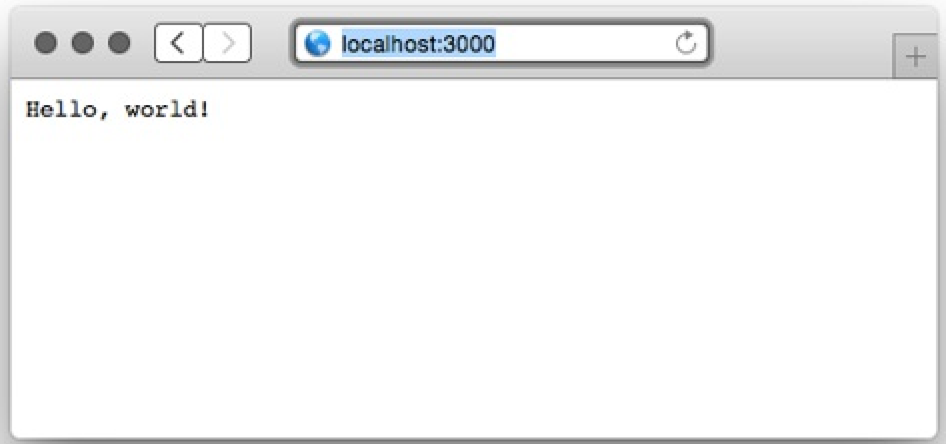
The first chunk simply requires the HTTP module and puts it into a variable called `http`. We saw this above with the URL module and the file system module—this is exactly the same.

Next, we define a request handler function. *Nearly every bit of code in this book* is either a request handler function or a way to call one, so listen up! These request handler functions take two arguments: an object that represents the request (often shortened to `req`) and an object that represents the response (often shortened to `res`). The request object has things like the URL that the browser requested (did they request the homepage or the about page?), or the type of browser visiting your page (called the user-agent), or things like that. You call methods on the response object and Node will package up the bytes and send them across the internet.

The rest of the code points Node's built-in HTTP server at the request handler function and starts it on port 3000.

> **WHAT ABOUT HTTPS?** Node also comes with a module called https. It's very similar to the http module, and creating a web server with it is almost identical. If you decide to swap things out later, it should take less than 2 minutes if you know how to do HTTPS. If you don't know much about HTTPS, don't worry about this.

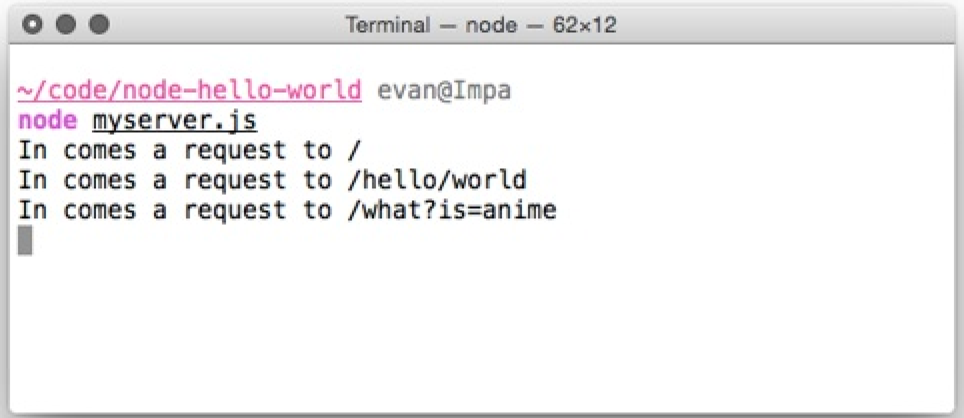
You can try saving the code above into a file called `myserver.js`. To run the server, type `node myserver.js` (or just `node myserver`). Now, if you visit http://localhost:3000 in your browser, you'll see something like Figure 2.3.



```
Hello, world!
```

Figure 2.3 A simple "hello world" app.

You'll also notice that something appears in your console every time you visit a page. Try visiting a few other URLs: http://localhost:3000/ or http://localhost:3000/hello/world or http://loca is=anime. The output will change in the console, but your server won't do anything different and will always just say "Hello, world!" Figure 2.4 shows what your console might look like:



```
~/code/node-hello-world evan@Impa
node myserver.js
In comes a request to /
In comes a request to /hello/world
In comes a request to /what?is=anime
```

Figure 2.4 The console from your "hello world" app might look something like this.

Notice that the request URL doesn't include "localhost:3000" anywhere. That might be a little unintuitive, but this is pretty helpful, as it turns out. This allows you to deploy your application anywhere, from your local server to your favorite .com address. It'll work without any changes!

One could imagine parsing the request URL. You could imagine doing something like this:

Listing 2.14 Parsing the request URL with a request handler function

```
// …

function requestHandler(req, res) {
  if (req.url === "/") {
    res.end("Welcome to the homepage!");
  }
  else if (req.url === "/about") {
    res.end("Welcome to the about page!");
  }
  else {
    res.end("Error! File not found.");
  }
}

// …
```

You could imagine building your entire site in this one request handler function. For very small sites, this might be easy, but you could imagine this function getting huge and unwieldy pretty quickly. You might want a framework to help you clean up this HTTP server…things could get messy!

That's where Express will come in.

2.5 Summary

In this chapter, we've learned:

- How to install Node.js
- How to use its module system by using `require` and `module.exports`

- `package.json` for describing our project's metadata with things like name, author, verison, and more
- Using npm to install packages with `npm install` (and a couple of other tricks like `init`)
- The asynchronous, evented I/O concepts of Node—you can sort of do two things at once
- How to use Node's built-in HTTP module to build a simple web server

3 Foundations of Express

As we saw in the previous chapter, Node.js comes with a number of built-in modules, one of which is called `http`. Node's HTTP module allows you to build an HTTP server that responds to HTTP requests from browsers (and more). In short, the HTTP module lets you build websites with Node!

While you can build full web servers with nothing but Node's built-in HTTP module, you might not want to. As we discussed in Chapter 1 and saw in Chapter 2, the API exposed by the HTTP module is pretty minimal and doesn't do a lot of heavy lifting for you.

That's where Express comes in: it's a helpful third-party module (that is, not bundled with Node.js). When you get right down to it, Express is really just an abstraction layer on top of Node's built-in HTTP server. You could, in theory, write everything with "vanilla" Node and never touch Express. But as we'll see, Express smooths out a lot of the difficult parts and says "don't worry, you don't need to deal with this ugly part. I'll handle this!" In other words, it's magic!

In this chapter, we'll spring off of our Node knowledge and make an effort to really understand Express. We'll talk about its relationship to bare Node, the concepts of middleware and routing, and learn the other nice features Express gives us. In future chapters, we'll go more in depth; this chapter will give a code-heavy overview of the framework.

At a high level, Express really just gives us four major features, which we'll be learning about in this chapter:

1. In contrast to "vanilla" Node, where your requests only flow through one function, Express has a "middleware stack", which is effectively an array of functions.
2. Routing is a lot like middleware, but the functions are only called when you visit a specific URL with a specific HTTP method. For example, you could only run a request handler when the browser visits `yourwebsite.com/about`.

3. Express also extends request and response with a bunch of extra methods and properties for developer convenience.
4. Views allow you to dynamically render HTML. This both allows you to change the HTML on the fly and allows you to write the HTML in other languages.

We'll build a simple guestbook in this chapter to get a feel for these four features.

3.1 Middleware

One of Express's biggest features is called "middleware". Middleware is very similar to the request handlers we saw in "vanilla" Node (accepting a request and sending back a response), but middleware has one important difference: rather than having just one handler, middleware allows for *many* to happen in sequence.

Middleware has a variety of applications, which we'll explore. For example, one middleware could log all requests, and then continue onto another middleware that sets special HTTP headers for every request, which could then continue further. While we could do this with one large request handler, we'll see that it's often preferable to decompose these disparate tasks into separate middleware functions. If this is confusing now, don't worry—we'll have some helpful diagrams and get into some concrete examples.

> **ANALOGS IN OTHER FRAMEWORKS** Middleware isn't unique to Express; it's present in a lot of other places in different forms. Middleware is present in other web application frameworks like Python's Django or PHP's Laravel. Ruby web applications also have this concept, often called "Rack middleware". This concept may not be radically new to you, though Express has its own flavor of middleware.

Let's start rewriting out "Hello, World" application using Express's middleware feature. We'll see that it has far fewer lines of code, which can help us speed up development time and reduce the number of potential bugs.

3.1.1 "Hello, World" with Express

Let's set up a new Express project. Make a new directory and put a file called `package.json` inside. Recall that `package.json` is how we store information about a Node project. It lists simple data like the project's name and author, and also contains information about its dependencies.

Start with a skeleton `package.json`:

Listing 3.1 A bare-bones package.json

```
{
  "name": "hello-world",
  "author": "Your Name Here!",
  "private": true,
  "dependencies": {}
}
```

...and then install Express and save it to your `package.json`:

```
npm install express –save
```

Running this command will find Express in the directory of third-party Node packages and fetch the latest version. It will put it in a folder called `node_modules/`. Adding `--save` to the installation command will save it under the`dependencies` key of `package.json`. After running this command, your `package.json` will look something like this:

Listing 3.2 package.json after installing Express with the --save flag

```
{
  "name": "hello-world",
  "author": "Your Name Here!",
  "private": true,
  "dependencies": {
    "express": "^4.10.5"
  }
}
```

Alright, now we're ready. Save this file into app.js:

Listing 3.3 "Hello, World" with Express

```
var express = require("express");  #A
var http = require("http");
var app = express();   #B

app.use(function(request, response) {  #C
  response.writeHead(200, { "Content-Type": "text/plain" });      #C
  response.end("Hello, World!");  #C
});  #C

http.createServer(app).listen(3000);  #D
```

#A There's a new kid on the block: the Express module. We require it just like we require the http module.
#B To start a new Express application, we simply call the express function.
#C This function is called "middleware". As we'll see, it looks an awful lot like the request handlers from before.
#D Start the server up!

Now let's step through this.

First, we require Express. We then require Node's HTTP module just like we did before. We're ready.

Then we make a variable called app like we did before, but instead of creating the server, we call `express()`, which *returns a request handler function*. This is important: it means that we can pass the result into `http.createServer` just like before.

Remember the request handler we had in the previous chapter, with "vanilla" Node? It looked like this:

```
var app = http.createServer(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Hello, world!");
});
```

We have a very similar function in this example (in fact, I copy-pasted it). It's also passed a request and a response object, and we interact with them in the same way.

Next we create the server and start listening. Recall that `http.createServer` took a function before, so guess what—`app` is just a function. It's an Express-made request handler that starts going through all the middleware until the end. At the end of the day, it's just a request handler function like before.

> **NOTE** You will see people using `app.listen(3000)`, which just defers to `http.createServer`. It's just shorthand, like how we will shorten `request` to `req` and `response` to `res` in following chapters.

3.1.2 How middleware works at a high level

In Node, everything goes through one big function. To resurface an example from Chapter 2, it looks like this:

Listing 3.4 A Node request handler function

```
function requestHandler(request, response) {
  console.log("In comes a request to: " + request.url);
  response.end("Hello, world!");
}
```

In a world without middleware, we find ourselves having one master request function that handles everything. If we were to draw the flow of our application, it might look like Figure 3.1.

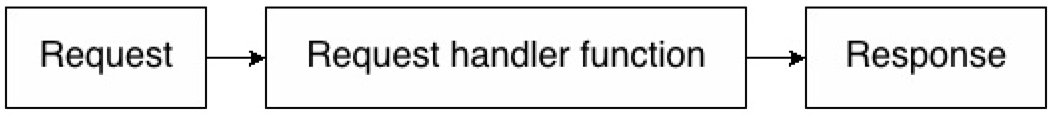


Figure 3.1 A request without middleware

Every request goes through just one request handler function, which eventually generates the response. That's not to say that the master handler function can't call other functions, but at the end of the day, the master function responds to every request.

With middleware, rather than having your request pass through one you write function, it passes through an *array* of functions you write called a "middleware stack". It might look like Figure 3.2.

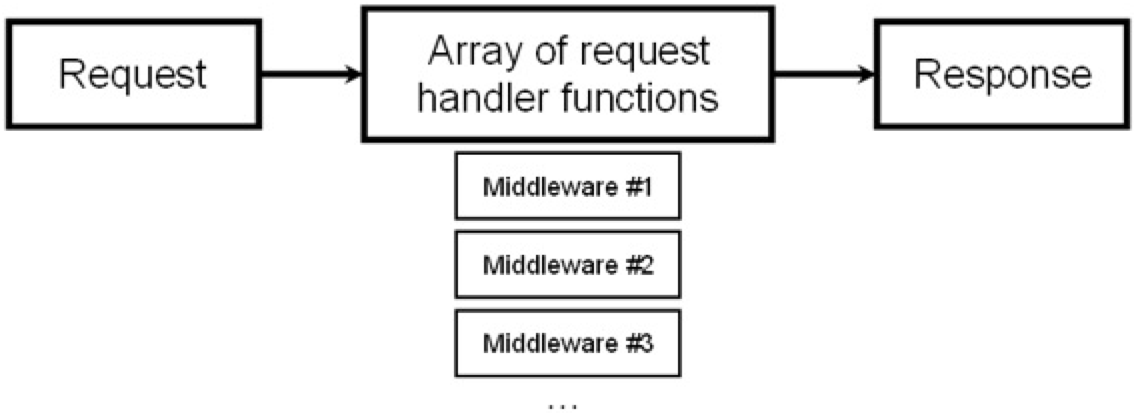


Figure 3.2 A request with middleware.

Okay, so Express lets you execute an array of functions instead of just one. What might some of these functions be? And why might we want this?

Let's resurface an example from Chapter 1: an application that authenticates users. If they're authenticated, it shows them some secret information. All the while, our server is logging every request that comes into our server, authenticated or not.

This app might have three middleware functions: one that does logging, one that does authentication, and one that responds with secret information. The logging middleware will log *every* request and continue onto the next middleware; the authentication middleware will only continue if the user is authorized; the final middleware will always respond and it won't continue on because nothing follows it.

There are two possible ways a request could flow through this simple app; an illustration of two possible options is shown in Figure 3.3.

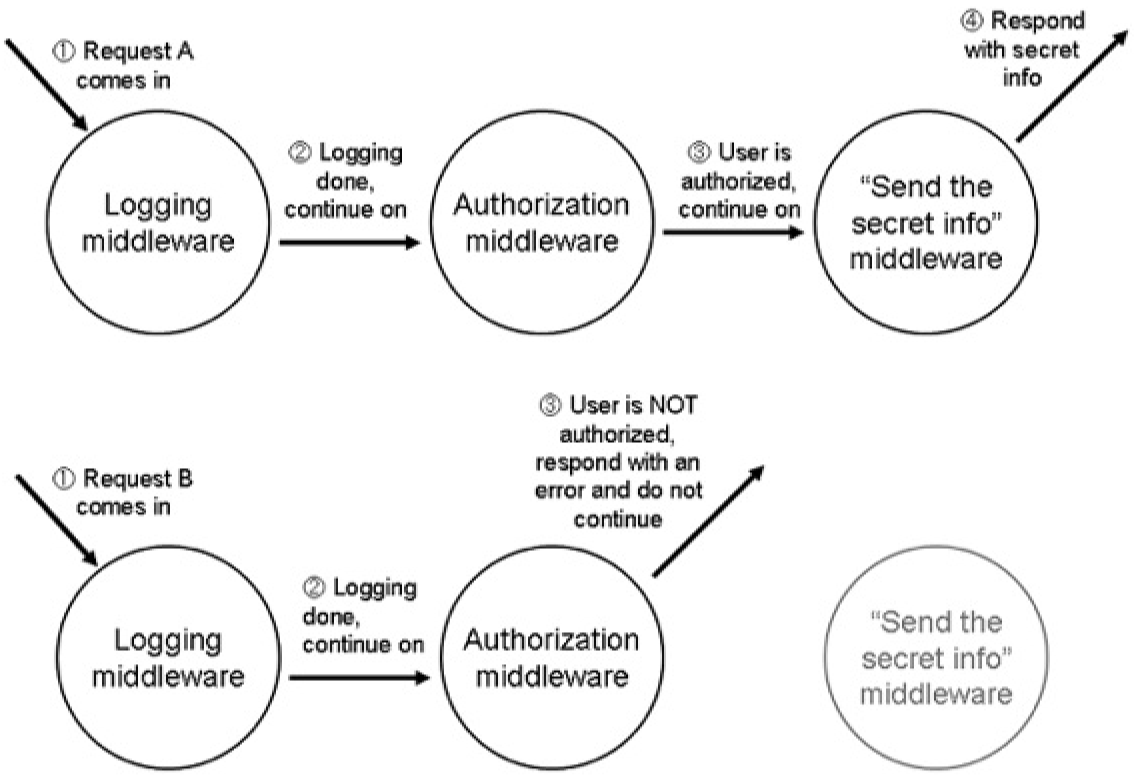

① Request A comes in
Logging middleware
② Logging done, continue on
Authorization middleware
③ User is authorized, continue on
"Send the secret info" middleware
④ Respond with secret info
① Request B comes in
Logging middleware
② Logging done, continue on
Authorization middleware
③ User is NOT authorized, respond with an error and do not continue
"Send the secret info" middleware

Figure 3.3 Two requests flowing through middleware functions. See that middleware sometimes continues on, but sometimes responds to requests.

Each middleware function can modify the request or the response, but it doesn't always have to. Eventually, *some* middleware should respond to the request. It could be the first one, it could be the last. If none of them respond, then the server will hang and the browser will sit alone, without a response.

This is powerful because we can split our application into many small parts, rather than having one behemoth. They become easier to compose and reorder, and it's also easy to pull in third-party middleware.

We'll see some examples that will (hopefully!) make all of this more clear.

3.1.3 Middleware code that's passive

Middleware can affect the response, but it doesn't have to. For example, the logging middleware from the previous section doesn't need to send different data—it just needs to log the request and move on.

Let's start by building a completely useless middleware and then moving on from there. Here's what an empty middleware function looks like:

Listing 3.5 An empty middleware that does nothing

```
function myFunMiddleware(request, response, next) {

  ... #A

  next(); #B
}
```

#A Do stuff with the request and/or response.
#B When we're all done, call next() to defer to the next middleware in the chain.

When we start a server, we start at the topmost middleware and work our way to the bottom. So if we wanted to add simple logging to our app, we could do it!

Listing 3.6 Logging middleware

```
var express = require("express");
var http = require("http");
var app = express();

app.use(function(request, response, next) { #A
  console.log("In comes a " + request.method + " to " + request.url);
  next();
});

app.use(function(request, response) {   #B
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Hello, World!");
});

http.createServer(app).listen(3000);
```

#A This is the logging middleware, which will log the request to the console and then advance to the next middleware.
#B This sends the actual response.

Run this app and visit `http://localhost:3000`. In the console, you'll see that your server is logging your requests (refresh to see). You'll also see your "Hello, World!" in the browser.

It's important to note that anything that works in the vanilla Node.js server also works in middleware. For example, you can inspect `request.method` in a vanilla Node web server, without Express. Express doesn't get rid of it--it's right there like it was before. If you want to set the `statusCode` of the response, you can do that too. Express adds some more things to these objects, but it doesn't *remove* anything.

The above example shows middleware that doesn't change the request or the response—it logs the request and always continues. While this kind of middleware can be useful, middleware can also change the request or response objects.

3.1.4 Middleware code that changes the request and response

Not all middleware should be passive, though—the rest of the middleware from our example doesn't work that way; they'll actually need to change the response.

Let's try writing the authentication middleware that we mentioned before. Let's choose a weird authentication scheme for simplicity: you're only authenticated if you visit on an even-numbered minute of the hour (which would be 12:00, 12:02, 12:04, 12:06, and so on). Recall that we can use the modulo operator (`%`) to help determine whether a number is divisible by another.

We add this middleware to our application in Listing 3.7:

Listing 3.7 Adding fake authentication middleware

```
app.use(function(request, response, next) {   #A
  console.log("In comes a " + request.method + " to " + request.url);
  next();
});

app.use(function(request, response, next) {
  var minute = (new Date()).getMinutes();
  if ((minute % 2) === 0) {
    next();    #B
  } else {
    response.statusCode = 403;         #C
    response.end("Not authorized.");  #C
  }
});

app.use(function(request, response) {
  response.end('Secret info: the password is "swordfish"!');  #D
});
```

#A This is the logging middleware, just as before.
#B If you're visiting at the first minute of the hour, call next() to continue onto the "send secret info" middleware.
#C If you're not authorized, send a status code of 403 ("Not authorized") and respond to the user. Notice that we don't call next() to continue on.
#D Send the secret information!

When a request comes in, it will always go through the middleware in the same order you `use` them. First, it will start with the logging middleware. Then, if you're visiting in an even-numbered minute, you'll continue onto the next middleware and see the secret information. But if you're visiting at any of the other minutes of the hour, you'll stop and never continue on.

3.1.5 Third-party middleware libraries

Like many parts of programming, it's often the case that someone else has done what you're trying to do. You can write your own middleware, but it's common to find that the functionality you want is already in somebody else's middleware.

Let's look at a couple of examples of helpful third-party middleware.

MORGAN: LOGGING MIDDLEWARE

Let's remove our logger and use Morgan, a nice logger for Express that has far more features. Loggers are pretty helpful for a number of reasons. First of all, they're one way to see what your users are doing. This isn't the best way to do things like marketing analytics, but it's really useful when your app crashes for a user and you're not sure why. I also find it really useful when developing—you can see when a request comes into your server. If something is wrong, you can use Morgan's logging as a sanity check.

Run `npm install morgan --save` and give this a try (saving it into app.js again):

Listing 3.8 Using Morgan for logging (in app.js)

```
var express = require("express");
var logger = require("morgan");
var http = require("http");

var app = express();

app.use(logger("short")); #A

app.use(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Hello, World!");
});

http.createServer(app).listen(3000);
```

#A Fun fact: logger("short") returns a function.

Visit `http://localhost:3000` and you'll see some logging! Thanks, Morgan.

EXPRESS'S STATIC MIDDLEWARE

There's more middleware out there than just Morgan.

It's very common for web applications to need to send static files over the wire. This is things like images or CSS or HTML—content that isn't dynamic.

`express.static` ships with Express, and helps you serve static files. The simple act of sending files turns out to be a lot of work, because there are a lot of edge cases and performance considerations to think about. Express to the rescue!

Let's say we want to serve files out of a directory called "public". Here's how we might do that with Express's static middleware:

Listing 3.9 Using express.static (in app.js)

```
var express = require("express");
var path = require("path");
var http = require("http");

var app = express();

var publicPath = path.resolve(__dirname, "public");  #A
app.use(express.static(publicPath));  #B

app.use(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Looks like you didn't find a static file.");
});

http.createServer(app).listen(3000);
```

#A Set up the public path, using Node's path module.
#B Send static files from the publicPath directory.

Now, any file in the public directory will be shown. We can put anything in there that we please and the server will send it. If no matching file exists in the `public` folder, it'll go onto the next middleware, and say "Hello, World!". If a matching file is found, `express.static` will send it off and stop the middleware chain.

Why use path.resolve?

What's all that business about path.resolve? Why can't we just say /public? The short answer is that we could, but it's not cross-platform.

On Mac and Linux, we want this directory:

```
/public
```

But on Windows, we want this directory:

```
\public
```

Node's built-in path module will make sure that things run smoothly on Windows, Mac, and Linux.

FINDING MORE MIDDLEWARE

I've shown Morgan and Express's static middleware, but there's a lot more. Here are a few other helpful ones:

- connect-ratelimit lets you throttle connections to a certain number of requests per hour. If someone is sending lots of requests to your server, you can start giving them errors to stop them from bringing your site down.
- helmet helps you add HTTP headers to make your app safer against certain kinds of attacks. We'll explore it in later chapters. (I'm a contributor to Helmet, so I'd definitely recommend it!)
- cookie-parser parses browser cookies.
- response-time sends the X-Response-Time header so you can debug the performance of your application.

We'll explore many of these middleware options further in the next chapter.

If you're looking for more middleware, you'll have luck searching for "Express middleware", but you should also search for "Connect middleware" too. There's another framework called Connect that's like Express but only does middleware. Connect middleware is compatible with Express, so if the "Express middleware" search isn't fruitful, try searching for Connect middleware.

3.2 Routing

Routing is a way to map requests to specific handlers depending on their URL and HTTP verb. You could imagine having a homepage and an about page and a 404 page. Routing can do all of this. I think is better explained with code than with English:

Listing 3.10 Express routing example

```
var express = require("express");
var path = require("path");
var http = require("http");

var app = express();

var publicPath = path.resolve(__dirname, "public");  #A
app.use(express.static(publicPath));  #A

app.get("/", function(request, response) {    #B
  response.end("Welcome to my homepage!");
});

app.get("/about", function(request, response) {    #C
  response.end("Welcome to the about page!");
});

app.get("/weather", function(request, response) {    #D
  response.end("The current weather is NICE.");
});

app.use(function(request, response) {    #E
  response.statusCode = 404;
  response.end("404!");
});

http.createServer(app).listen(3000);
```

#A This sets up static middleware like we've seen before. Every request will go through this middleware, and if no static file is found, it will continue onto the routes below.
#B This request handler is called when a request to the root is called. In this example's case, this handler is called when you visit http://localhost:3000.
#C This request handler is called when a request to /about (http://localhost:3000/about in this case) comes in.
#D This request handler is called when a request to /weather (http://localhost:3000/weather in this case) comes in.
#E If we didn't hit the static file middleware or any of the routes above, then we've tried everything and we'll wind up here. This will happen when you visit an unknown URL, like /edward_cullen or /delicious_foods/burrito.jpg.

After the basic requires, we add our static file middleware (just like we've seen before). This will serve any files in a folder called `public`.

The three calls to `app.get` are Express's magical routing system. They could also be `app.post`, which respond to POST requests, or PUT, or any of the HTTP verbs. (We'll talk more about these other HTTP verbs in later chapters.) The first argument is a path, like `/about` or `/weather` or simply `/`, the site's root. The second argument is a request handler function similar to what we've seen before in the middleware section.

They're the same request handler functions we've seen before. They work just like middleware; it's just a matter of *when* they're called.

These routes can get smarter. In addition to matching fixed routes, they can also match more complex ones (imagine a regular expression or more complicated parsing).

Listing 3.11 Grabbing data from routes

```
app.get("/hello/:who", function(request, response) {    #A
  response.end("Hello, " + request.params.who + ".");  #B
  // Fun fact: this has some security issues, which we'll get to!
});
```

#A This specifies that the "hello" part of the route is fixed, but the string afterward can vary.
#B req.params has a property called "who". It's no coincidence that this was also the name specified in the route above. Express will pull the value from the incoming URL and set it to the name you specify.

Restart your server and visit `localhost:3000/hello/earth` for the following message:

Hello, earth.

Note that this won't work if you add a slash—for example, `localhost:3000/hello/entire/earth` will give a 404 error.

It's likely that you've seen this sort of behavior all over the internet. For example, you've likely seen websites where you can visit a URL for a specific user. For example, if your username were ExpressSuperHero, the URL for your user page might look something like this:

```
https://mywebsite.com/users/ExpressSuperHero
```

Express allows us to do something like this. Rather than defining a route for *every single possible username* (or article, or photo, or whatever), you define one route that matches all of them.

The docs also show an example that uses regular expressions to do even more complex matching, and you can do lots of other stuff with this routing. For a conceptual understanding, I've said enough. We'll explore this in far more detail in Chapter 5.

But it gets cooler.

3.3 Extending request and response

Express augments the request and response objects that you're passed in every request handler. The old stuff is still there, but Express adds some new stuff too! The API docs (at http://expressjs.com/api.html) explain everything, but let's look at a couple of examples.

One nicety Express offers is a redirect method. Listing 3.12 shows how the redirect method might work:

Listing 3.12 Using redirect

```
response.redirect("/hello/world");
response.redirect("http://expressjs.com");
```

If we were just using Node, `response` would have no method called `redirect`; Express adds it to the response object for us. You *can* do this in vanilla Node, but it's a lot more code.

Express also adds methods like `sendFile` which lets you just send a whole file:

Listing 3.13 sendFile example

```
response.sendFile("/path/to/cool_song.mp3");
```

Once again, the `sendFile` method isn't available in vanilla Node; Express adds it for us. And just like the redirect example above, you *can* do this in vanilla Node, but it's a lot more code.

It's not just the response object that gets conveniences—the request object gets a number of other cool properties and methods, like `request.ip` to get the IP address or the `request.get` method to get incoming HTTP headers.

Let's use some of these things to build some middleware that blocks an evil IP address. Express makes this pretty easy:

Listing 3.14 Blacklisting an IP

```
var express = require("express");
var app = express();

var EVIL_IP = "123.45.67.89";

app.use(function(request, response, next) {
  if (request.ip === EVIL_IP) {
    response.status(401).send("Not allowed!");
  } else {
    next();
  }
});

// ... the rest of your app ...
```

Notice that we're using `req.ip`, a function called `res.status()`, and `res.send()`. None of these are built into vanilla Node—they're all extensions added by Express.

Conceptually, there's not much to know here, other than the fact that Express extends the request and response.

We've looked at a few niceties in this chapter, but I don't want to give you the full laundry list here. For every nice feature that Express gives you, check out its API documentation at http://expressjs.com/4x/api.html.

3.4 Views

Websites are built with HTML. They've been built that way for a long, long time. While single-page apps are en vogue (and totally possible with Express), it's often the case that you want the server to dynamically generate HTML. You might want to serve HTML that greets the currently logged-in user, or maybe you want to dynamically generate a data table.

There are a number of different view engines out there. There's EJS (which stands for "embedded JavaScript"), Handlebars, Jade, and more. There are even ports of templating languages from other programming worlds, like Swig and HAML. All of these have one thing in common: at the end of the day, they spit out HTML.

For the rest of these examples, we'll use EJS. I chose EJS because it's a popular option made by the people who created Express. I hope you'll like it, but if you don't, there are plenty of alternatives which we'll discuss in Chapter 7.

Here's what it looks like to set up views:

Listing 3.15 Setting up views with Express

```
var express = require("express");
var path = require("path");

var app = express();

app.set("views", path.resolve(__dirname, "views")); #A
app.set("view engine", "ejs");  #B
```

#A This tells Express that your views will be in a folder called views. We could put it in another path, but "views" is a common name.
#B This tells Express that you're going to use the EJS templating engine.

We'll add more to this file in a moment.

The first block is the same as always: require what we need to. Then we say "our views are in a folder called views". After that, we say "use EJS". EJS (documentation at https://github.com/tj/ejs) is a templating language that compiles to HTML. Make sure to install it with `npm install ejs --save`.

Now, we've set up these views on the Express side. How do we use them? What is this EJS business?

Let's start by making a file called `index.ejs` and put it into a directory called `views`. It might look like this:

Listing 3.16 A simple EJS file

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello, world</title>
  </head>
<body>
  <%= message %>
</body>
</html>
```

This should look exactly like HTML to you, but for the one weird bit inside the body tag. EJS is a superset of HTML, so everything that's valid HTML is valid EJS. But EJS also adds a few new features, like variable interpolation. `<%= message %>` will interpolate a variable called message, which we'll pass when we render the view from Express. Here's what that looks like:

Listing 3.17 Rendering a view from Express

```
app.get("/", function(request, response) {
  response.render("index", {
    message: "Hey everyone! This is my webpage."
  });
});
```

Express adds a method to response, called `render`. It basically looks at the view engine and views directory (which we defined earlier) and renders `index.ejs` with the variables you pass in.

The code in Listing 3.18 would render the following HTML:

Listing 3.18 A simple EJS file, rendered

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello, world</title>
  </head>
<body>
  Hey everyone! This is my webpage. #A
</body>
</html>
```

#A Notice that this is the variable we specified in the render method above.

EJS is a popular solution to views, but there are a number of other options.

We'll explore other options in later chapters.

3.5 Example: putting it all together in a guestbook

If you're like me, you saw the internet in its early days; awkward animated GIFs, crufty code, and Times New Roman on every page. In this chapter, we'll resurrect one component from that bygone era: the guestbook. A guestbook is pretty simple: users can write new entries in the online guestbook, and they can browse others' entries.

Let's use all that we've learned to build a more real application for this guestbook. It turns out that all of these things will come in handy! Our site will have two pages:

1. A homepage that lists all of the previously-posted guestbook entries
2. A page with an "add new entry" form

That's it! Before we start, we have to get set up. Ready?

3.5.1 Getting set up

Let's start a new project. Make a new folder, and inside, make a file called `package.json`. It should look something like this:

Listing 3.19 package.json for the guestbook

```
{
  "name": "express-guestbook",
  "private": true,
  "scripts": {
    "start": "node app"   #A
  }
}
```

#A Typing "npm start" in your terminal will run "node app", which will start your app.

You can add other fields (like author or version), but for this example, we don't need much. Now, let's install our dependencies like we've done before and save them into `package.json`:

```
npm install express morgan body-parser ejs --save
```

These modules should look familiar to you, except for `body-parser`. Our app will need to post new guestbook entries in HTTP POST requests, so we'll need to parse the body of the POST; that's where body will come in.

Check to make sure that Express, Morgan, body-parser, and EJS have been saved into `package.json`. If they haven't, make sure you've added the `--save` flag.

3.5.2 The main app code

Now we've installed all of our dependencies, create `app.js` and put the following app inside:

Listing 3.20 The Express guestbook, in app.js

```
var http = require("http");                #A
var path = require("path");                #A
var express = require("express");          #A
var logger = require("morgan");            #A
var bodyParser = require("body-parser");  #A

var app = express();  #B

app.set("views", path.resolve(__dirname, "views"));  #C
app.set("view engine", "ejs");                       #C

var entries = [];              #D
app.locals.entries = entries;  #E

app.use(logger("dev"));  #F

app.use(bodyParser.urlencoded({ extended: false })); #G

app.get("/", function(request, response) {  #H
  response.render("index");             #H
});                                #H

app.get("/new-entry", function(request, response) { #I
    response.render("new-entry");               #I
});                                          #I

app.post("/new-entry", function(request, response) {  #J
  if (!request.body.title || !request.body.content) {                      #K
    response.status(400).send("Entries must have a title and content."); #K
    return;                                                         #K
  }                                                                 #K
  entries.push({               #L
    title: request.body.title,     #L
    content: request.body.content, #L
    published: new Date()      #L
  });                          #L
  response.redirect("/");  #M
});

app.use(function(request, response) {     #N
  response.status(404).render("404"); #N
});                              #N

http.createServer(app).listen(3000, function() {     #O
  console.log("Guestbook app started on port 3000.");
});
```

#A First, we require all of the modules we need, just like before.
#B Next, we make an Express app, just like we've done before.
#C The first line tells Express that the views are in a folder called views, and the next line says that the views will use the EJS engine.

#D Create a "global" array to store all of our entries.
#E Make this entries array available in all views.
#F Use Morgan to log every request.
#G This middleware will populate a variable called req.body if the user is submitting a form. (The extended option is required and we choose false for slight security benefits. We'll discuss the reasons why in great detail in Chapter 10.)
#H When visiting the site root, render the homepage (which will be in a views/index.ejs).
#I Render the "new entry" page (at views/index.ejs) when GETting the URL.
#J Define a route handler when we POST to the "new entry" URL. Note that this is the same URL but a different HTTP verb.
#K If the user submits the form with no title or content (which we read out of req.body), respond with a 400 "bad request" error.
#L Add a new entry to the list of entries, with the title, body, and the time published.
#M Finally, redirect back to the homepage to see your new entry.
#N None of the other request handlers happened, so that means we're trying to request an unknown resource. Render a 404 page.
#O Start the server on port 3000!

3.5.3 Creating the views

We've referenced a few views here, so let's fill those in. Create a folder called views, and then create the header in `views/header.ejs`:

Listing 3.21 header.ejs

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Express Guestbook</title>
<link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">  #A
</head>
<body class="container">
  <h1>
    Express Guestbook
    <a href="/new-entry" class="btn btn-primary pull-right">
      Write in the guestbook
    </a>
  </h1>
```

#A This code loads Twitter's Bootstrap CSS from the Bootstrap CDN, an external server that hosts Bootstrap for your convenience.

Notice that we use Twitter Bootstrap for styling, but you could easily replace it with your own CSS. The most important part is that this is the header; this HTML will appear at the top of every page.

> **NOTE** In short, Bootstrap is a bunch of CSS and JavaScript that provides a bunch of default styling. You can absolutely write navbars and buttons and header CSS yourself, but Bootstrap helps us get up and running quickly. You can find out more at http://getbootstrap.com/.

Next, create the simple footer in `views/footer.ejs`, which will appear at the bottom of every page:

Listing 3.22 footer.ejs

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

Now that we've defined the common header and footer, let's define the three views: the homepage, the "add a new entry" page, and the 404 page.

Save the following into `views/index.ejs`:

Listing 3.23 index.ejs

```
<% include header %>
<% if (entries.length) { %>
  <% entries.forEach(function(entry) { %>
    <div class="panel panel-default">
      <div class="panel-heading">
        <div class="text-muted pull-right">
          <%= entry.published %>
        </div>
        <%= entry.title %>
      </div>
      <div class="panel-body">
        <%= entry.content %>
      </div>
    </div>
  <% }) %>
<% } else { %>
  No entries! <a href="/new-entry">Add one!</a>
<% } %>
<% include footer %>
```

...the following into `views/new-entry.ejs`...

Listing 3.24 new-entry.ejs

```
<% include header %>

<h2>Write a new entry</h2>

<form method="post" role="form">
  <div class="form-group">
    <label for="title">Title</label>
    <input type="text" class="form-control" id="title"
    [CA]name="title" placeholder="Entry title" required>
  </div>
  <div class="form-group">
    <label for="content">Entry text</label>
    <textarea class="form-control" id="content" name="content"
    [CA]placeholder="Love Express! It's a great tool for
    [CA]building websites." rows="3" required></textarea>
  </div>
  <div class="form-group">
    <input type="submit" value="Post entry" class="btn btn-primary">
  </div>
</form>

<% include footer %>
```

...and finally, the following into `views/404.ejs`:

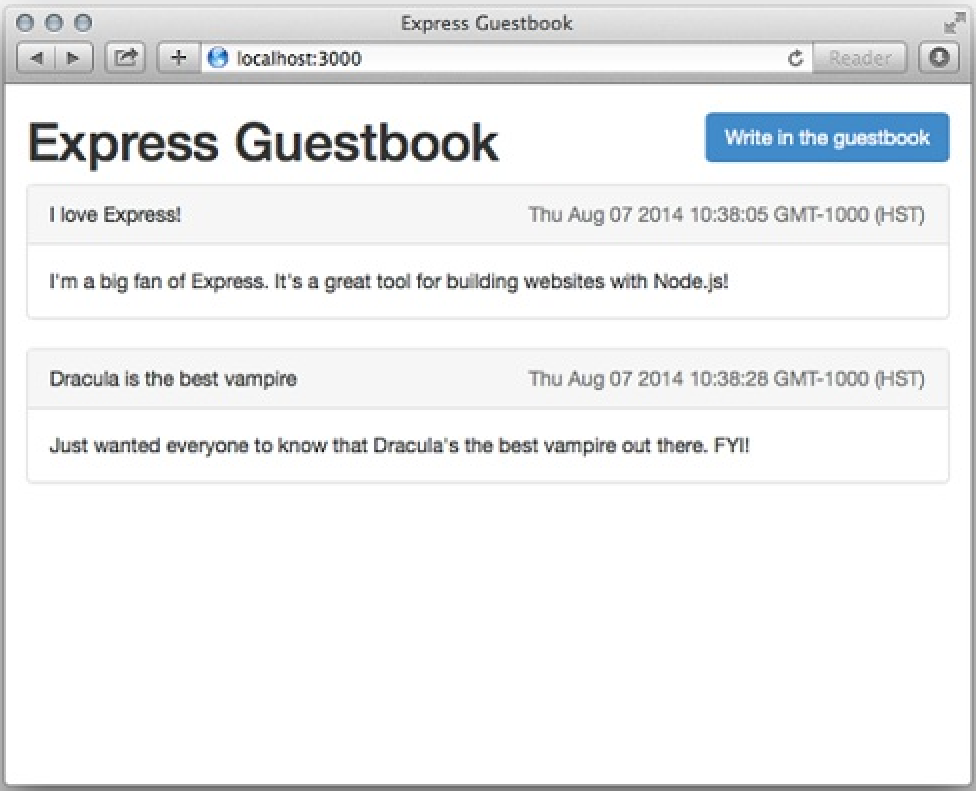
Listing 3.25 404.ejs

```
<% include header %>
<h2>404! Page not found.</h2>
<% include footer %>
```

And that's all your views!

3.5.4 Start it up!

Now, `npm start` up your app and visit `http://localhost:3000`, and see our guestbook.



Express Guestbook

I love Express! Thu Aug 07 2014 10:38:05 GMT-1000 (HST)

I'm a big fan of Express. It's a great tool for building websites with Node.js!

Dracula is the best vampire Thu Aug 07 2014 10:38:28 GMT-1000 (HST)

Just wanted everyone to know that Dracula's the best vampire out there. FYI!

Figure 3.4 The guestbook homepage

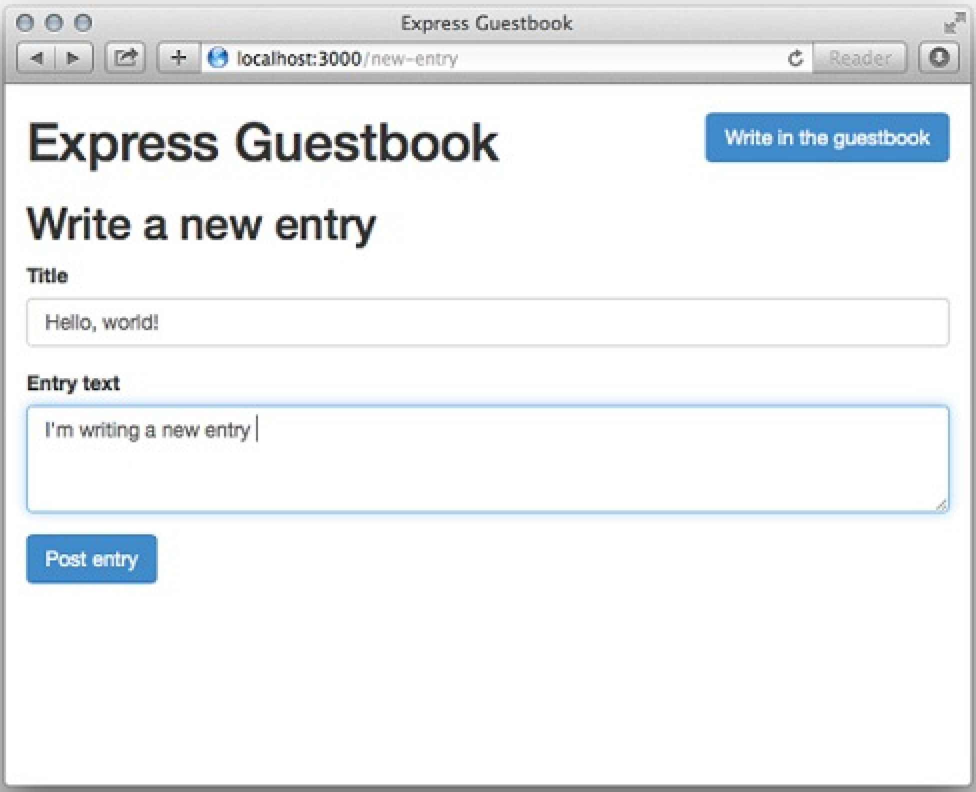


Figure 3.5 The guestbook homepage

Look at that! What a beautiful little guestbook. It reminds me of the 1990s.

Let's review the different parts of this little project:

- We use a middleware function to log all requests, which helps us do debugging. We also use a middleware at the end to serve the 404 page.
- We use Express's routing to direct users to the homepage, the "add a new entry" view, and the POST for adding a new entry.
- We use Express and EJS to render pages. It lets us dynamically create HTML; we use this to dynamically display the content.

3.6 Summary

In this chapter you saw that:

- Express is a library that sits on top of Node and abstracts away a lot of that complexity

- Express has four main features:
- Middleware for letting a request flow through multiple headers
    - Routing for handling a request at a specific spot
    - Convenience methods and properties
    - Views for dynamically rendering HTML
- Many templating engines have been ported to work with Express. A popular one is called EJS, which is the simplest for folks who know already HTML.

4 Middleware

Without any framework like Express, Node.js gives you a pretty simple API. Create a function that handles requests, pass it to `http.createServer`, and call it a day. While this API is simple, your request handler function can get unwieldy as your app grows.

Express helps to mitigate some of these issues. One of the ways it does this is through the use of something called *middleware*. Where framework-free Node has you writing a single large request handler function for your entire app, middleware allows you to break these request handler functions into smaller bits. These smaller functions tend to handle one thing at a time. One might log all of the requests that come into your server; another might parse special values of incoming requests; another might authenticate users.

In this chapter, we'll learn:

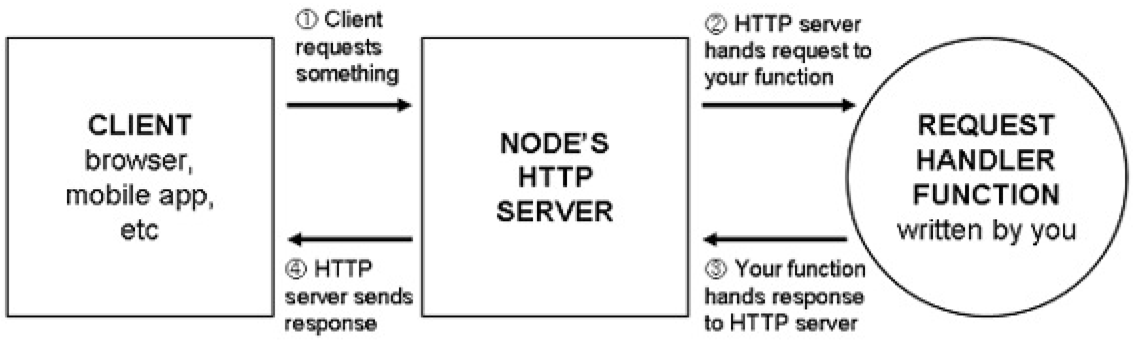
- What middleware is
- How a request flows through Express middleware; the "middleware stack"
- How to use middleware
- How to write your own middleware
- Helpful third-party Express middleware

Conceptually, middleware is the biggest part of Express. At the end of the day, most of the Express code you write is middleware in one way or another. Hopefully, after this chapter, you'll see why!

4.1 Middleware and the middleware stack

At the end of the day, web servers listen for requests, parse those requests, and send responses.

The Node runtime will get these requests first. It'll turn those requests from raw bytes into two JavaScript objects that you can handle: one object for the request and one object for the response. Conventionally, the request object is called`req` and the response object is called `res`.



① Client requests something
② HTTP server hands request to your function
③ Your function hands response to HTTP server
④ HTTP server sends response
CLIENT browser, mobile app, etc
NODE'S HTTP SERVER
REQUEST HANDLER FUNCTION written by you

Figure 4.1: When working with node.js by itself, we have one function that gives us a request object representing the incoming request and a response object representing the response node should send back to the client.

These two objects will be sent to a JavaScript function that you'll write. You'll parse `req` to see what the user wants and manipulate `res` to prepare your response.

After awhile, you're done writing to the response. When that's happened, you'll call `res.end`. This signals to Node that the response is all done and ready to be sent over the wire. The Node runtime will see what you've done to the response object, turn it into another bundle of bytes, and send it over the Internet to whoever requested it.

In Node, these two objects are passed through just one function. In Express, however, these objects are passed through an *array* of functions, called the *middleware stack*. Express will start at the first function in the stack and continue in order down the stack.

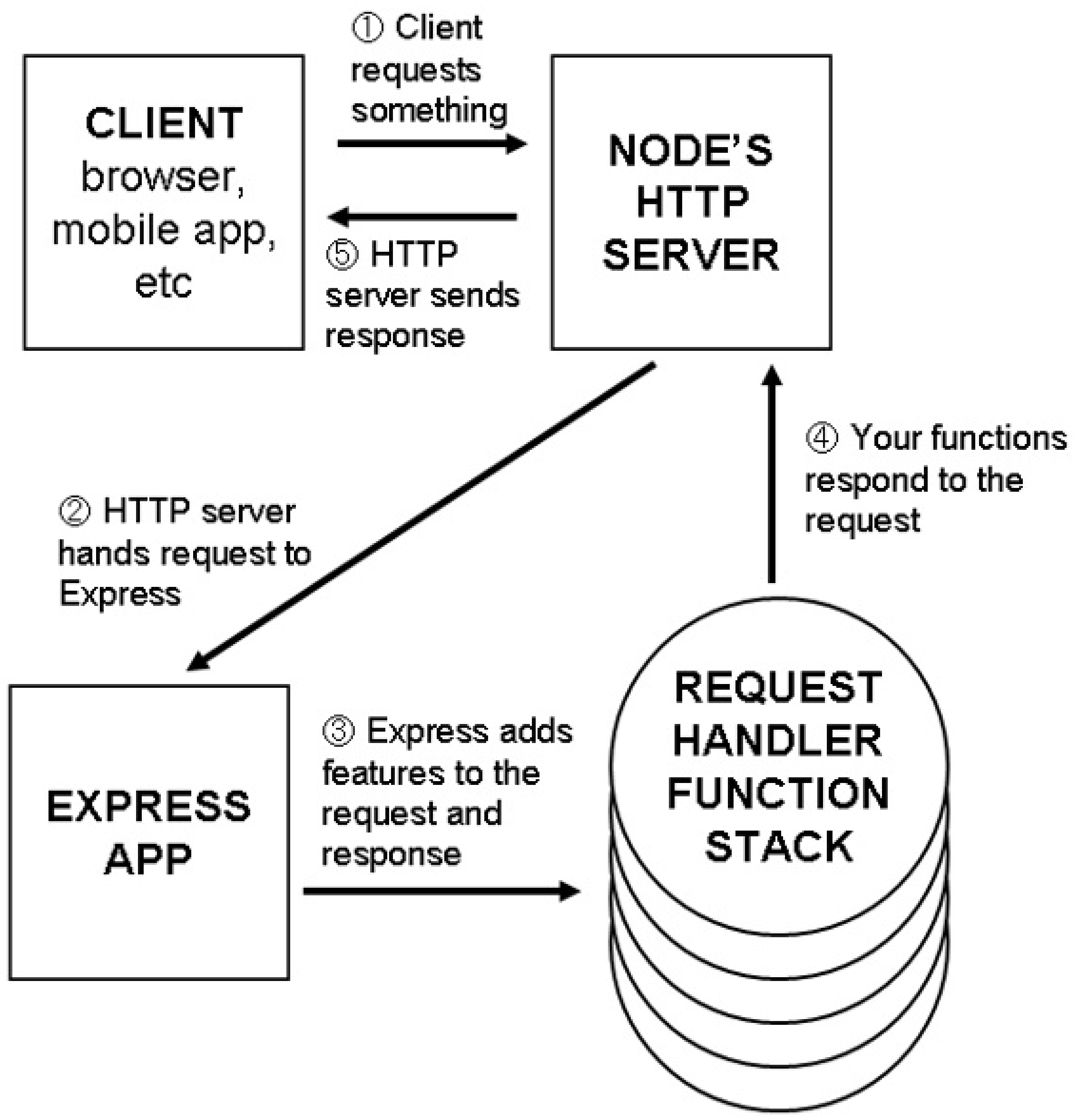

① Client requests something
CLIENT browser, mobile app, etc
NODE'S HTTP SERVER
⑤ HTTP server sends response
② HTTP server hands request to Express
④ Your functions respond to the request
EXPRESS APP
③ Express adds features to the request and response
REQUEST HANDLER FUNCTION STACK

Figure 4.2 When working in Express, the one request handler function is replaced with a stack of middleware functions instead.

Every function in this stack takes three arguments. The first two are the request and the response objects from before. They're given to us by Node, although Express decorates them with a few extra convenience features that we discussed in the previous chapter.

The third argument to each of these functions is itself a function (conventionally called `next`). When `next` is called, Express will go on to the next function in the stack.

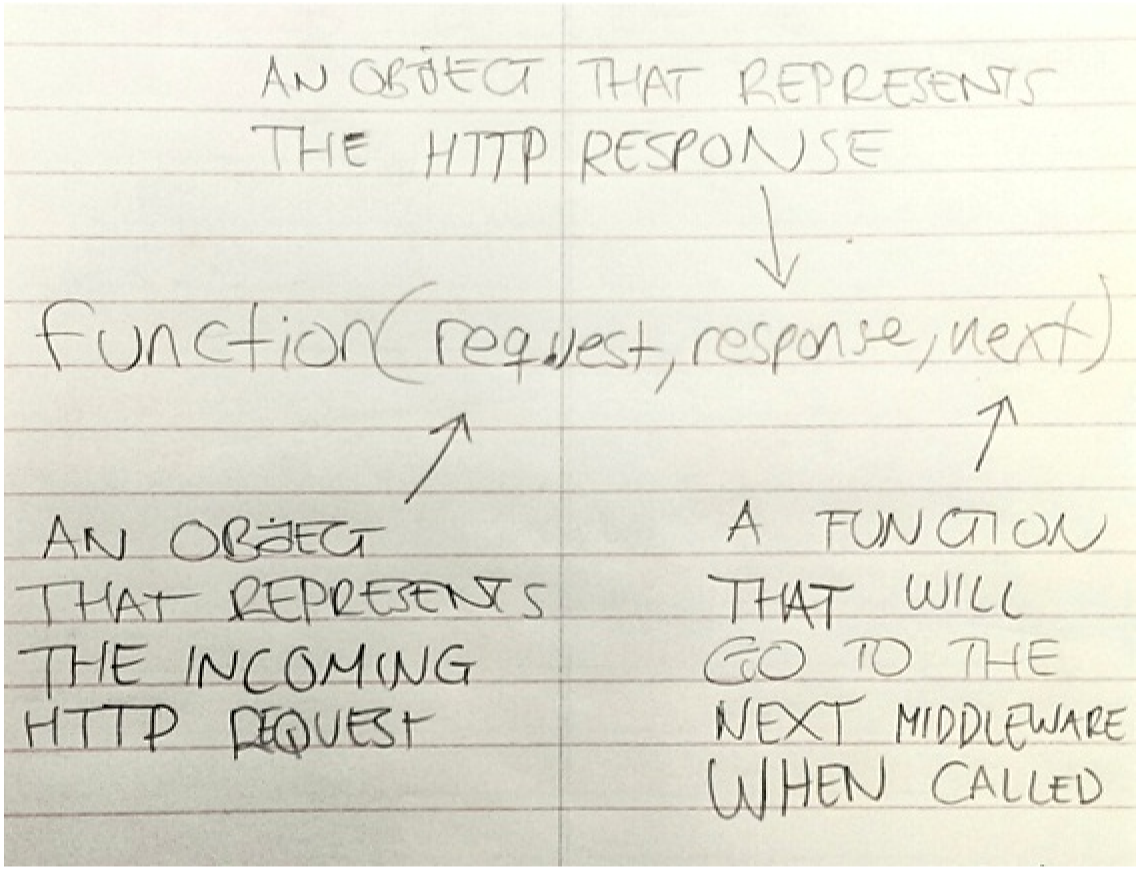

AN OBJECT THAT REPRESENTS THE HTTP RESPONSE
function(request, response, next)
AN OBJECT THAT REPRESENTS THE INCOMING HTTP REQUEST
A FUNCTION THAT WILL GO TO THE NEXT MIDDLEWARE WHEN CALLED

Figure 4.3: All middleware functions have the same signature with three functions: response, request and next.

Eventually, one of these functions in the stack must call `res.end`, which will end the request. (In Express, you can also call some other methods like `res.send` or `res.sendFile`, but these call `res.end` internally.) You can call `res.end` in any of the functions in the middleware stack, but you must only do it once or you'll get an error.

This might be a little abstract and foggy. Let's see an example of how this works by building ourselves a static file server.

4.2 Example app: a static file server

Let's build ourselves a simple little application that serves files from a folder. You can put anything in this folder and it'll be served—HTML files, images, or an MP3 of yourself singing "My Heart Will Go On" by Celine Dion.

This folder will be called "static" and it will live in our project's directory. If there's a file called `celine.mp3` and a user visits `/celine.mp3`, our server should send that MP3 over the internet. If the user requests `/burrito.html` no such file exists in the folder, our server should send a 404 error.

Another requirement: our server should log every request, whether it's successful or not. It should log the URL that the user requested with the time that they requested it.

This Express application will be made up of three functions on the middleware stack:

1. The logger. This will output the requested URL and the time it was requested to the console. It'll always continue onto the next middleware (in terms of code, it'll always call `next`).
2. The static file sender. This will check if the file exists in the folder. If it does, it'll send that file over the internet. If the requested file doesn't exist, it'll continue onto the final middleware (once again, calling `next`).
3. The 404 handler. If this middleware is hit, it means that the previous one didn't find a file, and we should return a 404 message and finish up the request.
4. You could visualize this middleware stack like this:

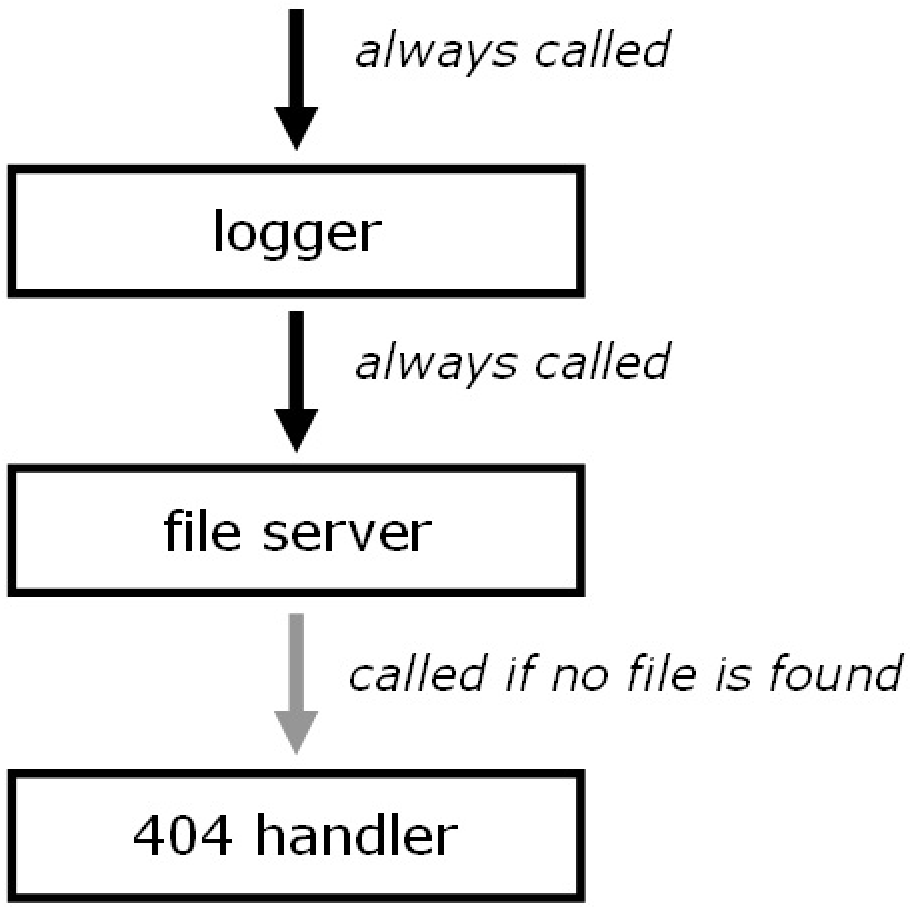


Figure 4.4 The middleware stack of our static file server application.

5. Okay, enough talking. Let's build this thing.

4.2.1 Getting set up

Start by making a new directory. You can call it whatever you'd like; let's choose `static-file-fun`. Inside of this directory, create a file called `package.json`. This file is present in every Node.js project and describes metadata about your package, from its title to its third-party dependencies.

Listing 4.1 The package.json file for our static file application

```
{
  "name": "static-file-fun",    #A
  "private": true,              #B
  "scripts": {
    "start": "node app.js"      #C
  }
}
```

#A The "name" key defines the name of your package. It's not required for private

projects (see #B), but we’ll add it.
#B The “private” key tells Node that this shouldn’t be published in the public Node module registry. This should be set to "true" for your own personal projects.
#C When you run “npm start”, it’ll run “node app.js”.

Once you've saved this `package.json`, you'll want to install the latest version of Express. From inside of this directory, run `npm install express --save`. This will install Express into a directory called `node_modules` inside of this folder. It'll also add Express as a dependency in `package.json`. `package.json` will now look like this:

Listing 4.2 The updated package.json file for our static file application

```
{
  "name": "static-file-fun",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.12.2"   #A
  }
}
```

#A Your dependency versions may vary.

Next, create a folder called "static" inside of this new project directory (right next to `package.json`). Put a few files inside; maybe an HTML file or an image or two. It doesn't *really* matter what you put in here, but put some files that your example app will serve.

Finally, create `app.js` in the root of your project, which will contain all of our app's code. Your folder structure will look something like this:

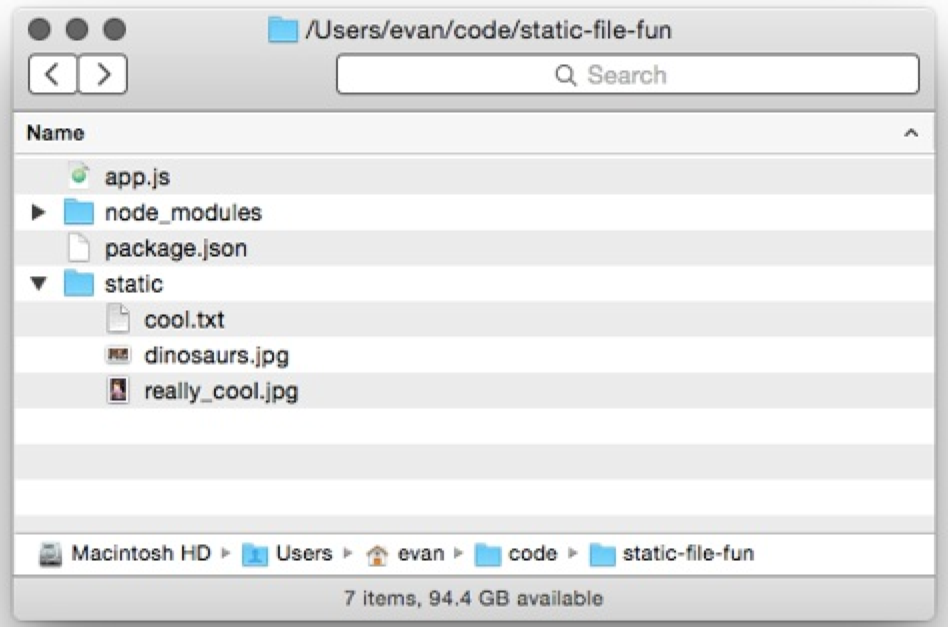


Figure 4.5 The directory structure of Static File Fun.

When you want to run this app, you'll run `npm start`. This command will look inside your `package.json` file, see that you've added a script called "start", and run that command. In this case, it'll run `node app.js`.

Running `npm start` won't do anything yet—we haven't written our app yet!—but you'll run that whenever you want to run your application.

Why use npm start?

You might be wondering why we used `npm start` at all—why didn't we just run `node app.js`? There are three reasons we might do this.

First, it's a convention. Most Node web servers can be started with `npm start`, regardless of the project's structure. If instead of `app.js` someone had chosen `application.js`, you'd have to know about that change. The Node community seems to have settled on a common convention here.

Second, it allows you to run a more complex command (or set of commands) with a relatively simple one. Our app is pretty simple now, but starting it could be more complex in the future. Perhaps we'll need to start up a database server or clear a giant log file. Keeping this complexity under the blanket of a simple command helps keep things consistent and more pleasant.

The third reason is a little more nuanced. npm lets you install packages globally, so you can run them just like any other terminal command. Bower is a common one, letting you install front-end dependencies from the command line with the newly-installed`bower` command. You install things like Bower globally on your system. npm scripts allow you to add new commands to your project *without* installing them globally, so that you can keep *all* of your dependencies inside of your project so that you can have unique versions per project. This reason comes in handy for things like testing and build scripts, as we'll see down the line.

At the end of the day, you could just run `node app.js` and never type `npm start`, but I find the above reasons compelling enough to do it.

Okay. Let's write the app!

4.2.2 Writing our first middleware function: the logger

We'll start by making our app log requests, just to get started.

Put the following inside of `app.js`:

Listing 4.3 Start app.js for our static file server

```
var express = require("express");  #A
var path = require("path");        #A
var fs = require("fs");            #A

var app = express();   #B

app.use(function(req, res, next) {                      #C
  console.log("Request IP: " + req.url);                #C
  console.log("Request date: " + new Date());           #C
});                                                     #C
```

```
app.listen(3000, function() {                #D
  console.log("App started on port 3000");  #D
});                                          #D
```

#A Require the modules we need. We'll use Express in this example, but we'll use Node's built-in Path and filesystem ("fs") modules soon.
#B Create a new Express application and put it inside the "app" variable.
#C This middleware logs all incoming requests. It has a bug, though!
#D Start the app on port 3000 and log out when it's started!

For now, all we have is an application that logs every request that comes into the server. Once we've set up our app (the first few lines), we call `app.use` to add a function to our application's middleware stack. When a request comes into this application, that function will be called.

Unfortunately, even this simple app has a critical bug. Run `npm start` and visit `localhost:3000` in your browser to see it.

You'll see the request being logged into the console, and that's great news. But your browser will hang—the loading spinner will spin and spin and spin, until the request eventually times out and you get an error in your browser. That's not good!

This is happening because we didn't call `next`.

When your middleware function is finished, it needs to do one of two things:

1. The function needs to finish responding to the request (with `res.end` or one of Express's convenience methods like `res.send` or `res.sendFile`).
2. The function needs to call `next` to continue onto the next function in the middleware stack.

If you do one of those two things, your app will work just fine. If you do neither, inbound requests will never get a response; their loading spinners will never stop spinning (this is what happened above). If you do *both*, only the first "response finisher" will go through and the rest will be ignored, which is almost certainly unintentional!

These bugs are usually pretty easy to catch once you know how to spot them. If you're not responding to the request and you're not calling `next`, it'll look like your server is super slow.

Let's fix our middleware by calling `next`.

Listing 4.4 Fixing our logging middleware

```
// …

app.use(function(req, res, next) {
  console.log("Request IP: " + req.url);
  console.log("Request date: " + new Date());
  next();   #A
});

// …
```

#A This is the critical new line!

Now, if you stop your app, run `npm start` again, and visit http://localhost:3000 in your browser, you should see your server logging all of the requests and immediately failing with an error message (something like "Cannot GET /"). Because we're never responding to the request ourselves, Express will give an error to the user, and it'll happen immediately.

Sick of restarting your server?

So far, when you change your code, you have to stop your server and start it again. This can get repetitive! You can install a tool called Nodemon. Nodemon will watch all of your files for changes and restart if it detects any.

You can install Nodemon by running `npm install nodemon --global`.

Once it's installed, you can start a file in watch mode by replacing "node" with "nodemon" in your command. For example, if you typed `node app.js` before, just change it to `nodemon app.js`, and your app will continuously reload when it changes.

Now that we've written our logger, let's write the next part—the static file server middleware.

4.2.3 The static file server middleware

At a high level, this is what the static file server middleware should do:

1. Check if the requested file exists in the static directory.
2. If it exists, respond with the file and call it a day. In code terms, this is a call to `res.sendFile`.
3. If the file doesn't exist, continue onto the next middleware in the stack. In code terms, this is a call to `next`.

Let's turn that requirement into code. We'll start by building it ourselves to understand how it works, and then we'll shorten it with some helpful third-party code.

We'll make use of Node's built-in `path` module, which will let us determine the path that the user requests. To determine whether the file exists, we'll use another Node built-in: the `fs` module.

Add this to `app.js` *after* your logging middleware:

Listing 4.5 Adding static file middleware to the middleware stack

```
// …
app.use(function(req, res, next) {
  // …
});

app.use(function(req, res, next) {
  var filePath = path.join(__dirname, "static", req.url);  #A
  fs.exists(filePath, function(exists) {                      #B
    if (exists) {                                             #C
      res.sendFile(filePath);                                 #C
    } else {                                                  #D
```

```
      next();                                                   #D
    }
  });
});

app.listen(3000, function() {
  // …
```

#A Use path.join to find the path where the file should be (whether it's there or not).
#B Built-in fs.exists will call your callback when it determines whether your file exists.
#C If the file exists, call res.sendFile.
#D Otherwise, continue onto the next middleware.

The first thing we do in this function is use `path.join` to determine the path of the file. If the user visits `/celine.mp3`, `req.url` will be the string `"/celine.mp3"`. Therefore, `filePath` will be something like`"/path/to/your/project/static/celine.mp3"`. The path will look pretty different depending on where you've stored your project and on your operating system, but it'll be the path to the file that was requested.

Next, we call `fs.exists`. This is a function that takes two arguments. The first is the path to check (the `filePath` we just figured out) and the second is a function. When Node has determined whether the file exists, it'll call this callback with one argument: `true` (the file exists) or `false` (the file doesn't exist).

Express applications have asynchronous behavior like this *all the time*. That's why we have to have `next` in the first place! If everything in were synchronous, Express would know exactly where every middleware ended: when the function finished (either by calling `return` or hitting the end). We wouldn't need to have `next` anywhere. But because things are asynchronous, we need manually to tell Express when to continue onto the next middleware in the stack.

Once the callback has completed, we run through a simple conditional. If the file exists, send the file. Otherwise, continue onto the next middleware.

Now, when you run your app with `npm start`, try visiting some resources you've put into the static file directory. If you have a file called `secret_plans.txt` in the static file folder, visit `localhost:3000/secret_plans.txt` to see it. You should also continue to see the logging, just as before.

If you visit a URL that doesn't have a corresponding file, you should still see the error message from before. This is because you're calling `next` and there's no more middleware in the stack. Let's add the final one—the 404 handler.

4.2.4 404 handler middleware

The 404 handler is the last function in our middleware stack. It'll always send a 404 error, no matter what. Add this after the previous middleware:

Listing 4.6 Our final middleware: the 404 handler

```
// …

app.use(function(req, res) {    #A
  res.status(404);              #B
  res.send("File not found!"); #C
});

// …
```

#A We've omitted the "next" argument because we won't use it.
#B Set the status code to 404.
#C Send the error "File not found!"

This is the final piece of the puzzle.

Now, when you start your server, you'll see the whole thing in action! If you visit a file that's in the folder, it'll show up. If not, you'll see your 404 error. And all the while, you'll see logs in the console.

For a moment, try moving the 404 handler. Make the *first* middleware in the stack instead of the last. If you re-run your app, you'll see that you always get a 404 error no matter what. Your app hits the first middleware and never continues on. The order of your middleware stack is important—make sure your requests flow through in the proper order.

Our app works! Here's what it should look like:

Listing 4.7 The first version of our static file app (app.js)

```
var express = require("express");
var path = require("path");
var fs = require("fs");

var app = express();

app.use(function(req, res, next) {
  console.log("Request IP: " + req.url);
  console.log("Request date: " + new Date());
  next();
});

app.use(function(req, res, next) {
  var filePath = path.join(__dirname, "static", req.url);
  fs.exists(filePath, function(exists) {
    if (exists) {
      res.sendFile(filePath);
    } else {
      next();
    }
  });
});

app.use(function(req, res) {
  res.status(404);
  res.send("File not found!");
});

app.listen(3000, function() {
  console.log("App started on port 3000");
});
```

But as always, there's more we can do.

4.2.5 Switching our logger to an open-source one: Morgan

A common piece of advice in software development is "don't reinvent the wheel". If someone else has already solved your problem, you should take their solution and move onto better things.

That's what we'll do with our logging middleware. We'll remove the hard work we put in (all five lines) and use a piece of middleware called Morgan (at https://github.com/expressjs/morgan). It's not baked into core Express but it's maintained by the Express team.

Morgan describes itself as "request logger middleware", which is exactly what we want!

To install it, run `npm install morgan --save` to install the latest version of the Morgan package. You'll see it inside a new folder inside of `node_modules` and it'll also appear in `package.json`.

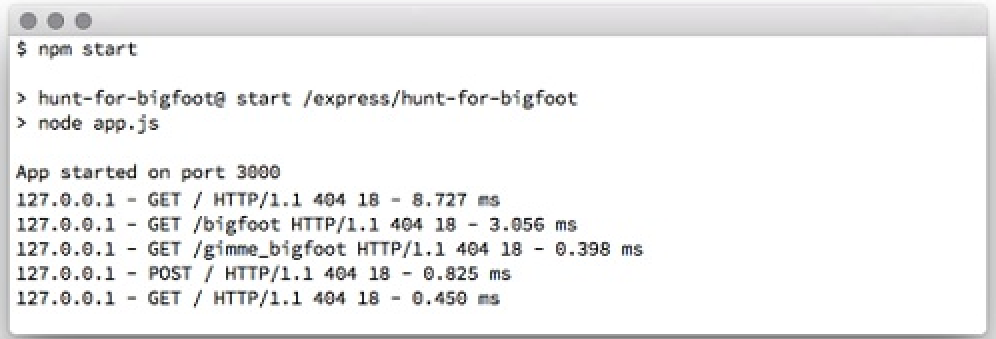
Now, let's change `app.js` to use Morgan instead of our logging middleware.

Listing 4.8 app.js that now uses Morgan

```
var express = require("express");  #A
var morgan = require("morgan");    #B
// …

var app = express();

app.use(morgan("short"));  #C

// …
```

#A Require Express, just like before.
#B Require Morgan.
#C Use the Morgan middleware instead of the one we used to have.

Now, when you run this app, you'll see output like Figure 4.6, with the IP address and a bunch of other useful information.



```
$ npm start

> hunt-for-bigfoot@ start /express/hunt-for-bigfoot
> node app.js

App started on port 3000
127.0.0.1 - GET / HTTP/1.1 404 18 - 8.727 ms
127.0.0.1 - GET /bigfoot HTTP/1.1 404 18 - 3.056 ms
127.0.0.1 - GET /gimme_bigfoot HTTP/1.1 404 18 - 0.398 ms
127.0.0.1 - POST / HTTP/1.1 404 18 - 0.825 ms
127.0.0.1 - GET / HTTP/1.1 404 18 - 0.450 ms
```

Figure 4.6 Our application's logs after adding Morgan.

So…what's happening here?

`morgan` is a function that *returns a middleware function*. When you call it, it will return a function like the one you'd written before; it'll take 3 arguments and call `console.log`. Most third-party middleware works this way—you call a function which returns the middleware, which you then use. You could have written the above like this:

Listing 4.9 An alternative usage of Morgan

```
var morganMiddleware = morgan("short");
app.use(morganMiddleware);
```

Notice that we're calling Morgan with one argument: a string, "short". This is a Morgan-specific configuration option that dictates what the output should look like. There are other format strings that have more or less information. "combined" gives a lot of info—"tiny" gives a very minimal output. When you call Morgan with different configuration options, you're effectively making it return a different middleware function.

Morgan is the first example of open-source middleware we'll use, but we'll use a lot throughout this book. We'll use another one to replace our second middleware function: the static file server.

4.2.6 Switching to Express's built-in static file middleware

There's only one piece of middleware that's bundled with Express, and it replaces our second middleware.

It's called `express.static`. It works a lot like the middleware we wrote, but it's got a *bunch* of other features. It does a bunch of complicated tricks to achieve better security and performance. For example, it adds a caching mechanism. If you're interested in more of its benefits, you can read my blog post about the middleware at http://evanhahn.com/express-dot-static-deep-dive/.

Like Morgan, `express.static` is a function that returns a middleware function. It takes one argument: the path to the folder we'll be using for static files. To get this path, we'll use `path.join`, like before. Then we'll pass it to the static middleware.

Replace your static file middleware with this:

Listing 4.10 Replacing our static file middleware with Express's

```
// …

var staticPath = path.join(__dirname, "static");  #A
app.use(express.static(staticPath));                 #B

// …
```

#A Put the static path in a variable.
#B Use express.static to serve files from the static path.

It's a bit more complicated because it's got more features, but `express.static` functions quite similarly to what we had before. If the file exists at the path, it will send it. If not, it'll call `next` and continue on to the next middleware in the stack.

If you restart your app, you won't notice much difference in functionality, but your code will be much shorter. Because you're using battle-tested middleware instead of your own, you'll also be getting a much more reliable set of features.

Now our app code looks like this:

Figure 4.11 The next version of our static file app (app.js)

```
var express = require("express");
var morgan = require("morgan");
var path = require("path");

var app = express();

app.use(morgan("short"));

var staticPath = path.join(__dirname, "static");
app.use(express.static(staticPath));

app.use(function(req, res) {
  res.status(404);
  res.send("File not found!");
});

app.listen(3000, function() {
  console.log("App started on port 3000");
});
```

I think we can call our Express-powered static file server complete for now. Well done, hero.

4.3 Error handling middleware

Remember when I said that calling `next` would continue onto the next middleware? I lied. It was mostly true but I didn't want to confuse you.

There are two types of middleware.

We've been dealing with the first type so far; these are just regular middleware functions that take three arguments (sometimes two when `next` is discarded). Most of the time, your app is in "normal mode", which only looks at these middleware functions and skips the other.

There's a second kind that's much less used: error handling middleware. When your app is in "error mode", all regular middleware is ignored and Express will only execute error handling middleware functions. To enter "error mode", simply call `next` with an argument. It's convention to call it with an error, like `next(new Error("Something bad happened!"))`.

These middleware functions take four arguments instead of two or three. The first one is the error (the argument passed into `next`), and the rest are the three from before: `req`, `res`, and `next`. You can do anything you want in this middleware. When you're done, it's just like other middleware: you can call `res.end` or `next`. Calling `next` with no arguments will exit "error mode" and move onto the next normal middleware; calling it with an argument will continue onto the next error-handling middleware if one exists.

For example, let's say you have four middleware functions in a row. The first two are normal, the third handles errors, and the fourth is a normal one. If no errors happen, the flow will look something like this:

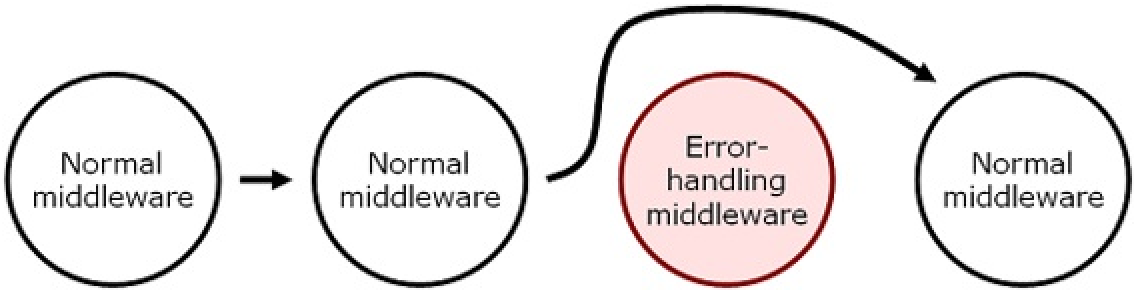


Figure 4.7 If all goes well, error handling middleware will be skipped.

If no errors happen, it'll be as if the error handling middleware never existed. To reiterate more precisely, "no errors" means "`next` was never called with any arguments". If an error *does* happen, then Express will skip over all other middleware until the first error-handling middleware in the stack. It might look like this:

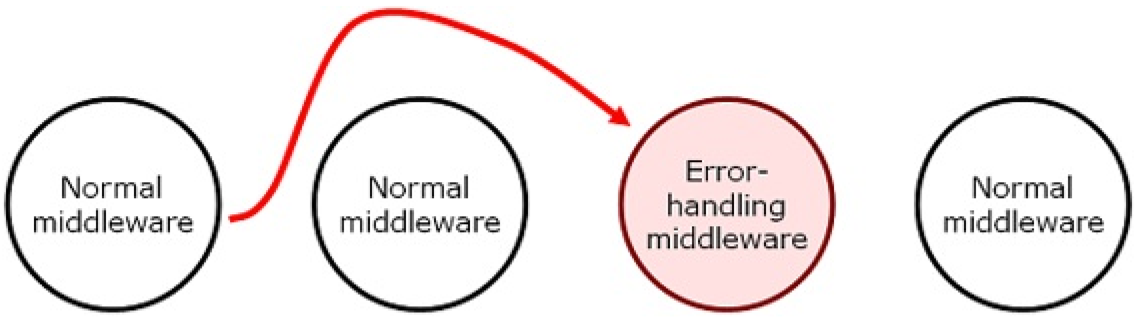


Figure 4.8 If there's an error, Express will skip straight to the error-handling middleware.

While not enforced, error handling middleware is conventionally placed at the end of your middleware stack, after all the normal middleware has been added. This is because you want to catch any errors that come cascading down from earlier in the stack.

No catching here

Express's error handling middleware does *not* handle errors that are thrown with the `throw` keyword, only when you call `next` with an argument.

Express has some protections in place for these exceptions. The app will return a 500 error and that request will fail, but the app will keep on running. Some errors like syntax errors, however, will crash your server.

Let's say that you're writing a really simple Express app that just sends a picture to the user, no matter what. We'll use `res.sendFile` just like before. Here's what that simple app might look like:

Listing 4.12 A simple app that always sends a file

```
var express = require("express");
var path = require("path");

var app = express();

var filePath = path.join(__dirname, "celine.jpg"); #A
app.use(function(req, res) {
  res.sendFile(filePath);
});

app.listen(3000, function() {
  console.log("App started on port 3000");
});
```

#A This will point to a file called celine.jpg that's in the same folder as this file.

This code should look like a simplified version of the static file server we built up above. It'll unconditionally send `celine.jpg` over the internet.

But what if that file doesn't exist on your computer for some reason? What if it has trouble reading the file because of some other weird issue? We'll want to have some way of handling that error. Error-handling middleware to the rescue!

To enter "error mode", we'll start by using a convenient feature of `res.sendFile`: it can take an extra argument which is a callback. This callback is executed after the file is sent, and if there's an error, it's passed an argument. If you wanted to print its success, you might do something like this:

Listing 4.13 Printing whether a file successfully sent

```
res.sendFile(filePath, function(err) {
  if (err) {
    console.error("File failed to send.");
  } else {
    console.log("File sent!");
  }
});
```

Instead of printing the success story to the console, we can enter "error mode" by calling `next` with an argument if there's an error. We can do something like this:

Listing 4.14 Entering "error mode" if a file fails to send

```
// …

app.use(function(req, res, next) {
  res.sendFile(filePath, function(err) {
    if (err) {
      next(new Error("Error sending file!"));
    }
  });
});

// …
```

Now that we're in this error mode, let's handle it.

It's common to have a log of all errors that happen in your app, but we don't usually display this to users. For one, a long JavaScript stack trace might be a pretty confusing to a non-technical user. It might also expose your code to hackers—if a hacker can get a glimpse into how your site works, they can find things to exploit.

Let's write some simple middleware that logs errors but doesn't actually respond to the error. It'll look a lot like our middleware from before, but instead of logging request information, it'll log the error. You could add the following to your file after all the normal middleware:

Listing 4.15 Middleware that logs all errors

```
// …

app.use(function(err, req, res, next) {   #A
  console.error(err);                     #B
  next(err);                              #C
});

// …
```

#A Notice that this is just like other middleware but with an extra argument.
#B Log the error.
#C Continue to the next middleware. Make sure to call it with the error argument to stay in "error mode".

Now, when an error comes through, we'll log it to the console so that we can investigate it later. But there's more that needs to be done to handle this error. This is similar to before—the logger did *something*, but it didn't respond to the request. Let's write that part.

You can add this after the previous middleware. This will simply respond to the error with a 500 status code.

Listing 4.16 Actually responding to the error

```
// …
```

```
app.use(function(err, req, res, next) {    #A
  res.status(500);                         #B
  res.send("Internal server error.");      #C
});

// …
```

#A Even though we're not going to use all four arguments, we have to specify them so that Express can recognize that this is error-handling middleware.
#B Set the status code to 500, which means "internal server error".
#C Send the error text.

Keep in mind that, no matter where this middleware is placed in your stack, it won't be called unless you're in "error mode"—in code, this means calling `next` with an argument.

For simple applications, there aren't loads and loads of places where things can go wrong. But as your apps grow, you'll want to remember to test errant behavior. If a request fails and it shouldn't, make sure you handle that gracefully instead of crashing. If an action should perform successfully but fails, make sure your server doesn't explode. Error-handling middleware can help this along.

4.4 *Other useful middleware*

Two different Express applications can have pretty different middleware stacks. Our example app’s stack is just one of many possible middleware configurations, and there are lots out there that you can use.

There's only one piece of middleware that's bundled with Express, and that's `express.static`. We'll be installing and using lots of other middleware throughout this book.

While it's not bundled with Express, the Express team maintains a number of middleware modules:

- `body-parser` for parsing request bodies. For example, when a user submits a form. See more at https://github.com/expressjs/body-parser.
- `cookie-parser` does what it says on the tin: parses cookies from users. It

needs to be paired with another Express-supported middleware like `express-session`. Once you've done this, you can keep track of users, providing them with user accounts and other features. We'll explore this in greater detail in Chapter 7. https://github.com/expressjs/cookie-session has more details.
- `compression` will compress responses to save on bytes. See more at https://github.com/expressjs/compression

You can find the full list on the Express homepage at http://expressjs.com/resources/middleware.html. There are also a *huge* number of third-party middleware modules that we'll explore. To name a few:

- Helmet is a bit of middleware that helps to secure your applications. It doesn't magically make you more secure, but a small amount of work can protect you from a lot of hacks. Read more at https://github.com/helmetjs/helmet. (I maintain this module, by the way, so I have to promote it!)
- `connect-assets` will compile and minify your CSS and JavaScript assets. It'll also work with CSS preprocessors like SASS, SCSS, LESS, and Stylus, should you choose to use them. See https://github.com/adunkman/connect-assets.
- Winston is a more powerful alternative to Morgan that does more robust logging (to files or databases, for example). See more at https://github.com/flatiron/winston.

This is hardly an exhaustive list. I also recommend a number of helpful modules in Appendix B if you're thirsty for even more helpers.

4.5 Summary

Middleware is the core foundation of Express, and we've explored it in this chapter. We've learned:

- Express's middleware stack, and how requests flow through that stack sequentially
- How to write our own middleware functions: a function with three arguments

- How to write and use error handling middleware: a function with four arguments
- Various open-source middleware functions, like Morgan for logging, `express.static` for serving static files, and many more

5 Routing

As we’ve seen, routing is one of Express's big features, allowing you to map different requests to different request handlers. In this chapter, we'll go far more in depth. We'll look at routing in detail, how to use Express with HTTPS, Express 4's new routers feature, and more. We'll also build a couple of routing-centric applications, one of which will be a running example throughout the book.

In this chapter, I'll tell you everything there is to know about routing in Express!

5.1 What is routing?

Let's imagine we're building the homepage for Olivia Example. She's a great lady and we're honored to build her website.

If we're a browser visiting `example.com/olivia`, here's what the "raw" HTTP request might look like:

Listing 5.1 The first line of an HTTP request

```
GET /olivia http/1.1
```

That HTTP request has a verb (`GET`), a URI (the `/olivia` part), and the HTTP version (1.1). When we're routing, we take the pair of the verb and the URI and map it to a request handler. We basically say, "Hey, Express! When you see a`GET` request to `/about_me`, run this code. And when you see a POST request to `/new_user`, run this other code."

That's pretty much it—routing maps verbs and URIs to specific code. Let's take a look at a simple example.

5.1.1 A simple example

Let's say we want to write a simple Express application that responds to the HTTP request above (an HTTP GET to `/olivia`). We'll call some methods on our Express app, like so:

Listing 5.2 A simple Express app that shows Olivia's homepage

```
var express = require("express");
var app = express();

app.get("/olivia", function(request, response) {   #B
  response.send("Welcome to Olivia's homepage!");
});

app.use(function(request, response) {   #C
  response.status(404).send("Page not found!");
});

app.listen(3000); #D
```

#B This is the magical part; this routes GET requests to /olivia to the request handler we specify.
#C If you load /olivia, that's all good. But if you load something else (like /olivia_example), we want to serve a 404 error.
#D Finally, we start the server on port 3000!

The real meat of this example is on the third line: when we get HTTP GET requests to `/olivia`, we run the specified request handler. To hammer this home: we'll ignore this if we see a GET request to some other URI, and we'll also ignore this if we see a non-GET request to `/olivia`.

This is a pretty simple example (hence the title of this section). Let's take a look at some more complex routing features.

5.2 The features of routing

So we've just looked at a simple example of routing. Conceptually, it's not too crazy: it maps an HTTP verb + URI combo to a request handler. This lets you map things like GET `/about` or POST `/user/log_in` to a specific bit of code. This is great!

But we're greedy. If Express were a vat of ice cream, we wouldn't be satisfied with one scoop. We want more scoops. We want sprinkles. We want chocolate sauce. We want more routing features.

> **NOTE** Some other frameworks (Ruby on Rails, for example) have a centralized routing file where all routes are defined in one place. Express is not this way—they can be defined in a lot of places.

5.2.1 *Grabbing parameters to routes*

The routes we've seen above could really be expressed in code with a strict equality operator (===); is the user visiting `/olivia`? That's very useful, but it doesn't give us all the expressive power we might want.

Imagine you've been tasked to make a website that has user profiles, and imagine that every user has a numeric ID. You want the URL for user #1 to be `/users/1`. User #2 should be found at `/users/2` (and so on). Rather than define, in code, a new route for *every single new user* (which would be crazy), you can define one route for everything that starts with `/users/` and then has an ID.

THE SIMPLEST WAY

The absolutely simplest way to grab a parameter is by simply putting it in your route with a colon in front of it. To grab the value, you'll look inside the params property of the request.

Listing 5.3 The simplest parameter

```
app.get("/users/:userid", function(req, res) {        #A
   var userId = parseInt(req.params.userid, 10);      #B
   // …
});
```

#A This will match requests coming into /users/123 and /users/horse_ebooks too.
#B The userid property is always a string in this case, so we have to convert it to an integer. If we visit /users/olivia, though, this will return NaN, which we'll need to handle.

In the above example, we see how to grab parameters from a more dynamic route. The code above will match what we want; things like `/users/123` and `/users/8`. But while it won't match a parameter-less `/users/` or `/users/123/posts`, it probably still matches more than what we want. For example, it'll also match `/users/cake` and `/users/horse_ebooks`. If we want to be more specific, we have a few options.

> **NOTE** While it's often that you'll want to be more specific with your parameter definitions, it might very well be that this is fine for your purposes! You might want to allow /users/123 and /users/count_dracula. Even if you only want to allow numeric parameters, you might prefer to have validation logic right in the route. As we'll see, there are other ways to do it, but that might be just fine for you.

5.2.2 Using regular expressions to match routes

Express allows you to specify your routes as strings, but it also allows you to specify them as regular expressions. This gives you more control over the routes you specify. You can also use regular expressions to match parameters, as we'll see.

> **NOTE** Regular expressions can get a little hairy. They scared me when I first started working with them, but I found that fear greatly reduced by the entry on the Mozilla Developer Network. If you need help, I strongly recommend checking it out at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Let's imagine that we want to match things like `/users/123` or `/users/456`, but not `/users/olivia`. We can code this into a regular expression and grab the number to boot.

Listing 5.5 Using regular expressions for numeric routes

```
app.get(/^\/users\/(\d+)$/, function(req, res) { #A
  var userId = parseInt(req.params[0], 10); #B
  // ...
});
```

#A This both defines the route (starts with /users/ and ends with one or more digits)
and captures the digits, which is used on the next line. If this regular expression looks daunting, that's because all regular expressions look daunting.
#B The parameters aren't named this time, so we access them by their ordinality. If we captured a second value, we'd look inside req.params[1], and so on. Note that we still capture them as strings and have to convert them manually.

This is one way to enforce the "the user ID must be an integer" constraint. Like the above, it's passed in as a string, so we have to convert it to a number (and probably to a user object further down the line).

Regular expressions can be a little difficult to read, but you can use them to define much more complex routes than these. For example, you might want to define a route that looks for ranges. That is, if you visit `/users/100-500`, you can see a list of users from IDs 100 to 500. Regular expressions make this relatively easy to express (no pun intended):

Listing 5.6 Using regular expressions for complex routes

```
app.get(/^\/users\/(\d+)-(\d+)$/, function(req, res) { #A
  var startId = parseInt(req.params[0], 10);  #B
  var endId = parseInt(req.params[1], 10);  #C
  // …
});
```

#A Like the above, this defines a route with a regular expression. This time, we capture two sets of digits on either side of a hyphen character.
#B Like before, we grab the first captured parameter as a string and have to do some conversion.
#C This is very similar to the previous line, but we're converting the second parameter, not the first.

You can daydream about the crazy number of possibilities this opens up.

For example, I once had to define a route that matched UUIDs (version 3 and 4). If you're not familiar, a UUID is a long string of hex digits that looks like this:

```
xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx
```

...where x is any hex digit and y is 8, 9, A, or B. Let's say you want to write a route that matches any UUID. It might look something like this:

Listing 5.7 UUID-matching routes with a regexp

```
var horribleRegexp = /^([0-9a-f]{8}-[0-9a-f]{4}-
[CA]4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12})$/i;

app.get(horribleRegexp, function(req, res) {
  var uuid = req.params[0];
  // ...
});
```

I could fill hundreds of pages with more examples, but I won't. The key takeaway here: you can use regular expressions to define your routes.

5.2.3 Grabbing query arguments

Another common way to dynamically pass information in URLs is to use something called "query strings". You've probably seen query strings every time you've done a search on the Internet. For example, if you searched for "javascript-themed burrito" on Google, you'd see a URL like this: `https://www.google.com/search?q=javascript-themed%20burrito`

This is passing a query. If Google were written in Express (it's not), it might handle a query like this:

Listing 5.8 Handling a search query string

```
app.get("/search", function(req, res) {
  // req.query.q == "javascript-themed burrito"
  // ...
});
```

This is pretty similar to how you handle parameters, but it allows you to grab this style of query.

NOTE There's a common security bug with query parameters, unfortunately.

If you visit ?arg=something, then req.query.arg will be a string. But if you visit ?arg=something&arg=somethingelse, then req.query.arg will be an array. We'll discuss coping with these types of issues in detail in Chapter 8, if you thirst for more. In general, though, you'll want to make sure that you don't blindly assume something is a string or an array.

5.3 Using routers to split up your app

It's likely that as your application grows, so will your number of routes. Your collaborative cat-photo montage site might start with routes for static files and for images, but you might later add user accounts, chat, forums, and the like. Your number of routes can get unwieldy.

Express 4 added a feature to help ease these growing pains; it added routers. To quote the Express documentation (don't worry if you don't perfectly understand all of this):

A router is an isolated instance of middleware and routes. Routers can be thought of as "mini" applications only capable of performing middleware and routing. Every express application has a built-in app router.

Routers behave like middleware themselves and can be "`.use()`'d" by the app o in other routers.

In other words, routers allow you to chunk your big app into many mini apps that you can later put together. For small apps, this might be overkill, but as soon as you think to yourself, "this `app.js` file is getting big", it's time to think about breaking your app down with routers.

NOTE Routers really shine when you're building a bigger application. I don't want to build a huge application in this section, so this example will have some spots that you should fill in with your imagination

Listing 5.9 Routers in action: the main app

```
var express = require("express");
var path = require("path");
var apiRouter = require("./routes/api_router");  #A
var app = express();

var staticPath = path.resolve(__dirname, "static");
app.use(express.static(staticPath));

app.use("/api", apiRouter);  #A

app.listen(3000);
```

#A We require our API router (which is defined below) and then we use it in our main app, just like we use middleware.

As you can see, we use our API router just like middleware. Routers are basically just middleware! In this case, any URL that starts with `/api` will be sent straight to our router. That means that `/api/users` and `/api/message` will use your router code, but something like `/about/celinedion` will not.

Now, let's go ahead and define our router. Think of it as a sub-application:

Listing 5.10 A sample router definition (at routes/api_router.js)

```
var express = require("express");

var ALLOWED_IPS = [
  "127.0.0.1",
  "123.456.7.89"
];

var api = express.Router();

api.use(function(req, res, next) {
  var userIsAllowed = ALLOWED_IPS.indexOf(req.ip) !== -1;
  if (!userIsAllowed) {
    res.status(401).send("Not authorized!");
  } else {
    next();
  }
});

api.get("/users", function(req, res) { /* ... */ });
api.post("/user", function(req, res) { /* ... */ });
api.get("/messages", function(req, res) { /* ... */ }); api.post("/message",
function(req, res) { /* ... */ });
```

```
module.exports = api;
```

This looks a lot like a mini-application; it supports middleware and routes. The main difference is that it can't stand alone; it has to be plugged into a "grown-up" app. Routers can do the same routing that "big" apps can do and they can use middleware.

You could imagine making a router with many sub-routers. Maybe you want to make an API router that further defers to a "users router" and a "messages router", or perhaps something else!

5.4 Serving static files

Unless you're building a web server that's 100% API (and I mean one hundred percent), you're probably going to send a static file or two. Maybe you have some CSS to send, maybe you have a single-page app that needs some static files sent, and maybe you're a donut enthusiast and have gigabytes of donut photos to serve your hungry viewers.

We've seen how to send static files before, but let's explore it in more depth.

5.4.1 Static files with middleware

We've sent static files with middleware before, but don't roll your eyes yet—we're going to dive just a little deeper.

We went over this in Chapter 2, so I won't preach the benefits of this stuff. I'll just review the code example we used before:

Listing 5.11 A simple example of express.static

```
var express = require("express");
var path = require("path");
var http = require("http");
var app = express();

var publicPath = path.resolve(__dirname, "public");  #A
app.use(express.static(publicPath)); #B

app.use(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Looks like you didn't find a static file.");
});

http.createServer(app).listen(3000);
```

#A Set up the path where our static files will sit, using Node's path module.
#B Send static files from the publicPath directory.

Recall that path.resolve helps keep our path resolution cross-platform (things are different on Windows and Mac and Linux). Also recall that this is much better than doing it all yourself! If any of this is unclear, go back and take a look at Chapter 2.

Now let's go deeper.

CHANGING THE PATHS FOR CLIENTS

It's common that you'll want to serve files at the root of your site. For example, if your URL is http://jokes.edu and you're serving `jokes.txt`, the path will be http://jokes.edu/jokes.txt.

But you might also want to mount some static files at a different URL for clients. For example, you might want a folder full of offensive-but-hilarious photos to look like it's in a folder called "offensive", so a user might visit http://jokes.edu/offensive/photo123.jpg. How might we do this?

Express to the rescue: middleware can be "mounted" at a given prefix. In other words, you can make a middleware only respond if it starts with `/offensive`.

Here's how that's done:

Listing 5.12 Mounting static file middleware

```
// …
var photoPath = path.resolve(__dirname, "offensive-photos-folder");
app.use("/offensive", express.static(photoPath));
// …
```

Now web browsers and other clients can visit your offensive photos at a path other than the root. Note that this can be done for any middleware, not just the static file middleware. Perhaps the biggest example is the one we saw above: mounting Express's routers at a prefix.

ROUTING WITH MULTIPLE STATIC FILE DIRECTORIES

I frequently find myself with static files in multiple directories. For example, I sometimes have static files in a folder called "public" and another in a folder called "user_uploads". How can we do this with Express?

Express already solves this problem with the built-in middleware feature, and because express.static is middleware, we can just apply it multiple times.

Here's how we might do that:

Listing 5.13 Serving static files from multiple directories

```
// …

var publicPath = path.resolve(__dirname, "public");  #A
var userUploadsPath = path.resolve(__dirname, "user_uploads");

app.use(express.static(publicPath)); app.use(express.static(userUploadsPath));

// …
```

#A Note that this depends on the "path" module, so make sure you require it before you use it!

Now, let's quickly imagine four scenarios, and see how the above code deals with them:

1. The user requests a resource that isn't in the public folder or the user uploads folder. In that case, both static middleware functions will continue onto the next routes and middleware.
2. The user requests a resource that's in the public folder. In that case, the first middleware will send the file and no following routes or middleware functions will be called.
3. The user requests a resource that's in the user uploads folder, but not

the public folder. The first middleware will continue on (it's not in "public"), so the second middleware will pick it up. After that, no other middleware or route will be called.

4. The user requests a resource that's in *both* the public folder and the uploads folder. In this case, because the public-serving middleware is first, you'll get the file in "public" and you'll never be able to reach the matching file in the user uploads folder.

As always, you can mount middleware at different paths to avoid the issue presented in #4. Here's how you might do that:

Listing 5.14 Serving static files from multiple directories without conflict

```
// …

app.use("/public", express.static(publicPath));
app.use("/uploads", express.static(userUploadsPath));

// …
```

Now, if "image.jpg" is in both folders, you'll be able to grab it from the public folder at `/public/image.jpg` and from the uploads folder in `/uploads/image.jpg`.

5.4.2 Routing to static files

It's possible that you'll want to send static files with a route. For example, you might want to send a user's profile picture if they visit `/users/123/profile_photo`. The static middleware has no way of knowing about this, but Express has a nice way of doing this, which uses a lot of the same internal mechanisms as the static middleware.

Let's say we want to send profile pictures when someone visits `/users/:userid/profile_photo`. Let's also say that we've got a magic function called `getProfilePhotoPath` that takes a user ID and returns the path to their profile picture. Here's how we might do that:

Listing 5.15 Sending profile pictures

```
app.get("/users/:userid/profile_photo", function(req, res) {
  res.sendFile(getProfilePhotoPath(req.params.userid));
});
```

In Chapter 2, we saw that this would be a big headache without Express. We'd have to open the file, figure out its content-type (HTML, plain text, image...), its file size, et cetera. Express's `sendFile` does all of this for us and lets you send files easily.

You can use this to send any file you want!

5.5 Using Express with HTTPS

As we discussed earlier in the chapter, HTTPS is HTTP's more secure sister. It adds a secure layer to HTTP, adding more security (although nothing is invincible). This secure layer is called TLS or SSL. The names are used interchangeably, but TLS is technically the successor to SSL.

I won't go into the crazy math involved, but TLS uses what's called public-key cryptography. Public-key crypto works like this: every peer has a public key that they share with everybody and a private key that they share with nobody. If I want to send something to you, I encrypt the message with my private key (probably somewhere on my computer) and your public key (publicly available to anyone). I can then send you messages that look like garbage to any eavesdroppers, and you decrypt it with your private key and my public key. Through crazy cool math, we can have a secure conversation even if everyone is listening to us, and we never had to agree on some kind of secret code beforehand.

If this is a bit confusing, just remember that both peers have a private key and a public key.

In TLS, the public key also has a special property: it's also something called a certificate. If I'm talking to you, you'll present me with your certificate (AKA your public key), and I'll make sure it's actually you by making sure a certificate authority says "yeah, that's you." Your browser has a list of certificate authorities that it trusts; companies like VeriSign and Google run these certificate authorities.

I imagine certificate authorities as a bodyguard. When I'm talking to somebody, I look up at my bodyguard and say "hey, is this person who they say they are?". My bodyguard looks down at me and gives a small nod, or maybe a shake of the head.

> **NOTE** Some hosting providers like Heroku will do all the HTTPS for you so that you don't have to worry about it. This section is only useful if you have to do HTTPS yourself!

First, you'll need to generate your public and private keys. We'll use OpenSSL for this. If you're on Windows, grab a binary from https://www.openssl.org/related/binaries.html. It should come preinstalled on Mac OS X. If you're on a Linux machine with a package manager (like Arch, Gentoo, Ubuntu, or Debian) and it's not already installed, install it with your OS's package manager. You can check if OpenSSL is installed by typing `openssl version` at your command prompt.

From there, we'll run the following two commands:

Listing 5.16 Using OpenSSL to create your private key and signing request

```
openssl genrsa -out privatekey.pem 1024   #A
openssl req -new -key privatekey.pem -out request.pem  #B
```

#A This generates your private key into privatekey.pem.
#B This generates a certificate signing request into request.pem. You'll have to fill out a bunch of information.

The first command simply generates your private key; anyone can do this. The next command generates a certificate signing request. It'll ask you a bunch of information, and then spit out a file into `request.pem`. From here, you have to request a certificate from a certificate authority. Several groups on the Internet are working on Let's Encrypt, a free and automated certificate authority. You can check out the service at https://letsencrypt.org/. If you'd prefer a different certificate authority, you can shop around online.

Once they've given you a certificate, you can use Node's built-in HTTPS module with Express. It's very similar to the HTTP module, but you'll have to supply your certificate and private key.

Listing 5.17 Using HTTPS with an Express app

```
var express = require("express");  #A
var https = require("https");   #A
var fs = require("fs");   #A

var app = express();
// ... define your app ...

var httpsOptions = {
   key: fs.readFileSync("path/to/private/key.pem"),   #B
   cert: fs.readFileSync("path/to/certificate.pem")   #B
};
https.createServer(httpsOptions, app).listen(3000);   #C
```

#A First, we require the modules we need.
#B After defining our application, we define an object that contains our private key and our certificate.
#C Now we pass that object into https.createServer, which is otherwise just like the http.createServer that we've seen before.

Other than the fact that we have to pass the private key and certificate as arguments, this is otherwise very similar to the http.createServer we've seen before.

If you want to run both an HTTP server and an HTTPS server, just start both!

Listing 5.18 Using HTTP and HTTPS with Express

```
var express = require("express");
var http = require("http");
var https = require("https");
var fs = require("fs");

var app = express();

// ... define your app ...
```

```
var httpsOptions = {
  key: fs.readFileSync("path/to/private/key.pem"),
  cert: fs.readFileSync("path/to/certificate.pem")
};
http.createServer(app).listen(80);
https.createServer(httpsOptions, app).listen(443)
```

All we have to do is run both servers on different ports, and we're done! That's HTTPS.

5.6 Putting it all together: a simple routing demo

Let's take what we've learned and build a simple web application that returns the temperature by your United States ZIP code.

> **NOTE** I'm an American, so this example will use the US-style postal code, called a ZIP code. ZIP codes are five digits long and can give you a pretty good ballpark location. There are 42,522 of them, so if the US is 3.7 million square miles, each ZIP code covers about 87 square miles on average. Because we're going to use ZIP codes, this example will only work in the United States. It shouldn't be too much of a stretch to make a similar application that works elsewhere (if you're inspired, you could try using the HTML5 Geolocation API!).

Our application will basically have two parts:

1. A static homepage that asks the user for their ZIP code. After the user types it in, it will load the weather via an asynchronous JavaScript request (also known as an AJAX request).
2. Because we're using JavaScript, we'll send the temperature as JSON. We'll define a route for `/12345` which will return the weather at ZIP code 12345.

Let's get started.

5.6.1 Getting set up

For this application, we'll use four Node packages: Express (obviously), ForecastIO (for grabbing weather data from the free API called Forecast.io), Zippity-do-dah (for turning ZIP codes into latitude/longitude pairs), and EJS (for rendering HTML views). (These are some pretty good names, right? Especially "zippity-do-dah".)

Make a new Express application. You'll want to make sure the `package.json` looks something like this when it's time to start:

Listing 5.19 package.json for this application

```
{
  "name": "temperature-by-zip",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "ejs": "^2.3.1",
    "express": "^4.12.4",
    "forecastio": "^0.2.0",
    "zippity-do-dah": "0.0.x"
  }
}
```

Make sure you have all of these dependencies installed by running `npm install` in your application's directory.

On the client, we'll depend on jQuery and a minimal CSS framework called Pure (more information at http://purecss.io/). It's likely that you already know about jQuery, but Pure is a bit more obscure (though most everything is more obscure than jQuery). Pure gives us a little bit of styling for text and forms, similar to Twitter's Bootstrap. The difference with Pure is that it's far lighter-weight, which better suits this kind of application.

Make two directories: one called `public` and one called `views`.

Next, we'll need to get an API key from Forecast.io. Visit their developers URL at https://developers.forecast.io). Register for an account. At bottom of the dashboard page is your API key, which is a string of 32 characters. You'll need to copy this API key into your code in just a moment, so make sure you have it ready.

We're ready to get started!

5.6.2 The main app code

Now that we're all set up, it's time to code! Let's start with the main application JavaScript. If you followed the example at the end of Chapter 2, this business should be pretty familiar.

Create `app.js` and put this inside:

Listing 5.20 app.js

```
var path = require("path");  #A
var express = require("express");  #A
var zipdb = require("zippity-do-dah");  #A
var ForecastIo = require("forecastio");  #A

var app = express();  #B
var weather = new ForecastIo("YOUR FORECAST.IO API KEY HERE");  #C

app.use(express.static(path.resolve(__dirname, "public")));  #D

app.set("views", path.resolve(__dirname, "views"));  #E
app.set("view engine", "ejs");  #E

app.get("/", function(req, res) { #F
  res.render("index");
});

app.get(/^\/(\d{5})$/, function(req, res, next) {
  var zipcode = req.params[0];  #G
  var location = zipdb.zipcode(zipcode); #H
  if (!location.zipcode) {#I
    next(); #I
    return;
  }

  var latitude = location.latitude;
  var longitude = location.longitude;

  weather.forecast(latitude, longitude, function(err, data) {
    if (err) {
      next();
      return;
    }

    res.json({ #J
      zipcode: zipcode,
      temperature: data.currently.temperature
    });
  });
});

app.use(function(req, res) { #K
  res.status(404).render("404");
});
app.listen(3000);  #L
```

#A We start by including Node's built-in path module, Express, zippity-do-dah, and ForecastIO. Nothing too different than what we've seen before!
#B Create a new Express application.
#C Create a new ForecastIO object with your API key. Make sure to fill this in!
#D Serve static files out of "public" with Express's built-in static file middleware.
#E Use EJS as our view engine, and serve the views out of a folder called "views".
#F Render the "index" view if we hit the homepage.
#G This is an example of Express's regular expression routing feature. Regular expressions are always nasty to read, but this one basically says "give me five numbers". The parenthesis "capture" the specified ZIP code and pass it as req.params[0].
#H Use zippity-do-dah to grab location data with the ZIP code.
#I zippity-do-dah just returns an empty object ({}) when no results are found. This probes the object for a zipcode property, and if we're missing it, then this won't work, and we should continue on.
#J We'll send this JSON object with Express's convenient json method.
#K If we miss the static files middleware, miss the handler for the root URL (/), and miss the weather URL, then show a 404 error.
#L Start the app on port 3000!

Now we need to fill in the client. This means making some views with EJS, and as we'll see, we'll add a splash of CSS and a bit of client-side JavaScript.

5.6.3 The two views

There are two views in this application; the 404 page and the homepage. We want our site to look consistent across pages, so let's make a template. We'll need to make a header and a footer.

Let's start with the header. Save the following into a file called header.ejs:

Listing 5.21 views/header.ejs

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Temperature by ZIP code</title>
  <link rel="stylesheet" href="http://yui.yahooapis.com/pure/0.4.2/pure-min.css">
  <link rel="stylesheet" href="/main.css">
</head>
<body>
```

Next, let's close off the page in footer.ejs:

Listing 5.22 views/footer.ejs

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

Now that we have our template, let's fill in the simple 404 page (as 404.ejs):

Listing 5.23 views/404.ejs

```
<% include header %>
  <h1>404 error! File not found.</h1>
<% include footer %>
```

The index homepage isn't too complex, either. Save this sucker as index.ejs.

Listing 5.24 views/index.ejs

```
<% include header %>

<h1>What's your ZIP code?</h1>

<form class="pure-form">
  <fieldset>
```

```
    <input type="number" name="zip" placeholder="12345"[CA]
    autofocus required>
    <input type="submit" class="pure-button[CA]
    pure-button-primary" value="Go">
  </fieldset>
</form>

<script src="//ajax.googleapis.com/ajax/libs/[CA/]
jquery/2.1.1/jquery.min.js"></script>
<script src="/main.js"></script>

<% include footer %>
```

There are a couple of references to the Pure CSS framework in the index code; all they do is apply some styling so our page looks a little better.

Speaking of styling, we'll need to fill in main.css that we specified in the layout. Save the following into `public/main.css`:

Listing 5.25 public/main.css

```
html {
  display: table;
  width: 100%;
  height: 100%;
}
body {
  display: table-cell;
  vertical-align: middle;
  text-align: center;
}
```

This CSS effectively centers all the page's content, both horizontally and vertically. This isn't a CSS book, so don't worry if you don't understand exactly what's going on in the above.

Now we have everything other than our client-side JavaScript! You can try to `npm start` this app right now. You should be able to see the homepage at `http://localhost:3000`, the 404 page at`http://localhost:3000/some/garbage/url`, and the weather should load 12345's temperature as JSON at `http://localhost:3000/12345`.

Let's finish it off with our client-side JavaScript.

Save this stuff in `public/main.js`:

Listing 5.26 public/main.js

```
$(function() {

  var $h1 = $("h1");
  var $zip = $("input[name='zip']");

  $("form").on("submit", function(event) {

    event.preventDefault();  #A

    var zipCode = $.trim($zip.val());
    $h1.text("Loading...");

    var request = $.ajax({   #B
      url: "/" + zipCode,
      dataType: "json"
    });
    request.done(function(data) {     #C
      var temperature = data.temperature;
      $h1.text("It is " + temperature + "&#176; in " + zipCode + "."); #D
    });
    request.fail(function() {    #E
      $h1.text("Error!");
    });

  });

});
```

#A We don't want the form doing what HTML would normally make it do -- we want to drive!

#B We spin off an AJAX request. If we've typed "12345" into the ZIP code field, we'll be visiting /12345 to do this request.

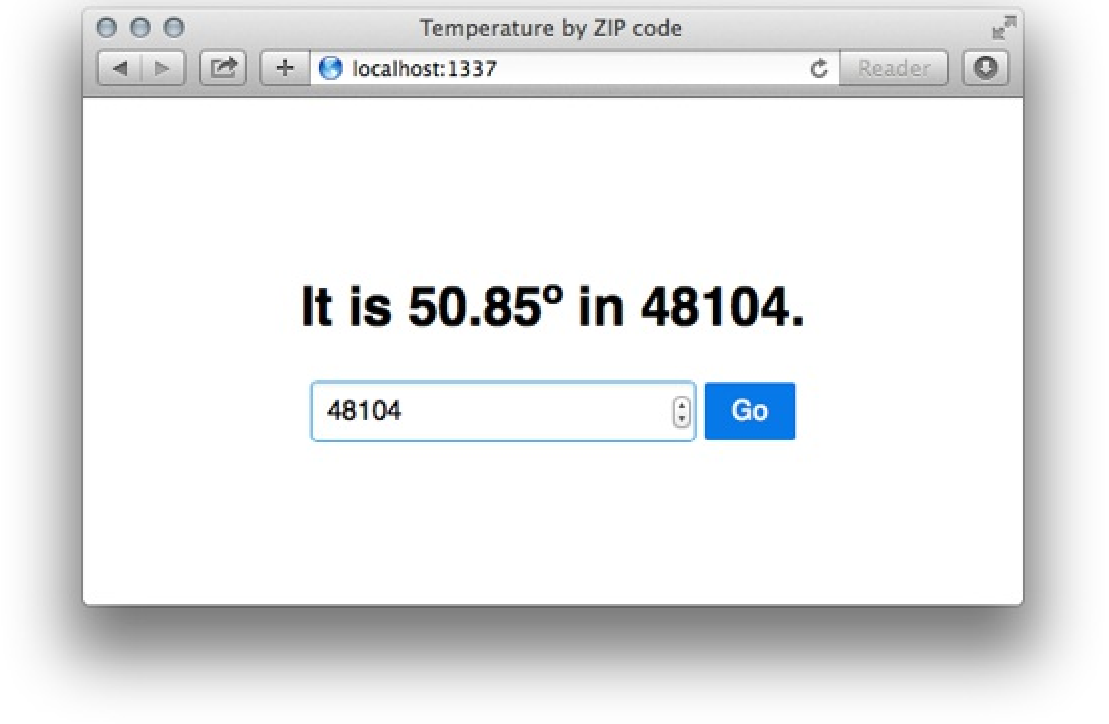
#C When the request succeeds, we'll update the header with the current temperature.

#D &#176; is the HTML character code for the ° degree symbol.

#E If there's an error (either on the client or the server), make sure that an error is shown.

5.6.4 The application in action

With that, you can start the application with `npm start`. Visit `http://localhost:3000`, type in a ZIP code, and watch the temperature appear!



It is 50.85° in 48104.

48104

Figure 5.1 Temperature by ZIP code in action

That's our simple application! It takes advantage of Express's helpful routing features, serves HTML views, JSON, and static files.

If you'd like, you can extend this application to work with more than just United States ZIP codes, or show more than just the temperature, or add API documentation, or add better error handling, or maybe more!

5.7 Summary

In this chapter, you saw:

- What routing is at a conceptual level: a mapping of a URL to a piece of code

- Simple routing, pattern-matching routing, and more
- Grabbing parameters from routes
- Using Express 4's new routers feature
- Using middleware with routing
- Serving static files with `express.static`, Express's built-in static file middleware
- How to use Express with Node's built-in HTTPS module

6 Building APIs

Friends, gather round. This chapter marks a new beginning. Today, we exit the abstract but critical "core Express" and enter the real world. For the rest of this book, we'll be building much more real systems atop Express. We'll start with APIs.

"API" is a pretty broad term.

It stands for "Application Programming Interface", which doesn't demystify the term much. If it were up to me (and it isn't), I'd rename it to something like "Software Interface". Where a user interface is meant to be consumed by human users, a software interface is meant to be consumed by code. At some level, all user interfaces sit on top of software interfaces—all user interfaces sit on top of some APIs.

At a high level, APIs are just ways for one piece of code to talk to another piece of code. This could mean a computer talking to itself or a computer talking to another computer over a network. For example, a video game might consume an API that allows the code to draw graphics to the screen. We've seen a few methods available in the Express API, like `app.use` or `app.get`. These are just interfaces that you as a programmer can use to "talk to" other code.

There are also computer-to-computer APIs. These happen over a network, and usually over the Internet. These computers may be running different programming languages and/or different operating systems, so they've developed common ways to communicate. Some simply send plain text, while others might choose JSON or XML. They might send things over HTTP or over another protocol like FTP. Either way, both parties have to agree that they're going to send data a certain way. In this chapter, the APIs we create will use JSON.

We'll talk about APIs that interact you can build with Express. These APIs will take HTTP requests and respond with JSON data.

By the end of this chapter, programmers will be able to build applications that consume JSON APIs of your creation. We'll also aim to design *good* APIs. The core principle behind good API design is to do what developers consuming your API expect. Most of these expectations can be met by following the HTTP specification. Rather than instruct you to read a long, dry (but very interesting) specification document, I'll tell you the parts you need to know so that you can write a good API.

Just like the nebulous concepts of "good code" versus "bad code", there aren't a lot of hard lines in the sand here. A lot of this is open to your interpretation. You could come up with many examples where you might want to deviate from these established best practices, but remember: the goal is to do what other developers expect.

In this chapter, we'll learn:

- What an API is and isn't
- The fundamentals of building an API with Express
- HTTP methods and how they relate to common application actions
- How to create different versions of your API and why you want to do it
- How to properly use HTTP status codes

Let's get started.

6.1 A basic JSON API example

Let's talk about a simple JSON API and how it could be used so that we see a concrete example of the kind of thing we'll be building.

Let's imagine a simple API that takes a timezone string like `"America/Los_Angeles"` or `"Europe/London"` and returns a string that represents the current time in that timezone (like `"2015-04-07T20:09:58-07:00"`). Notice that these strings aren't things that a human would naturally type or be able to easily read—they're for a computer to understand.

Our API might accept an HTTP request to this URL:

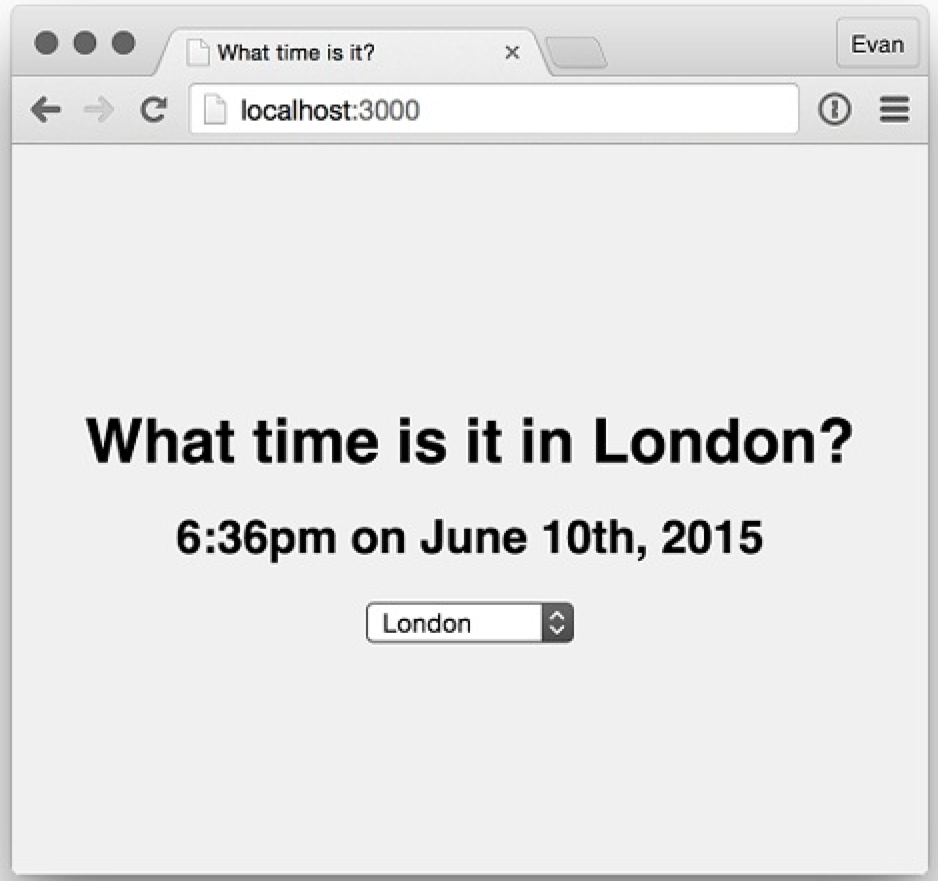
```
/timezone?tz=America+Los_Angeles
```

And our API server might respond with JSON, like this:

```
{
  "time": "2015-06-09T16:20:00+01:00",
  "zone": "America/Los_Angeles"
}
```

One could imagine writing simple applications that used this API. These applications could run on a variety of platforms, and as long as they communicated with this API and could parse JSON (which most platforms can), they can build whatever they want!

You could build a simple webpage that consumed this API, as shown in Figure 6.1. It might send AJAX requests to your server, parse the JSON, and display it in the HTML.



What time is it in London?

6:36pm on June 10th, 2015

Figure 6.1 A website that consumes our JSON API.

You could also build a mobile application, like Figure 6.2. It would make a request to our API server, parse the JSON, and display the results on the screen.

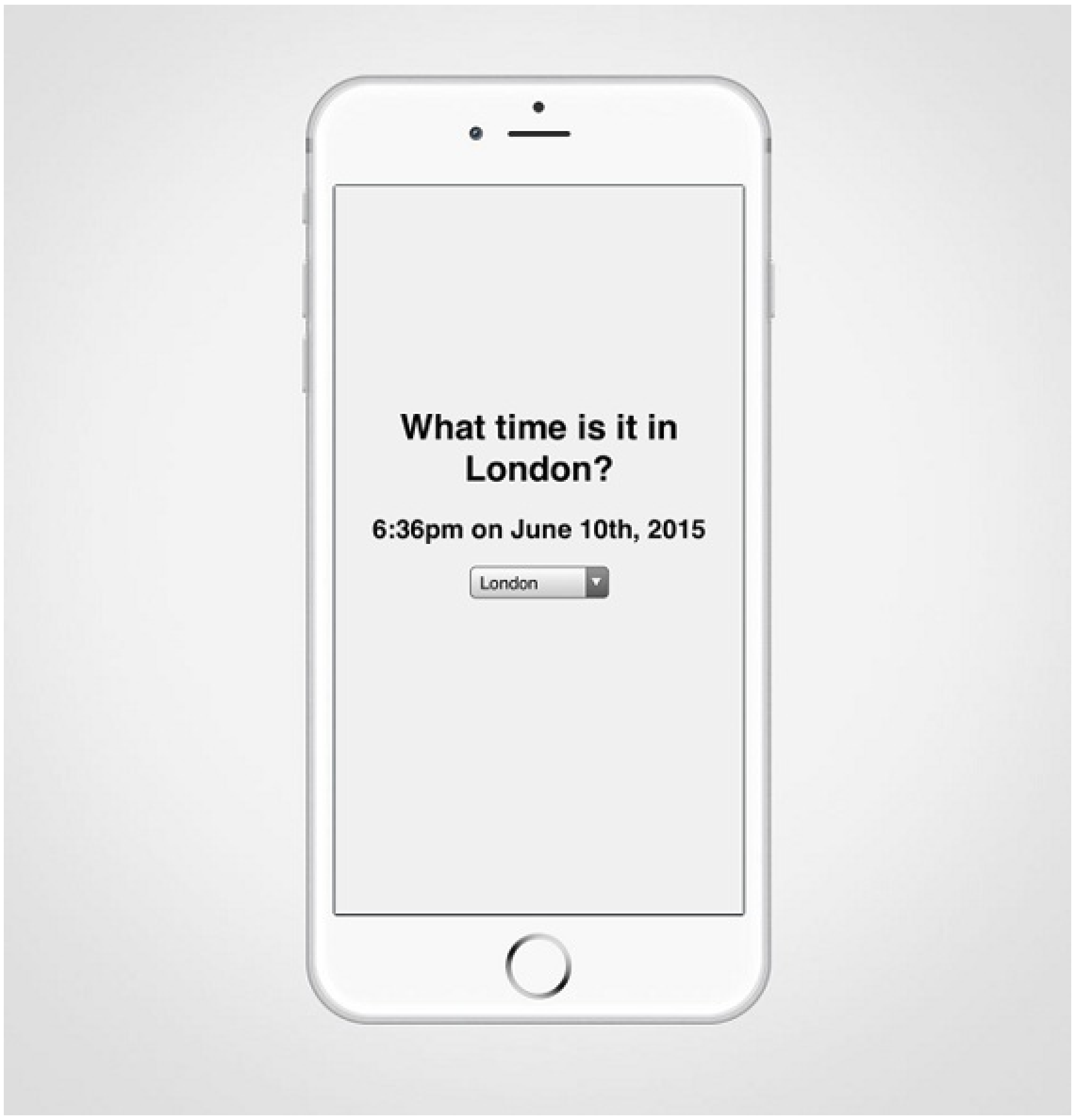
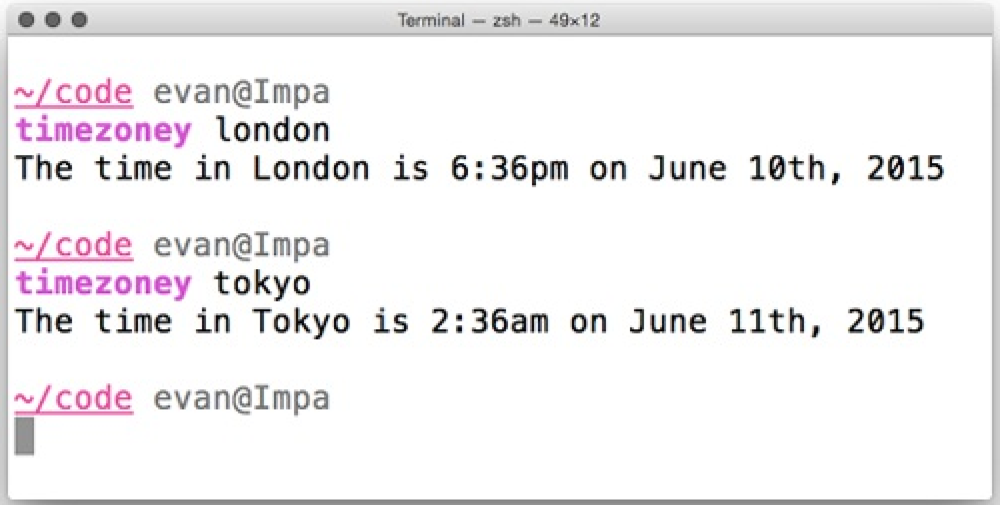


Figure 6.2 A mobile app that uses your API.

You could even build a command-line tool that runs in the terminal, like in Figure 6.3. Once again, it'd make a request to the API server, parse the JSON, and display the results for humans in the terminal.



```
~/code evan@Impa
timezoney london
The time in London is 6:36pm on June 10th, 2015

~/code evan@Impa
timezoney tokyo
The time in Tokyo is 2:36am on June 11th, 2015

~/code evan@Impa
```

Figure 6.3 Even terminal-based applications can consume a JSON API.

The point is this: if you make an API that takes requests from computers and spits out responses for computers (not humans!), you can build user interfaces atop that API. We did this in the previous chapter with the weather app—it used an API to get weather data and display it to the user.

6.2 A simple Express-powered JSON API

Now that we know what an API is, let's build a simple one with Express.

The fundamentals of an Express API are pretty straightforward: take a request, parse it, and respond with a JSON object and an HTTP status code. We'll use middleware and routing to take requests and parse them, and we'll use Express's conveniences to respond to requests.

> **NOTE** Technically, APIs don't have to use JSON—they can use other data interchange formats like XML or plain text. JSON has the best Express integration, plays nicely with browser-based JavaScript, and is one of the most popular API choices, so we'll use it here. You can use other formats if you want to!

Let's build a simple API that generates random integers. This might seem a bit of a contrived example, but we might want a consistent random number generator across multiple platforms (iOS, Android, web, and more) and we don't want to write the same code.

- Anyone who requests the API must send a minimum value and a maximum value.
- We'll parse those values, calculate our random number, and send it back as JSON.

You might think that JSON is overkill for this situation—why not stick to plain text?—but it'll help us learn how to do it and make it easy to expand our functionality later.

To build this project, we will:

1. Create a `package.json` to describe the metadata of our app
2. Create a file called `app.js` which will contain all of our code
3. In `app.js`, we'll create an Express application and attach a single route that gives a random number

Let's get started.

As usual, to start a project, make a new folder and create a package.json. You can create this file by running `npm init` or you can just manually type out the file. In any case, you'll want to create it and install Express. Your package.json should look *something* like this:

Listing 6.1 package.json for our random number project

```
{
  "name": "random-number-api",
  "private": true,
  "scripts": {
    "start": "node app"
  },
  "dependencies": {
    "express": "^4.12.3"       #A
  }
}
```

#A As always, your package version numbers may vary.

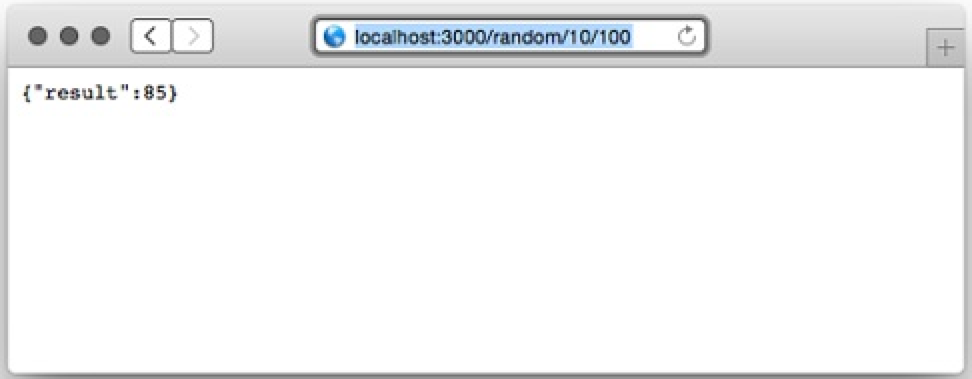
Next, we'll want to create app.js. Create it in the root of your project and put the following inside:

Listing 6.2 Our random number app

```
var express = require("express");

var app = express();

app.get("/random/:min/:max", function(req, res) {
  var min = parseInt(req.params.min);  #A
  var max = parseInt(req.params.max);  #A

  if (isNaN(min) || isNaN(max)) {                       #B
    res.status(400);                                    #B
    res.json({ error: "Bad request." });                #B
    return;                                             #B
  }                                                     #B

  var result = Math.round((Math.random() * (max - min)) + min);  #C

  res.json({ result: result });   #C
});

app.listen(3000, function() {
  console.log("App started on port 3000");
});
```

#A We pass two parameters in the URL of the request: min and max.
#B Do some error checking. If either of the numbers are malformed, we respond with an error.
#C Calculate and send the result as JSON.

If you start this app and visit http://localhost:3000/random/10/100, you'll see a JSON response with a random number between 10 and 100. It will look something like this:



```
{"result":85}
```

Figure 6.4 Testing your API in your browser. Try refreshing and you'll see different numbers!

Let's step through this code.

The first two lines simply require Express and create a new Express application as we've seen before.

Next, we create a route handler for GET requests. This will take requests like /random/10/100, or /random/50/52, but it will also handle requests like /random/foo/bar. We'll have to make sure that both fields are numbers, and we'll do that soon.

Next, we parse out the numbers using the built-into-JavaScript parseInt function. This function either returns a number or NaN. If either of the values are NaN, we show an error to the user. Let's look at these five lines in detail, because they're pretty important.

Listing 6.3 Drilling down into the error handler

```
if (isNaN(min) || isNaN(max)) {
  res.status(400);
  res.json({ error: "Bad request." });
  return;
}
```

The first line shouldn't be too new to you: it just checks if either of the numbers are NaN, meaning they're badly-formatted. If they are, we do three things:

1. Set the HTTP status code to 400. If you've ever seen a 404 error, this is just a variant: it signals that something about the user's request was bad. We'll talk about it more later in this chapter.
2. Send a JSON object. In this case, we send an object that has the error.
3. Return. If we *didn't* return, we'd continue onto the rest of the function and we'd send the request twice and Express would start throwing nasty errors.

Finally, we calculate the result and send it as JSON!

This is a pretty basic API, but it shows the fundamentals of building an API with Express: parsing requests, setting HTTP status codes, and sending JSON!

Now that we know the fundamentals, we can start learning more about building bigger, better APIs.

6.3 "Create, Read, Update, Delete" APIs

There's a common application pattern: Create, Read, Update, and Delete. It's shortened to CRUD, which is a fun word.

Lots of applications use CRUD. For example, imagine a photo-sharing app that has no user accounts; anyone can upload photos. Here's how you might envision that in CRUD style:

- Users can upload photos; this is the *create* step.
- Users can browse photos; this is the *read* part.
- Users can update photos, perhaps by giving them different filters or changing captions; this would be an *update*.
- Users can delete photos from the website. This would be, well, a *delete*.

You could imagine lots of your favorite applications fitting into this model, from photo sharing to social networks to file storage.

Before we can talk about how CRUD fits into APIs, we need to talk about something called HTTP methods, also known as HTTP verbs.

6.3.1 HTTP verbs (also known as HTTP methods)

The HTTP spec defines methods like this:

> *The Method token indicates the method to be performed on the resource identified by the Request-URI. The method is case-sensitive.*

Ugh, that's hard to read.

A human might understand it this way: a client sends an HTTP request to the server with a method. They can choose any method they want, but there are really only a handful that you use. The server sees that method and responds accordingly.

There's nothing baked into HTTP that prevents it from defining any method you want, but web applications typically use the following four:

1. GET is probably the most common HTTP method anyone uses. As the name suggests, it gets resources. When you load someone's homepage, you GET it. When you load an image, you GET it. GET methods shouldn't change the state of your app; the other methods do that.

   Idempotence is important to GET requests. "*Idempotent*" is a fancy word that means "doing it once should be no different than doing it many times". If you GET an image once and then refresh 500 times, the image shouldn't ever change. That's not to say that the response can never change—a page could change based on a changing stock price or a new time of day—but GETs shouldn't *cause* that change. That's idempotent.
2. POST is another common one, and is generally used to request a change to the state of the server. You POST a blog entry; you POST a photo to your favorite social network; you POST when you sign up for a new account on a website. POST is used to create records on servers, not modify existing records -- that's what PUT and DELETE are for, as discussed below.

   POST is also used for actions, like "buy this item".

   POST, unlike GET, is non-idempotent. That means that the state will change the first time you POST, and the second time, and the third time, and so on.

3. PUT has the worst name of the four, in my opinion; I think a name like "update" or "change" would suit it better. If I've published (POSTed) a job profile online and later want to update it, I would PUT those changes. I could PUT changes to a document, or to a blog entry, or something else. (We don't use PUT to delete entries, though; that's what DELETE is for, as we'll see next.)

   PUT has another interesting part; if you try to PUT changes to a record that doesn't exist, the server can (but doesn't have to) create that record. You probably wouldn't want to update a profile that doesn't exist, but you might want to update a page on a personal website whether it exists or not.

   PUT is idempotent, which wasn't immediately intuitive to me, but it eventually made sense. Let's say I'm "Evan Hahn" on a website but I want to change it to "Max Fightmaster". I don't PUT "change name *from* Evan Hahn *to* Max Fightmaster"; I PUT "change my name to Max Fightmaster; I don't care what it was before". This allows it to be idempotent. I could do this once or 500 times, and my name would still be Max Fightmaster. It is idempotent in this way.

4. DELETE is probably the easiest to describe. Like PUT, you basically specify "DELETE record 123". You could DELETE a blog entry, or DELETE a photo, or DELETE a comment. That's it!

   DELETE is idempotent in the same way that PUT is. Let's say I've accidentally published (POSTed) an embarrassing photo of me wearing a lampshade over my head. If I don't want it on there, I can DELETE it. Now it's gone! It doesn't matter whether I ask for it to be deleted once or 500 times; it's going to be gone. (Phew!)

There's nothing that strictly enforces these constraints -- you could theoretically use GET requests to do what POST requests should do, for example -- but it's bad practice and against the HTTP specification. It's not what people expect. Many browsers also have different behaviors depending on the type of HTTP request, so you always make an effort to use the right ones.

HTTP specifies a number of other verbs, but I've never had a need to stray very far from those four.

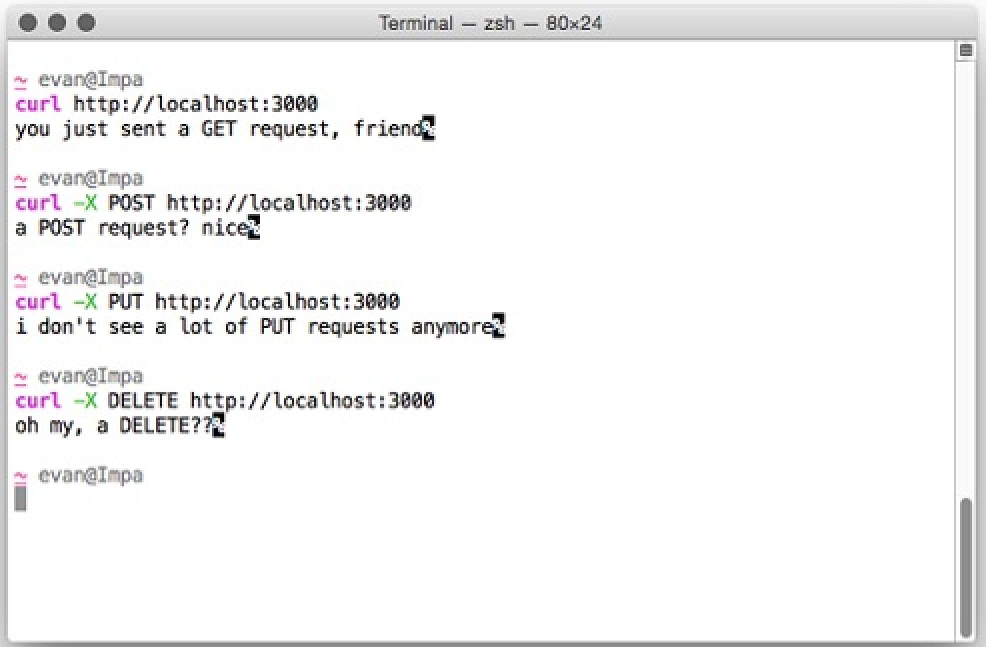
"VERBS" OR "METHODS"? The specification for HTTP 1.0 and 1.1 uses the word "method" when describing this concept, so I suppose that's technically correct. "Verb" is also used. For our purposes, I'll mostly call them "verbs" because that's what the Express documentation says. Know that you can use both (and that the nitpicky should call them "methods").

In Express, you've already seen how to handle different HTTP methods. To refresh your memory, here's a simple application that responds to each different method with a little message:

Listing 6.4 Handling different HTTP verbs

```
var express = require("express");

var app = express();

app.get("/", function(req, res) {
  res.send("you just sent a GET request, friend");
});

app.post("/", function(req, res) {
  res.send("a POST request? nice");
});

app.put("/", function(req, res) {
  res.send("i don't see a lot of PUT requests anymore");
});

app.delete("/", function(req, res) {
  res.send("oh my, a DELETE??");
});

app.listen(3000, function() {
  console.log("App is listening on port 3000");
});
```

If you start this application (if it's saved as `app.js`, run `node app.js`), you can use the handy cURL command-line tool to try sending different requests. cURL sends GET requests by default, but you can use its `-X` argument to send other verbs. For example, `curl -X PUT http://localhost:3000` will send a PUT request.



```
~ evan@Impa
curl http://localhost:3000
you just sent a GET request, friend%

~ evan@Impa
curl -X POST http://localhost:3000
a POST request? nice%

~ evan@Impa
curl -X PUT http://localhost:3000
i don't see a lot of PUT requests anymore%

~ evan@Impa
curl -X DELETE http://localhost:3000
oh my, a DELETE??%

~ evan@Impa
```

Figure 6.5 Using the cURL tool to send different requests to our server.

This should all be review from previous chapters: you can handle different HTTP methods with different handlers.

6.3.2 CRUD applications with HTTP methods

Let's recall our photo-sharing app. Here's how you might envision that in CRUD style:

- Users can upload photos; this is the create step.
- Users can browse photos; this is the read part.
- Users can update photos, perhaps by giving them different filters or changing captions; this would be an update.
- Users can delete photos from the website.

If you're like me, you didn't immediately see the connection between CRUD and the four main HTTP verbs I listed above. But if GET is for reading resources, and POST is for creating resources...woah! We see the following:

- Create = POST
- Read = GET

- Update = PUT
- Delete = DELETE

The four main HTTP methods lend themselves pretty well to CRUD-style applications, which are very common on the web.

POST versus PUT

There's a little bit of debate about which HTTP verbs correspond to which CRUD operations. Most people agree that Read == GET and Delete == DELETE, but Create and Update are a little murkier.

Because PUT can create records just like POST can, one could say that PUT better corresponds to Create. PUT can both Create and Update records, so why not put it in both spots?

Similarly, the PATCH method (which we haven't yet mentioned) sometimes takes the Update role. To quote the specification, "the PUT method is already defined to overwrite a resource with a complete new body, and cannot be reused to do partial changes." PATCH allows you to partially overwrite a resource. PATCH was only formally defined in 2010, so it's relatively new on the HTTP scene, which is why it's less used. In any case, some people think PATCH is better suited to Update than PUT.

Because HTTP doesn't specify this stuff too strictly, it's up to you to decide what you want to do. In this book, we'll be using the convention above, but know that the expectations are a little murky here.

6.4 API versioning

Let me walk you through a scenario.

You design a public API for your time zone app and it becomes a big hit. People all over the world are using it. People are using it to find times all across the globe. It's working well.

But, after a few years, you want to update your API. You've decided you want to change something, but there's a problem: if you change it, all of the people using your API will have to update their code. At this point, you might feel kind of stuck. What do you do? Do you make the changes you want to make and break old users, or does your API stagnate and never stay up-to-date?

There's a solution to all of this: version your API.

All you have to do is add some version information to your API. So a request that comes into this URL might be for version 1 of your API:

```
/v1/timezone
```

And a request coming into version 2 of your API might visit this URL:

```
/v2/timezone
```

This allows you to make changes to your API by simply making a new version! Now, if someone wants to upgrade to version 2, they'll do it by consciously changing their code, not having a version pulled out from under them.

Express makes this kind of separation pretty easy through its use of routers, which we saw in the previous chapter.

To create version 1 of your API, you can create a router that handles version 1 exclusively. The file might be called api1.js and look like this:

Listing 6.5 Version 1 of your API, in api1.js

```
var express = require("express");

var api = express.Router();   #A

api.get("/timezone", function(req, res) {       #B
  res.send("Sample response for /timezone");
});

api.get("/all_timezones", function(req, res) { #B
  res.send("Sample response for /all_timezones");
});

module.exports = api;  #C
```

#A Create a new Router, a mini-application.
#B These are just some example routes. You can add whatever routes or middleware you want to these routers.
#C Export the router so that other files can use it.

Notice that "v1" doesn't appear anywhere in the routes. To use this router in your app, you'll create a full application and use the router from your main app code. It might look like Listing 6.6.

Listing 6.6 The main app code in app.js

```
var express = require("express");

var apiVersion1 = require("./api1.js");   #A

var app = express();

app.use("/v1", apiVersion1);  #A

app.listen(3000, function() {
  console.log("App started on port 3000");
});
```

#A Require and use the router, like we saw in the previous chapter.

Then, many moons later, you decide to implement version 2 of your API. It might live in api2.js. It'd also be a router, just like api1.js. It might look like Listing 6.7.

Listing 6.7 Version 2 of your API, in api2.js

```
var express = require("express");

var api = express.Router();

api.get("/timezone", function(req, res) {       #A
  res.send("API 2: super cool new response for /timezone");
});
```

```
module.exports = api;
```

#A Once again, notice that these are just some example routes.

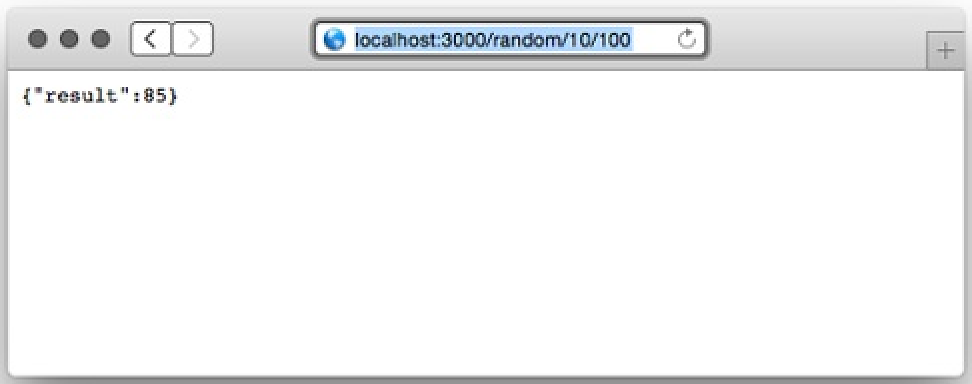
Now, to add version 2 of your API to the app, simply require and use it just like version 1:

Listing 6.8 The main app code in app.js

```
var express = require("express");

var apiVersion1 = require("./api1.js");
var apiVersion2 = require("./api2.js");   #A

var app = express();

app.use("/v1", apiVersion1);
app.use("/v2", apiVersion2);  #A

app.listen(3000, function() {
  console.log("App started on port 3000");
});
```

#A Note the two new lines. It's just like using version 1 of the router!

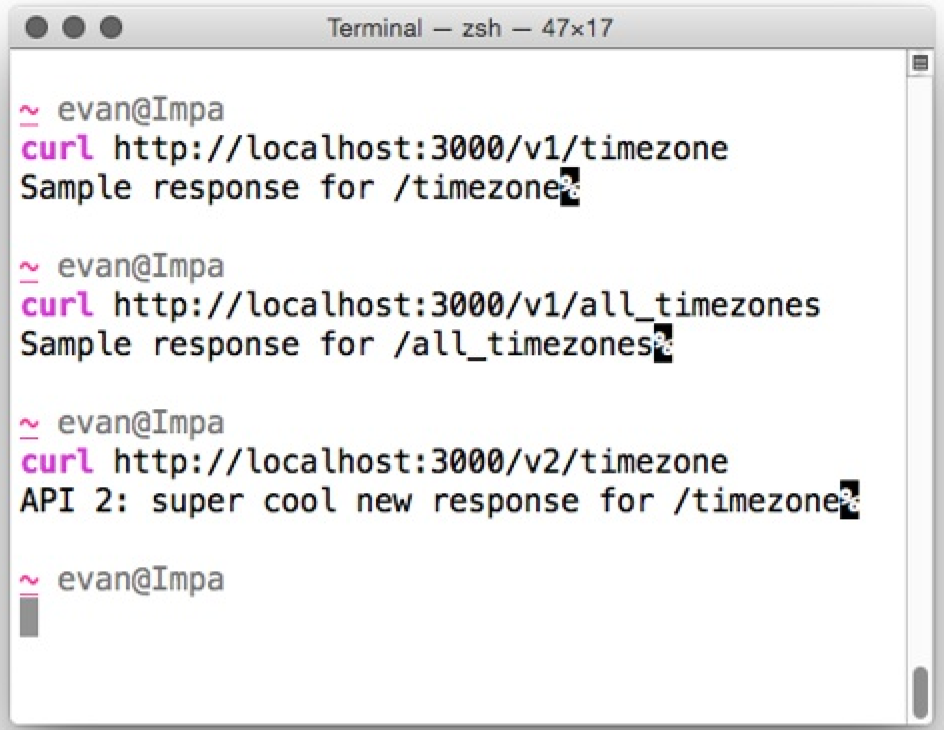
You can try visiting these new URLs in your browser to make sure that the versioned API works.



```
{"result":85}
```

Figure 6.6 Testing the two API versions in your browser.

You can also use the cURL tool to test your app at the command line.



```
~ evan@Impa
curl http://localhost:3000/v1/timezone
Sample response for /timezone%

~ evan@Impa
curl http://localhost:3000/v1/all_timezones
Sample response for /all_timezones%

~ evan@Impa
curl http://localhost:3000/v2/timezone
API 2: super cool new response for /timezone%

~ evan@Impa
```

Figure 6.7 Testing your versioned API using the cURL command-line tool.

As we saw in the previous chapter, routers let you segment different routes into different files. Versioned APIs are a great example of the utility of routers.

6.5 Setting HTTP status codes

Every HTTP response comes with an HTTP status code. The most famous one is 404, which stands for "Resource Not Found". You've likely seen 404 errors when visiting a URL that the server can't find—maybe you've clicked an expired link or typed a URL wrong.

While 404 is the most famous, 200 is perhaps the most common, which is simply defined as "OK". Unlike 404, you don't usually see the text "200" on the webpage when you're browsing the web. Every time you successfully load a webpage or an image or a JSON response, you'll probably get a status code of 200.

It turns out that there are a lot more HTTP status codes than 404 and 200, each with a different meaning. There are a handful of 100 codes (like 100 and 101), several in the 200s, 300s, 400s, and 500s. The ranges aren't "filled"--that is, the first four codes are 100, 101, 102, and then it skips all the way to 200.

Each range has a certain theme. Steve Losh sent a great tweet that summarizes them (which I had to paraphrase a bit), as told from the perspective of the server:

HTTP status ranges in a nutshell:

> 1xx: hold on
> 2xx: here you go
> 3xx: go away
> 4xx: you messed up
> 5xx: I messed up
>
> @stevelosh, https://twitter.com/stevelosh/status/372740571749572610

I love that summary. (The real one is a little more vulgar.)

Beyond the sixty-or-so codes in the specification (at https://tools.ietf.org/html/rfc7231#section-6), HTTP doesn't define any more. You can specify your own—HTTP allows it—but it typically isn't done. Remember the first principle of good API design: defining your own HTTP status codes wouldn't be what people expect. People expect you to stick to the usual suspects.

Wikipedia has a great list of every standard (and some nonstandard) HTTP response code at https://en.wikipedia.org/wiki/List_of_HTTP_status_codes, but there are a few that really pertain to building an API with Express. We'll go through each range (the 100s, then the 200s, etc) and explain some common HTTP codes you should be setting in your applications.

> **WHAT ABOUT HTTP 2?** Most HTTP requests are HTTP 1.1 requests, with a handful of them still using version 1.0. HTTP 2, the next version of the

standard, is slowly being implemented and rolled out across the web. Luckily for us, most of the changes happen at a low level and you don't have to deal with them. It does define one new status code—421—but that shouldn't affect you much.

But first, how do you set HTTP status codes in Express?

6.5.1 Setting HTTP status codes

By default, the status code is 200. If someone visits a URL where no resource is found and you don't have a handler for it, Express will send a 404 error. If you have some other error in your server, Express will send a 500 error.

But you want to have control of what status code you get, so Express gives it to you. Express adds a method called `status` to the HTTP response object. All you have to do is call it with the number of your status code and you'll be in business.

This method might be called like this inside of a request handler:

Listing 6.9 Setting the HTTP status code in Express

```
// …

res.status(404);

// …
```

This method is "chainable", so you can pair it with things like the `json` to set the status code and send some JSON in one line, as shown in Listing 6.10.

Listing 6.10 Setting the HTTP status code and sending some JSON

```
res.status(404).json({ error: "Resource not found!" });

// This is equivalent to:
res.status(404);
res.json({ error: "Resource not found!" });
```

The API isn't too complicated!

Express extends the "raw" HTTP response object that Node gives you. While you should use the Express way when you're using Express, you might be reading some code that sets the status code this way:

Listing 6.11 Setting the status code the "raw" way

```
res.statusCode = 404;
```

You sometimes see this code when reading through middleware, or when someone is using the "raw" Node APIs instead of the Express ones.

6.5.2 The 100 range

The 100 range is weird.

There are only two official status codes in the 100 range: 100 ("Continue") and 101 ("Switching Protocols"). You will likely never deal with these yourself. If you do, check the specification or the list on Wikipedia.

Look at that! Already one-fifth of the way through the status codes.

6.5.3 The 200 range

Steve Losh summarized the 200 range as "here you go". The HTTP spec defines several status codes in the 200 range, but four of them are by the most common.

200: "OK"

200 is the most common HTTP status code on the web by a long shot. HTTP calls status code 200 "OK", and that's pretty much what it means: everything about this request and response went through just fine.

Generally, if you're sending the whole response just fine and there aren't any errors or and redirects (which we'll see in the 300s section), then you'll send a 200 code.

201: "CREATED"

Code 201 is very similar to 200, but it's for a slightly different use case.

It's common for a request to create a resource (usually with a POST or a PUT request). This might be creating a blog post, sending a message, or uploading a photo. If the creation succeeds and everything's fine, and you'll want to send a 201 code.

This is a little bit nuanced, but it's typically the correct status code for the situation.

202: "ACCEPTED"

Just like 201 is a variant on 200, 202 is a variant of 201.

I hope I've beaten it into your head by now: asynchronousity is a big part of Node and Express. Sometimes, you'll asynchronously queue a resource for creation but it won't be created yet.

If you're pretty sure that the request is requesting to create a valid resource (perhaps you've checked that the data is valid) but you *haven't created it yet*, you can send a 202 status code. It effectively tells the client, "hey, you're all good, but I haven't made the resource yet."

Sometimes you'll want to send 201 codes and other times you'll want to send 202; it depends on the situation.

204: "NO CONTENT"

204 is the delete version of 201. When you create a resource, you typically send a 201 or a 202 message. When you delete something, you often don't have anything to respond with other than "yeah, this was deleted". That's when you typically send a 204 code. There are a few other times when you don't need to send any kind of response back, but deletion is the most common use case.

6.5.4 The 300 range

There are several status codes in the 300 range, but you'll really only set three of them, and they both involve redirects.

301: "MOVED PERMANENTLY"

HTTP status code 301 means "don't visit this URL any more; see another URL". 301 responses are accompanied with an HTTP header called `Location`, so you know where to redirect to.

You've probably been browsing the web and have been redirected in your life—this probably happened because of a 301 code. This is usually because the page has moved.

303: "SEE OTHER"

HTTP status code 303 is also a redirect, but it's a little bit different. Just like code 200 is for "regular" requests and 201 is for requests where a resource is created, 301 is for "regular" requests and 303 is for requests where a resource is created and you want to redirect to a new page.

307: "Temporary Redirect"

There's one last redirect status code: 307. Like the 301 code above, you've probably been browsing the web and been redirected because of a 307 code. They're similar, but they have an important distinction.

Where 301 signals "don't visit this URL *ever again*; see another URL", 307 signals "see another URL *just for now*". This might be used for temporary maintenance on a URL.

6.5.5 The 400 range

The 400 range is the largest, and it generally means that something about the request was bad. In other words, the client screwed something up and it's not the server's fault. There are a lot of different errors here.

401 AND 403 AUTHENTICATION ERRORS

There are two different errors for failed client authentication, and they're 401 ("Unauthorized") and 403 ("Forbidden"). The words "unauthorized" and "forbidden" sound pretty similar—what's the difference between those two?

In short, a 401 error is when the user isn't logged in at all. A 403 error is when the user is logged in as a valid user, but they don't have permissions to do what they're trying to do.

For example, imagine a website where you couldn’t see any of it unless you logged in. This website also has an administrator panel, but not all users can administer the site. Until you logged in, you'd see 401 errors. Once you logged in, you would stop seeing 401 errors. If you tried to visit the administrator panel as a non-admin user, you'd see 403 errors.

Send these response codes when the user isn't authorized to do whatever they're doing.

404: "NOT FOUND"

I don't think I have to tell you much about 404—you've probably run into it when browsing the web. One thing I found a little surprising about 404 errors is that you can visit a valid route but still get a 404 error.

For example, let's say you want to visit a user's page. The homepage for User #123 is at /users/123. But if you mistype and visit /users/1234 and no user exists with ID 1234, then you'd get a 404 error.

OTHER ERRORS

There are a lot of other client errors you can run into—far too many to enumerate here. Visit the list of status codes at https://en.wikipedia.org/wiki/List_of_HTTP_status_codes to find the right status code for you.

When in doubt, though, send a 400 "Bad Request" error. It's a generic response to any kind of bad request, and encompasses anything. Typically, it means that the request has malformed input—a missing parameter, for example. While there might be a status code that better describes the client error, 400 will do the trick when you're not sure which one to choose.

6.5.6 The 500 range

The final range in the HTTP specification is the 500 range, and while there are several errors in here, the most important one is 500: "Internal Server Error". Unlike 400 errors, which are the client's fault, 500 errors are the *server's* fault. This can be any number of things, from an exception to a broken connection to a database.

Ideally, you should never be able to cause a 500 error from the client—that means that your client can cause bugs in your server.

If you catch an error and it really does seem to be your fault, then you can respond with a 500 error. Unlike the rest of the status codes where you want to be as descriptive as possible, it's often better to be vague and say "Internal Server Error", that way hackers can't know where the weaknesses in your system lie. We'll talk much more about this in Chapter 9 when we talk about security.

6.6 Summary

In this chapter, you learned:

- How to use Express to build an API: parsing responses with routing and route parameters, choosing status codes with `res.status`, and sending JSON with `res.json`.
- The HTTP methods and how they respond to common CRUD operations (Create = POST, Read = GET, Update = PUT, Delete = DELETE).
- How to version your API using Express's routers: create routers for each API version and then `use` them in your main application
- What HTTP status codes are and what they mean. Remember Steve Losh's tweet: 100 means "hold on", 200 is "here you go", 300 is "go away", 400 is "you messed up", and 500 is "I messed up".

7 Views & Templates: Jade & EJS

In the previous chapters, we learned what Express is, how Express works, and how to use its routing feature. Starting in this chapter, we're going to stop learning about Express!

...well, okay, not exactly. We'll still be using Express to power our applications, don't worry. But as we've discussed throughout the book so far, Express is unopinionated and requires a lot of third-party accessories to make a full-fledged application. In this chapter and beyond, we'll start really digging into some of these modules, learning how they work, and how they can make your applications lovely.

In this chapter we will talk about views. Views are great. They give us a convenient way to dynamically generate content (usually HTML). We've seen a view engine before; EJS has helped us inject special variables into HTML. But while EJS gave us a conceptual understanding of views, we never really explored everything that Express (and the other view engines) had to offer. We'll learn the many ways to inject values into templates, see the features of EJS, Jade, and other Express-compatible view engines, and some subtleties in the world of views.

Let's get started.

7.1 Express's view features

Before I begin, let me define a term I'll be using a lot: view engine. When I say "view engine", I basically mean "module that does the actual rendering of views".

Jade and EJS are view engines, and there are many others.

American singer-songwriter India.Arie has an excellent song called "Brown Skin". About brown skin she sings "I can't tell where yours begins, I can't tell where mine ends". Similarly, when I first started using Express views, I was confused where Express ended and the view engines began. Luckily, it's not too difficult!

Express is unopinionated about which view engine you use. As long as the view engine exposes an API that Express expects, you're good to go. Express offers a convenience function to help you render your views; let's take a look.

7.1.1 A simple view rendering

We've seen simple examples of how to render views before, but just in case, here's an app that renders a simple EJS view:

Listing 7.1 Simple view rendering example

```
var express = require("express");  #A
var path = require("path");  #A

var app = express();  #A

app.set("view engine", "ejs");  #B
app.set("views", path.resolve(__dirname, "views"));  #C

app.get("/", function(req, res) { #D
    res.render("index");  });

app.listen(3000);  #E
```

#A First, we require what we need and create our application.
#B This tells Express that any file ending in ".ejs" should be rendered with whatever comes out of require("ejs"). This is a convention followed by some view engines; we'll later see how to use view engines that don't conform to this convention.
#C This tells Express where your views directory is. It happens to default to this, but I much prefer to be explicit. This also makes sure things work on Windows.

#D When we visit the root, we'll render a file called "index". This resolves to "views/index.ejs", which is then rendered with EJS.
#E This starts the server on port 3000!

Once you've done an `npm install` of EJS (and Express, of course), this should work! When you visit the root, it'll find `views/index.ejs` and render it with

EJS! 99% of the time, you'll do something like this; one view engine all the time. But things can get more complicated if you decide to mix things up.

7.1.2 A complicated view rendering

Here's a complex example of rendering a view from a response, using two different view engines: Jade and EJS. This should illustrate how crazy this can get:

Listing 7.3 Complex rendering example

```
var express = require("express");
var path = require("path");
var ejs = require("ejs");

var app = express();

app.locals.appName = "Song Lyrics";

app.set("view engine", "jade");
app.set("views", path.resolve(__dirname, "views"));
app.engine("html", ejs.renderFile);

app.use(function(req, res, next) {
  res.locals.userAgent = req.headers["user-agent"];
  next();
});

app.get("/about", function(req, res) {
  res.render("about", {
    currentUser: "india-arie123"
  });
});

app.get("/contact", function(req, res) {
  res.render("contact.ejs");
});

app.use(function(req, res) {
  res.status(404);
  res.render("404.html", {
    urlAttempted: req.url
  });
});

app.listen(3000);
```

Here's what happens when you call `render` in these three cases. While it looks complicated at a high level, it's just a number of straightforward steps:

1. Express builds up the context object every time you call render. These context objects will get passed to the view engines when it's time to render. These are effectively the variables available to views.
   It first adds all the properties from `app.locals`, an object available to every request. Then it adds all the properties in `res.locals`, overwriting anything added from `app.locals` if it was present. Finally, it adds the properties of the object passed to `render` (once again overwriting any previously-added properties). At the end of the day, if we visit `/about`, we'll create a context object with three properties: `appName`, `userAgent`, and `currentUser`. `/contact` will only have `appName` and `userAgent` in its context, while the 404 handler will have `appName`, `userAgent`, and `urlAttempted`.
2. Next, we decide whether view caching is enabled. "View caching" might sound like Express caches the entire view rendering process, but it doesn't; it only caches the lookup of the view file and its assignment to the proper view engine. For example, it will cache the lookup of `views/my_view.ejs` and figure out that this view uses EJS, but it won't cache the actual render of the view. A bit misleading!

   It decides whether view caching is enabled in two ways, only one of which is documented.

   The documented way: there's an option that you can set on the app. If `app.enabled("view cache")` is truthy, Express will cache the lookup of the view. By default, this is disabled in development mode and enabled in production, but you can change it yourself with `app.enable("view cache")` or `app.disable("view cache")`.

   The undocumented way: if the context object generated in the previous step has a truthy property called `cache`, then caching will be enabled for that view. This overrides any application settings. This enables you to cache on a view-by-view basis, but I think it's more important to know that it's there so that you can avoid doing it unintentionally!
3. Next, we have to look up where the view file resides and what view

engine to use. In this case, we want to turn "about" into `/path/to/my/app/views/about.jade` + Jade and "contact.ejs" into `/path/to/my/app/views/contact.ejs`+ EJS. The 404 handler should associate `404.html` with EJS by looking at our earlier call to `app.engine`. If we've already done this lookup before and view caching is enabled, we'll pull from the cache and skip to the final step. If not, we'll continue on.

4. If you don't supply a file extension (like with "about"), Express appends the default you specify. In this case, "about" becomes "about.jade" but "contact.ejs" and "404.html" stay the same. If you don't supply an extension and don't supply a default view engine, Express will throw an error. Otherwise, it'll continue on.
5. Now that we definitely have a file extension, Express looks at the extension to determine which engine to use. If it matches any engine you've already specified, it'll use that. In this case, it'll match Jade for `about.jade` because it's the default. `contact.ejs` will try to `require("ejs")` based on the file extension. We explicitly assigned `404.html` to EJS's `renderFile` function, so it'll use that.
6. Express looks the file up in your views directory. If it doesn't find the file, it throws an error, but it'll continue if it finds something.
7. If view caching is enabled, we cache all this lookup logic for next time.
8. Finally, we render the view! This calls out to the view engine and is literally one line in Express's source code. This is where the view engine takes over and produces actual HTML (or whatever you'd like).

This turns out to be a bit hairy, but the 99% case is "I pick one view engine and stick with it", so you're likely to be shielded from most of this complexity.

Rendering non-HTML views

Express's default content-type is HTML, so if you don't do anything special, res.render will render your responses and send them to the client as HTML.

Most of the time, I find this to be enough. But it doesn't have to be this way! You can render plain text, XML, JSON, or whatever you want. Just change the content-type by changing the parameter to res.type:

```
app.get("/", function(req, res) {
  res.type("text");
  res.render("myview", {
    currentUser: "Gilligan"
  });
});
```

There are often better ways to render some of these things—`res.json`, for example, should be used instead of a view that renders JSON—but this option is totally available!

7.1.3 Making all view engines compatible with Express: Consolidate.js

We've talked about some view engines like EJS and Jade already, but there are plenty more that you might want to choose. You might've heard of Mustache, Handlebars, or Underscore.js's templating. You might also want to use a Node port of other templating languages like Jinja2 or HAML.

Many of these view engines will "just work" with Express, like EJS and Jade. Others, however, don't have an Express-compatible API, and need to be wrapped in something Express can understand.

Enter Consolidate.js (at https://github.com/tj/consolidate.js), a library that wraps a ton of view engines to be compatible with Express. It has support for the classics like EJS, Jade, Mustache, Handlebars, and Hogan. It supports a ton of others, too, in case you're using a more obscure/hipster view engine. You can see the whole list of supported engines on the project's page.

For example, let's say you're using Walrus, a JavaScript view engine that's not compatible with Express out of the box. We'll need to use Consolidate to make this compatible with Express.

After installing Walrus and Consolidate (with `npm install walrus consolidate`), you'll be able to use Walrus with Express!

Listing 7.4 Rendering with Walrus

```
var express = require("express");
var engines = require("consolidate"); #A
var path = require("path");
var app = express();

app.set("view engine", "wal");  #B
app.engine("wal", engines.walrus); #C
app.set("views", path.resolve(__dirname, "views")); #D

app.get("/", function(req, res) {  #E
   res.render("index"); });

app.listen(3000);
```

#A First, we must require the Consolidate library. For readability, we place it in a variable called "engines".
#B Next, we specify .wal files as our default view file extension.
#C Here, we associate.wal files with the Walrus view engine.
#D As usual, we specify our views directory.
#E Finally, we render the view! This will render views/index.wal.

I recommend using Consolidate instead of trying to wrangle non-compatible view engines yourself.

7.2 Everything you need to know about EJS

One of the simplest and most popular view engines out there is called EJS, for "Embedded JavaScript". It can do templating for simple strings, HTML, plain text—you name it—it lightly integrates itself with whatever tool you use. It works in the browser and Node.js. If you've ever used ERB from the Ruby world, EJS is very similar. In any case, it's pretty simple!

THERE ARE TWO VERSIONS OF EJS OUT THERE There are actually two versions of EJS maintained by two different groups of people. They're similar, but not identical. The one we'll be using is by TJ Holowaychuck, the creator of Express. If you look for a package called "ejs" on npm, this is the one you'll find. But if you visit http://embeddedjs.com/, you'll find a very similar library claiming with the same name. A lot of the functionality is the same, but it's a different library last updated in 2009. It doesn't work in Node, and it's even got some debatably-sexist sentences in its documentation; avoid it!

7.2.1 The syntax of EJS

EJS can be used for templating HTML, but it can be used for anything. Let's take a look at a short bit of EJS, and what that looks like when you render it.

Listing 7.5 An EJS template

```
Hi <%= name %>!
You were born in <%= birthyear %>, so that means you're[CA]
<%= (new Date()).getFullYear() - birthyear %> years old.
<% if (career) { -%>
  <%=: career | capitalize %> is a cool career!
<% } else { -%>
  Haven't started a career yet? That's cool.
<% } -%>
Oh, let's read your bio: <%- bio %> See you later!
```

If we passed the following context to EJS...

Listing 7.6 An EJS context

```
{
  name: "Tony Hawk",
  birthyear: 1968,
  career: "skateboarding",
  bio: "<b>Tony Hawk</b> is the coolest skateboarder around."
}
```

Then we'd get the following result (as of 2015, anyway):

```
Hi Tony Hawk!
You were born in 1968, so that means you're 47 years old.
Skateboarding is a cool career!
Oh, let's read your bio: Tony Hawk is the coolest skateboarder around. See you later!
```

This little example shows four major features of EJS: JavaScript that's evaluated, escaped, and printed, JavaScript that's evaluated but not printed, JavaScript that's evaluate and printed (but not escaped for HTML), and filters.

You can print the results of JavaScript expressions in two ways, as we see. `<% expression %>` prints the result of the expression, while `<%= expression %>` prints the result of the expression and escapes any HTML entities that might be inside. In general, I'd recommend using the latter option when you can, because it's more secure.

You can also run arbitrary JavaScript and keep it from being printed. This is useful for things like loops and conditionals, as we see in the above example. This is done with `<% expression %>`. As you can see, you can use brackets to group loops and conditionals across multiple lines. You can also avoid adding extraneous newlines with `<% expression -%>` (note the hyphen at the end).

Finally, appending a colon (`:`) to an output will allow filters to be applied. Filters take the output of an expression and filter it to change the output. In the above example, we use the capitalization filter, but there are plenty of others and you can define your own (as we'll see in just a moment!).

> **NOTE** If you want to play around with EJS, I made "Try EJS" (at https://evanhahn.github.io/try-EJS/), a simple browser app to play around with EJS in your browser. I'll admit it's not polished, but it's sufficient for just playing around with EJS in your browser and seeing the rendered output.

INCLUDE-ING OTHER EJS TEMPLATES WITHIN YOUR OWN

EJS also lets you include other EJS templates, too. This is incredibly useful for a variety of reasons. You can add headers and footers to pages, split out common widgets, and more! If you find yourself writing the same code several times, it might be time to use EJS's include feature.

Let's look at two examples.

First, let's imagine you have pages that all share the same header and footer. Rather than duplicate everything over and over again, you could create a header EJS file, a footer EJS file, and then your pages that go "between" the header and footer.

Here's how a header file (saved at `header.ejs`) might look:

Listing 7.7 A header EJS file

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="/the.css">
  <title><%= appTitle %>/title>
</head>
<body>
  <header>
    <h1><%= appTitle %>
  </header>
```

Then, you'd define a footer in footer.ejs:

Listing 7.8 A footer EJS file

```
<footer>
  All content copyright <%= new Date().getFullYear() %> <%= appName %>.
</footer>
</body>
</html>
```

Now that you've defined your header and footer, you can include it in sub-pages pretty easily!

Listing 7.9 Including a header and footer from EJS

```
<% include header %>
  <h1>Welcome to my page!</h1>
  <p>This is a pretty cool page, I must say.</p>
<% include footer %>
```

We use include to, well, include other EJS files. Notice that we don't use `<%= ... %>` or `<%- ... %>`; everything is finally printed by EJS, not you.

We could also imagine ourselves using this to build a widget. For example, let's say we had a widget that showed user profiles. Given an object called user, this template would spit out some HTML for that user. Here's how `userwidget.ejs`might look:

Listing 7.10 A user widget in userwidget.ejs

```
<div class="user-widget">
  <img src="<%= user.profilePicture %>">
  <div class="user-name"><%= user.name %></div>
  <div class="user-bio"><%= user.bio %></div>
</div>
```

Now we can use that template when rendering the current user...

Listing 7.11 Rendering a user widget for the current user

```
<% user = currentUser %>
<% include userwidget %>
```

...or when rendering a list of users.

Listing 7.12 Rendering a user widget many times

```
<% userList.forEach(function(user) { %>
  <% include userwidget %>
<% } %>
```

EJS's include is versatile; it can be used to create templates or to render sub-views many times.

ADDING YOUR OWN FILTERS

There are 22 built-in filters, ranging from mathematic operations to array/string reversal to sorting. They're often enough for your needs, but sometimes you'll want to add your own.

Assuming you've required EJS into a variable called `ejs`, you simply add a property to `ejs.filters`. If we're frequently summing arrays, we might find it useful to make our own custom "array summer" filter.

Here's how we might add such a filter:

Listing 7.13 Add an EJS filter to sum an array

```
ejs.filters.sum = function(arr) {
  var result = 0;
  for (var i = 0; i < arr.length; i ++) {
    result += arr[i];
  }
  return result;
};
```

Now you can use it just like any other filter!

Listing 7.14 Using our new EJS sum filter

```
<%=: myarray | sum %>
```

Pretty simple! There are lots of filters you could dream up—code them as you need them!

7.3 Everything you need to know about Jade

View engines like Handlebars, Mustache, and EJS don't completely replace HTML—they augment it with some new features. This is really nice if you have designers, for example, who've already learned HTML and don't want to learn a whole new language. It's also useful for non-HTML-like templating solutions. If you're in this sort of situation, Jade is probably the wrong choice.

But Jade also promises some other features. It allows you to write far fewer lines of code and the lines you write are much prettier. Doctypes are easy; tags are nested by indentation, not close tags. It's got a number of EJS-style features built into the language, like conditionals and loops. It's more to learn, but more powerful.

7.3.1 The syntax of Jade

Languages like HTML are nested. There's a root element (`<html>`) and then various sub-elements (like `<head>` and `<body>`), which each have their own sub-elements...and so on. HTML and XML choose to have an open (`<a>`) and a close (`</a>`) for each element.

Jade takes a different approach by using indentation and a different syntax for HTML. Here's a very simple webpage that uses Jade:

Listing 7.15 A simple Jade example

```
doctype html
html(lang="en")   #A
  head
    title Hello world!
  body
    h1 This is a Jade example
    #container    #B
      p Wow.
```

#A Adding attributes to elements looks like function calls. (They look a lot like keyworded method calls in Python, if you're familiar with that!)
#B No element is specified, so this is a div.

This turns into the following HTML:

Listing 7.16 Listing 7.15 rendered as HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Hello world!</title>
  </head>
  <body>
    <h1>This is a Jade example</h1>
    <div id="container">
      <p>Wow.</p>
    </div>
  </body>
</html>
```

You can play around with Jade on the project's homepage at http://jade-lang.com/—try experimenting to see what happens!

7.3.2 Layouts in Jade

Layouts are an important feature of any templating language. They allow you to include, in one form or another, other HTML. This allows you to define your header and footer once, and then include them on pages where you need them.

A very common case is to define a layout file for your pages. That way, everything can have a consistent header and footer while allowing the content to change per page.

First, we define the "master" layout. This is the Jade common to every page, like a header and footer. This master layout defines empty blocks that are filled in by any pages that use this master layout. Let's take a look at an example.

First, let's define a simple layout file. This file will be shared by all of our pages.

Listing 7.15 A simple layout file for Jade

```
doctype html
html

  head
     meta(charset="utf-8")
     title Cute Animals website
     link(rel="stylesheet" href="the.css")

     block header  #A

  body

     h1 Cute Animals website

     block body  #B
```

#A In the parent layout file, we define a "header" block and "body" block. These will be used by anyone who extends this layout.

Notice how we've defined two blocks with block header and block body. These will get filled in by Jade files using this layout. Save that file into layout.jade. We can use these in "real" pages that use this layout, like this:

Listing 7.16 Using a Jade layout file

```
extends layout.jade
block body
  p Welcome to my cute animals page!
```

That will render the following HTML:

Listing 7.17 The output of using a Jade layout

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Cute Animals website</title>
    <link rel="stylesheet" href="the.css">
  </head>
  <body>
    <h1>Cute Animals website</h1>
    <p>Welcome to my cute animals page!</p>
  </body>
</html>
```

Notice that we simply put something in a block when we extend a layout and it's magically inserted! Also notice that we don't have to use a block just because it's defined -- we never touch the header block because we don't need to.

If we wanted to, we could define another page that uses this layout very easily.

Listing 7.18 Using a Jade layout file again

```
extends layout.jade
```

```
block body
  p This is another page using the layout.
  img(src="cute_dog.jpg" alt="A cute dog!")
  p Isn't that a cute dog!
```

Layouts let us separate out common components, which means we don't have to repeat the same code over and over again.

7.3.3 Mixins in Jade

Jade has another cool feature called mixins. Mixins are basically functions that you define in your Jade file to cut down on repetitive tasks.

Let's re-implement the users widget example from the EJS section. We'll make a widget that's given an object called user and returns an HTML widget for that user. Here's we might do that:

Listing 7.19 A user widget mixin

```
mixin user-widget(user)
  .user-widget
    img(src=user.profilePicture)
    .user-name= user.name
    .user-bio= user.bio

//- Render the user widget for the current user
+user-widget(currentUser)

//- Render the user widget for a bunch of users
- each user in userList
  +user-widget(user)
```

This would render the user widget for the `currentUser` and for every other user in the `userList`. No duplicated code for us!

That's all that we'll look at with Jade. For more about Jade's syntax, you can look at Jade's reference documentation at http://jade-lang.com/reference/.

7.4 Summary

In this chapter, we've learned the following:

- Express's view system. We learned how to pass variables to views to dynamically generate HTML, and we've learned how the view engine works.
- The EJS templating language, to dynamically generate HTML with little bits of JavaScript.
- The Jade templating language, to reimagine HTML and dynamically generate it with a whole new language.

8 Persisting your data with MongoDB

I have three favorite chapters in this book.

You've already passed my first favorite: Chapter 3, where we discuss the foundations of Express. I like that chapter because the goal is to really *explain* Express. In my opinion, it's the most important chapter of the book, because it explains the framework conceptually.

Chapter 10 is the second of my three favorites. As you'll see, it discusses security, and I love putting a hacker hat on and trying to break Express applications. It's a lot of fun (and, incidentally, terribly important).

This chapter is my final favorite. Why? Because after this chapter, your applications will feel *real*. No more dinky example apps. No more data that quickly disappears. Your Express applications will have user accounts, blog posts, friend requests, calendar appointments...all with the power of data persistence.

Nearly every application has some kind of data, be it blog posts or user accounts or cat pictures. As we've discussed, Express is generally an unopinionated framework. Fitting in with this unopinionated mantra, Express doesn't dictate how you store your data. So how should we approach it?

You could store your application's data in memory, by simply setting variables. For example, Chapter 3's guestbook example stored the guestbook entries in an array. While this is useful in very simple cases, it's got a number of disadvantages. For one, if your server stops (either because you manually stop it or because it crashes), your data is gone! And if you grow to hundreds of millions of data points, you'll run out of memory. This method also runs into issues when you have multiple servers running your application, because data can be on one machine but not the other.

You could try to store your application's data in files, by writing to a file or multiple files. This *is* how many databases work internally, after all. But that leaves you to figure out how to structure and query that data. How do you save your data? How do you efficiently get data out of those files when you need it? You might wind up building a database of your own, which is a huge headache. And once again, this doesn't magically work with multiple servers.

We'll need another plan. And that's why we choose software designed for this purpose: a database. Our database of choice is something called MongoDB.

In this chapter, we'll learn the following:

- How MongoDB works
- How to use Mongoose, an official MongoDB library
- How to securely create user accounts
- How to use Passport for authentication

Let's get real.

8.1 Why MongoDB?

MongoDB (often shortened to Mongo) is a popular database that's wiggled its way into the hearts of many Node developers. Its pairing with Express is beloved enough to have spawned the acronym "MEAN", for MongoDB, Express, Angular (a front-end JavaScript framework), and Node. In this book, we'll be discussing everything but the "A" of that acronym...the "MEN" stack, if you will.

At this point, you may be saying, "There are a lot of choices for databases out there, like SQL or Apache Cassandra or Couchbase. Why choose MongoDB?" That's a good question!

In general, Web applications store their data in one of two kinds of databases: *relational* and *non-relational*.

Typically, *relational databases* are a lot like spreadsheets. Their data is structured, and each entry is generally a row in a table. They are a bit like strongly-typed languages such as Java, where each entry must fit into rigid requirements (called a schema). Most relational databases can be controlled with some derivative of SQL, the Structured Query Language; you likely have heard of MySQL or SQL Server or PostgreSQL. "Relational databases" and "SQL databases" are often terms used interchangeably.

Non-relational databases, on the other hand, are often called "NoSQL" databases. (In reality, "NoSQL" just means "anything that isn't SQL", but it tends to refer to a certain class of database.) I like to imagine "NoSQL" as both a different technology and a fist-up cry against the status quo. Perhaps "NoSQL" is tattooed on a protester's arm. In any case, it's different from relational databases in that it is generally not structured like a spreadsheet. They are generally a bit less rigid than SQL databases. This is very much like JavaScript in this way; JavaScript is generally less rigid. In general, NoSQL databases "feel" a bit more like JavaScript than SQL databases.

For this reason, we choose a NoSQL database. The NoSQL database we'll choose is called MongoDB. But why choose that?

For one, MongoDB is popular. That isn't in itself a merit, but this has a few benefits. You won't have trouble finding help online. It's also useful to know; it's used in lots of places by lots of people. Mongo is also a mature project. It's been around since 2007 and is trusted by companies like eBay, Craigslist, and Orange. You won't be using buggy, unsupported software.

Mongo is popular in part because it's mature, feature-filled, and reliable. It's written in performant C++ and is trusted by lots and lots of users.

While Mongo isn't written in JavaScript, its native shell uses JavaScript. That means that when you open Mongo to play around in the command line, you send it commands with JavaScript. It's pretty nice to be able to "talk to" the database with a language you're already using!

I also chose Mongo for this chapter because I think it's easier to learn than SQL for a JavaScript developer. SQL is a powerful programming language unto itself, but you already know JavaScript!

I hardly believe that Mongo is the right choice for all Express applications. Relational databases are incredibly important and can be used well with Express, and other NoSQL databases like CouchDB are also powerful. But Mongo fits well with the Express ecosystem and is relatively easy to learn (compared to SQL), which is why I choose it for this chapter.

NOTE If you're like me, you know SQL and want to use it for use SQL for some Express projects. This chapter will discuss Mongo, but if you're looking for a helpful SQL tool, check out Sequelize at http://sequelizejs.com/. It interfaces with many SQL databases and has a number of helpful features.

In this chapter, we'll deal heavily with a module called Mongoose; for your reference as you read, Mongoose is to MongoDB as Sequelize is to SQL. Keep that in mind if you want to use SQL!

8.1.1 How Mongo works

Before we start, let's talk about how Mongo works.

Most applications have one database, like MongoDB. These databases are hosted by servers. A Mongo server can have many databases on it, but there is generally one database per application. If you're only developing one application on your computer, you'll likely only have one Mongo database. (These databases can be replicated across multiple servers, but you treat them as if it's one database.)

To access these databases, you'll run a MongoDB server. Clients will talk to these servers, viewing and manipulating the database. There are client libraries for most programming languages; these libraries are called drivers and let you talk to the database in your favorite programming language. In this book, we'll be using the Node.js driver for MongoDB.

Every database will have one or more collections. I like to think of collections as fancy arrays. A blog application might have a collection for blog posts, or a social network might have a collection for user profiles. They're like arrays in that they're just giant lists, but you can also query them ("give me all users in this collection older than age 18", for example) much more easily than arrays.

Every collection will have any number of documents. Documents aren't technically stored as JSON, but you can think of them that way; they're basically objects with various properties. Documents are things like users and blog posts; there is one document per thing. Documents don't have to have the same properties, even if they're in the same collection—you could theoretically have a collection filled with completely different objects (although you seldom do this in practice).

Documents look a lot like JSON, but they're technically something called Binary JSON, or BSON. You almost never deal with BSON directly; rather, you'll translate to and from JavaScript objects. The specifics of BSON encoding and decoding are a little different from JSON. BSON also supports a few types that JSON does not, like dates, timestamps, and undefined values.

Here's a diagram that shows how things are put together:

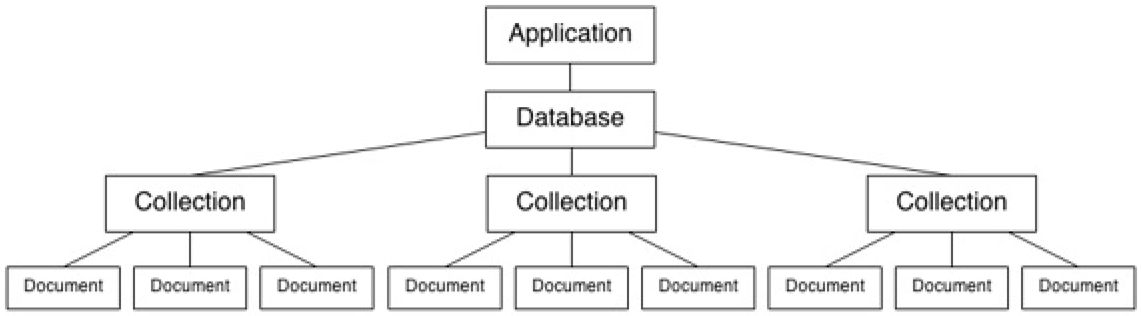


Figure 8.1 Hierarchy of Mongo's databases, collections, and documents

One last important point: Mongo adds a unique `_id` property to every document. Because these IDs are unique, two documents are the same if they have the same `_id` property, and you can't store two documents with the same ID in the same collection. This is a miscellaneous point but an important one that we'll come back to!

8.1.2 For you SQL users out there…

If you come from a relational/SQL background, many of Mongo's structures map one-to-one with structures from the SQL world. (If you're not familiar with SQL, you can skip this section!)

Documents in Mongo correspond with rows or records in SQL. In an application with users, each user would correspond to one document in Mongo or one row in SQL. In contrast to SQL, Mongo doesn't enforce any schema at the database layer, so it's not invalid in Mongo to have a user without a last name or an email address that's a number.

Collections in Mongo correspond to SQL's tables. Mongo's collections contain many documents, where SQL's tables contain many rows. Once again, Mongo's collections don't enforce a schema, which is unlike SQL. In addition, these documents can embed other documents, unlike in SQL—blog posts could contain the comments, which would likely be two tables in SQL. In a blog application, there would be one Mongo collection for blog posts or one SQL table. Each Mongo collection contains many documents, where each SQL table contains many rows or records.

Databases in Mongo are very similar to databases in SQL. Generally, there is one database per application. Mongo databases can contain many collections, where SQL databases can contain many tables. A social networking site would likely have just one of these databases in SQL, Mongo, or another type of database.

For a full list of "translations" from SQL terminology to MongoDB terminology (queries, too!), check out the official SQL to MongoDB Mapping Chart at http://docs.mongodb.org/manual/reference/sql-comparison/index.html.

8.1.3 Setting up Mongo

You'll want to install Mongo locally so that you can use it while you're developing.

If you're on OSX and aren't sure you want to use the command line, I'm a big fan of Mongo.app. Instead of wrangling the command line, you simply launch an application that runs in the menu bar at the top right of your screen. You can tell when it's running and when it's not, easily start up a console, and shut it down effortlessly. You can download it at http://mongoapp.com/.

If you're on OSX and would prefer to use the command line, you can use the Homebrew package manager to install MongoDB with a simple `brew install mongodb`. If you're using MacPorts, `sudo port install mongodb` will do the job. If you're not using a package manager and you don't want to use Mongo.app, you can download it from the MongoDB downloads page at http://www.mongodb.org/downloads.

If you're on Ubuntu Linux, Mongo's website has helpful instructions at http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/. If you're using a Debian distribution like Mint (or Debian!), check out the official documentation athttp://docs.mongodb.org/manual/tutorial/install-mongodb-on-debian/. Other Linux users can check out http://docs.mongodb.org/manual/tutorial/install-mongodb-on-linux/.

If you're a Windows user or on any of the OSes I didn't mention above, the MongoDB downloads page will help you. You can either download it from their website or scroll down to the bottom of that page to see other package managers that have Mongo. Take a look at http://www.mongodb.org/downloads. If you can, make sure you download the 64-bit version; the 32-bit version has a limit on storage space.

Throughout this book, we'll assume that your MongoDB database is at `localhost:27017/test`. Port 27017 is the default port and the default database is one called "test", but your results may vary. If you can't connect to your database, check your specific installation for help.

8.2 Talking to MongoDB from Node with Mongoose

We'll need a library that will let us talk to MongoDB from Node, and therefore from Express. There are a number of lower-level modules, but we'd like something easy-to-use and feature-filled. What should we use?

Look no further than Mongoose (at http://mongoosejs.com/), an officially-supported library for talking to MongoDB from Node.js. To quote its documentation:

> Mongoose provides a straight-forward, schema-based solution to modeling your application data and includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

In other words, Mongoose gives us much more than simply talking to the database. Let's learn how it works by creating a simple website with user accounts.

8.2.1 Setting up your project

In order to learn the topics in this chapter, we'll develop a very simple social network application. This app will let users register new profiles, edit those profiles, and browse each others' profiles. We'll call it "Learn About Me", for lack of a creative name. We'll call it "LAM" for short.

Our site will have a few pages on it:

- The homepage, which will list all users. Clicking on a user in the list will take you to their profile page.
- The profile page will show the user's display name (or username if no display name is defined), the date they joined the site, and their biography.
- The user will be able to sign up for a new account, log into accounts, and log out.
- After signing up, users will be able to edit their display names and biographies, but only when they're logged in.

As always, create a new directory for this project. As always, we'll need to create a package file with metadata about our project and its dependencies. Create a `package.json` file and put this inside:

Listing 8.1 package.json for LAM

```
{
  "name": "learn-about-me",
  "private": true,
  "scripts": {
    "start": "node app"
  },
  "dependencies": {
    "bcrypt-nodejs": "0.0.3",  #B
    "body-parser": "^1.6.5",
    "connect-flash": "^0.1.1",
    "cookie-parser": "^1.3.2",
    "ejs": "^1.0.0",
    "express": "^4.8.5",
    "express-session": "^1.7.6",
    "mongoose": "^3.8.15",
    "passport": "^0.2.0",
    "passport-local": "^1.0.0"
  }
}
```

#B There's a different module called "bcrypt" which builds a bunch of C code. It is probably faster but it can be a little harder to install. If you need speed, it's a drop-in replacement.

After you've created this file, run `npm install` to install our slew of dependencies. We'll see what each of these dependencies do as we chug through the rest of the chapter, so if any of them are unclear, don't worry! As usual, we've set this up so that `npm start` will start our app (which we'll save into `app.js`).

Now it's time to start putting things into databases!

8.2.2 Creating a user model

As we've discussed, MongoDB stores everything in BSON, which is a binary format. A simple "hello world" BSON document might look like this internally:

```
\x16\x00\x00\x00\x02hello\x00\x06\x00\x00\x00world\x00\x00
```

A computer can deal with all that mumbo-jumbo, but that's hard to read for humans like us. We want something that's more amenable to us, which is why developers have created the concept of a *database model*. A model is a representation of a database record as a nice object in your programming language of choice. In this case, our models will be JavaScript objects.

Models can serve as a simple object that stores database values, but they often have things like data validation, extra methods, and more. As we'll see, Mongoose has a lot of those features.

In this example, we'll be building a model for users. Before we start, we should consider the properties User objects should have:

- Username, a unique name. This will be required.
- Password. This will also be required.
- Time joined, a record of when the user joined the site.
- Display name, name that's displayed instead of the username. This will be optional.
- Biography, an optional bunch of text that's displayed on the user's profile

page.

To specify this in Mongoose, we must define a *schema*, which contains information about properties, methods, and more. (Personally, I don't think "Schema" is the right word; it's a lot more like a class or a prototype.) It's pretty easy to translate the English above into Mongoose code.

Create a folder called `models` in the root of your project, and create a new file called `user.js` inside that folder. To start, put the following contents:

Listing 8.2 Defining the user schema (in models/user.js)

```
var mongoose = require("mongoose");
var userSchema = mongoose.Schema({
  username: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  createdAt: { type: Date, default: Date.now },
  displayName: String,
  bio: String
});
```

After we require Mongoose, it's pretty straightforward to define our fields. As you can see, we define the username as `username`, the password as `password`, the time joined as `createdAt`, the display name as `displayName`, and the biography as `bio`. Notice that some fields are required, some are unique, some have default values, and others are simply a declaration of their types.

Once we've created the schema with the properties, we can add some methods to it. The first we'll add is simple: get the user's name. If the user has defined a display name, return that; otherwise, return their username. Here's how we'll add that:

Listing 8.3 Adding a simple method to the user model (in models/user.js)

```
…

userSchema.methods.name = function() {
  return this.displayName || this.username;
};
```

We'll also want to make sure we store the password securely. We *could* store the password in plain text in our database, but that has a number of security issues. What if someone hacked our database? They'd get all the passwords! We also want to be responsible administrators and not be able to see our users passwords in the clear. In order to ensure that we never store the "real" password, we'll apply a one-way hash to it using the Bcrypt algorithm.

First, to start using Bcrypt, add the `require` statement to the top of your file. Bcrypt works by running a part of the algorithm many times to give you a secure hash, but that number of times is configurable. The higher the number, the more secure the hash but the longer it will take. We'll use a value of 10 for now, but we could increase that number for higher security (but, once again, slower speed):

Listing 8.4 Requiring Bcrypt (in models/user.js)

```
var bcrypt = require("bcrypt-nodejs");
var SALT_FACTOR = 10;
```

Next, after you've defined your schema, we'll define a pre-save action. Before we save our model to the database, we'll run some code that will hash the password. Here's how that looks:

Listing 8.5 Our pre-save action to hash the password (in models/user.js)

```
…

var noop = function() {};                                  #1

userSchema.pre("save", function(done) {                    #2
  var user = this;                                         #3
  if (!user.isModified("password")) {                      #4
    return done();                                         #4
  }                                                        #4
  bcrypt.genSalt(SALT_FACTOR, function(err, salt) {        #5
    if (err) { return done(err); }
    bcrypt.hash(user.password, salt, noop,
    [CA]function(err, hashedPassword) {                    #6
      if (err) { return done(err); }
      user.password = hashedPassword;                      #7
      done();                                              #7
    });
  });
});
```

#1 We'll need this do-nothing function for use with the bcrypt module.
#2 We'll define a function that will be run before our model is saved.
#3 Because we'll be using inner functions, we'll save a reference to the user.
#4 Skip this logic if the user hasn't modified their password.
#5 We'll generate a salt for our hash, and call the inner function once completed.
#6 Next, we'll hash the user's password with that generated salt.
#7 Store the password and continue on with the saving!

Now, we never have to call any fancy logic to hash the password for the database—it'll happen every time we save the model into Mongo.

Finally, we'll need to write some code to compare the real password to a password guess. When a user logs in, we'll need to make sure the password they typed is correct. Let's define another method on the model to do this:

Listing 8.6 Checking the user's password (in models/user.js)

```
…

userSchema.methods.checkPassword = function(guess, done) {
  bcrypt.compare(guess, this.password, function(err, isMatch) {    #1
    done(err, isMatch);
  });
};
```

#1 For complicated security reasons (called "timing attacks" if you're interested), we'll use Bcrypt's compare function rather than something like a === check.

Now we'll be storing our users' passwords securely!

Once we've defined our schema with its properties and methods, we'll need to attach that schema to an actual model. It only takes one line to do this, and because we're defining this user model in a file, we'll make sure to export it into`module.exports` so other files can `require` it. Here's how we do that:

Listing 8.7 Creating and exporting the user model (in models/user.js)

```
…

var User = mongoose.model("User", userSchema);
module.exports = User;
```

That's how you define a user model! Here's what the full file will look like when you're done:

Listing 8.8 Finished models/user.js

```
var bcrypt = require("bcrypt-nodejs");
var mongoose = require("mongoose");

var SALT_FACTOR = 10;

var userSchema = mongoose.Schema({
  username: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  createdAt: { type: Date, default: Date.now },
  displayName: String,
  bio: String,
});

var noop = function() {};

userSchema.pre("save", function(done) {

  var user = this;

  if (!user.isModified("password")) {
    return done();
  }

  bcrypt.genSalt(SALT_FACTOR, function(err, salt) {
    if (err) { return done(err); }
    bcrypt.hash(user.password, salt, noop, function(err, hashedPassword) {
      if (err) { return done(err); }
      user.password = hashedPassword;
      done();
    });
  });

});

userSchema.methods.checkPassword = function(guess, done) {
  bcrypt.compare(guess, this.password, function(err, isMatch) {
    done(err, isMatch);
  });
};

userSchema.methods.name = function() {
  return this.displayName || this.username;
};

var User = mongoose.model("User", userSchema);

module.exports = User;
```

8.2.3 Using our model

Now that we've defined our model, we'll want to...well, use it! We'll want to do things like list users, edit profiles, and register new accounts. While defining the model and its schema can be a little hairy, using it could hardly be easier; let's see how.

In order to start using it, let's first create a simple `app.js` in the root of our project which will set up our app. This is incomplete and we'll come back and fill in some more later, but for now, here's what we'll do:

Listing 8.9 app.js, to start

```
var express = require("express");                       #1
var mongoose = require("mongoose");                     #1
var path = require("path");                             #1
var bodyParser = require("body-parser");                #1
var cookieParser = require("cookie-parser");            #1
var session = require("express-session");               #1
var flash = require("connect-flash");                   #1

var routes = require("./routes");                       #2

var app = express();

mongoose.connect("mongodb://localhost:27017/test");    #3

app.set("port", process.env.PORT || 3000);

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

app.use(bodyParser.urlencoded({ extended: false }));    #4
app.use(cookieParser());                                #4
app.use(session({                                       #4
  secret: "TKRv0IJs=HYqrvagQ#&!F!%V]Ww/4KiVs$s,<<MX",   #4
  resave: true,                                         #4
  saveUninitialized: true                               #4
}));                                                    #4
app.use(flash());                                       #4

app.use(routes);

app.listen(app.get("port"), function() {
  console.log("Server started on port " + app.get("port"));
});
```

#1 Require everything we need, including Mongoose.
#2 We'll put all of our routes in another file.
#3 Connect to our MongoDB server in the test database.
#4 Use four middlewares. We'll explain these in much more detail in a moment.

Above, we've specified that we're going to be using an external routes file. Let's define that too. Create `routes.js` in the root of your project:

Listing 8.10 routes.js, to start

```
var express = require("express");

var User = require("./models/user");

var router = express.Router();

router.use(function(req, res, next) {
  res.locals.currentUser = req.user;       #1
  res.locals.errors = req.flash("error");  #1
  res.locals.infos = req.flash("info");    #1
  next();
});

router.get("/", function(req, res, next) {
  User.find()                                   #2
  .sort({ createdAt: "descending" })            #2
  .exec(function(err, users) {                  #2
    if (err) { return next(err); }
    res.render("index", { users: users });
  });
});
```

```
module.exports = router;
```

#1 We'll come back to this, but this sets a few useful variables for our templates. If you don't understand it yet, don't worry—it'll return.
#2 This queries the users collection, returning the newest users first.

These two files have a few new things we haven't seen before.

First, we're connecting to our Mongo database with Mongoose, using `mongoose.connect`. We simply pass an address and Mongoose does the rest. Depending on how you've installed MongoDB, this URL might be different —for example, the server could be at `localhost:12345/learn_about_me_db`. Without this line, we won't be able to interact with the database at all!

Second, we're grabbing a list of users with `User.find`. Then we sort these results by the `createdAt` property, and then we run the query with `exec`. We don't actually run the query until `exec` is called. As we'll see, we can also specify a callback in `find` to skip having to use `exec`, but then we can't do things like sorting.

Let's create the homepage view. Create the `views` directory, where we'll put three files inside. The first will be `_header.ejs`, which is the HTML that will appear at the beginning of every page:

Listing 8.11 views/_header.ejs

```
<!DOCTYPE html>
<html>

<head>

<meta charset="utf-8">
<title>Learn About Me</title>
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap
[CA]/3.3.1/css/bootstrap.min.css">

</head>

<body>

<div class="navbar navbar-default navbar-static-top" role="navigation">
```

```
  <div class="container">

    <div class="navbar-header">
      <a class="navbar-brand" href="/">Learn About Me</a>
    </div>

    <ul class="nav navbar-nav navbar-right">    #1
      <% if (currentUser) { %>                  #1
        <li>                                    #1
          <a href="/edit">                      #1
            Hello, <%= currentUser.name() %>    #1
          </a>                                  #1
        </li>                                   #1
        <li><a href="/logout">Log out</a></li>  #1
      <% } else { %>                            #1
        <li><a href="/login">Log in</a></li>    #1
        <li><a href="/signup">Sign up</a></li>  #1
      <% } %>                                   #1
    </ul>                                       #1

  </div>

</div>

<div class="container">

  <% errors.forEach(function(error) { %>
    <div class="alert alert-danger" role="alert">
      <%= error %>
    </div>
  <% }) %>

  <% infos.forEach(function(info) { %>
    <div class="alert alert-info" role="alert">
      <%= info %>
    </div>
  <% }) %>
```

#1 We'll change the navbar if the user is logged in. We don't have this code yet, so the user will always appear to be logged out.

You may notice that this file starts with an underscore. It's not `header.ejs`, it's `_header.ejs`. This is a common convention: views that aren't rendered directly start with underscores. You'd never render the header directly—another view would include the header.

Next, let's create the footer in `_footer.ejs`:

Listing 8.12 views/_footer.ejs

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

Finally, create `index.ejs` which is the actual homepage. This will pull from the `users` variable that we're passed when we render this view.

Listing 8.13 views/index.ejs

```
<% include _header %>

<h1>Welcome to Learn About Me!</h1>

<% users.forEach(function(user) { %>

  <div class="panel panel-default">
    <div class="panel-heading">
      <a href="/users/<%= user.username %>">
        <%= user.name() %>
      </a>
    </div>
    <% if (user.bio) { %>
      <div class="panel-body"><%= user.bio %></div>
    <% } %>
  </div>

<% }) %>

<% include _footer %>
```

If you save everything, start up your MongoDB server, and `npm start`, and visit `localhost:3000` in your browser, you won't see much, but you'll see a homepage that looks something like this:

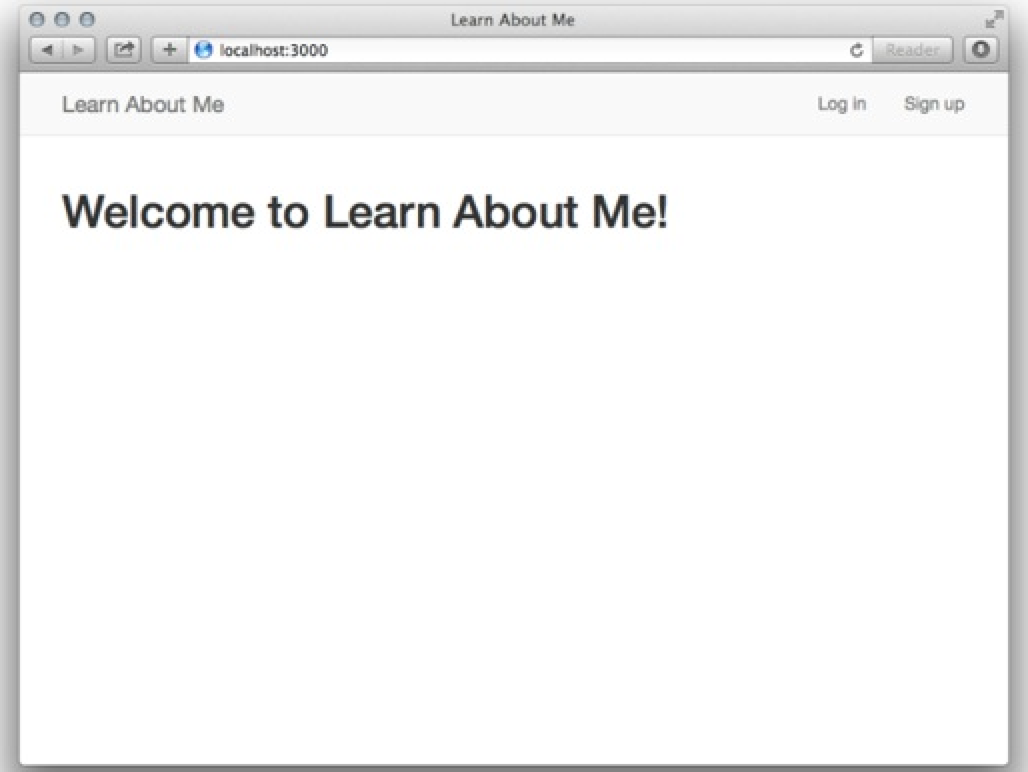


Figure 8.2 The empty LAM homepage

If you're not getting any errors, that's great! That means you're querying your Mongo database and getting all of the users in there...there just happen to be 0 users at the moment!

Let's add two more routes to our page: one for the signup page and one to do the actual signing up. In order to use that, we'll need to make sure we use the `body-parser` middleware to parse form data. Here's what those will look like:

Listing 8.14 Adding body-parser middleware (to app.js)

```
var bodyParser = require("body-parser");                #1

…

app.use(bodyParser.urlencoded({ extended: false }));    #1

…
```

#1 Require and use the body-parser middleware in our app.

Listing 8.15 Adding signup routes (in routes.js)

```
…

router.get("/signup", function(req, res) {
  res.render("signup");
});

router.post("/signup", function(req, res, next) {

  var username = req.body.username;                            #1
  var password = req.body.password;                            #1

  User.findOne({ username: username }, function(err, user) {   #2

    if (err) { return next(err); }
    if (user) {                                                #3
      req.flash("error", "User already exists");               #3
      return res.redirect("/signup");                          #3
    }                                                          #3

    var newUser = new User({                                   #4
      username: username,                                      #4
      password: password                                       #4
    });                                                        #4
    newUser.save(next);                                        #5

  });
}, passport.authenticate("login", {              #6
  successRedirect: "/",
  failureRedirect: "/signup",
  failureFlash: true
}));););
```

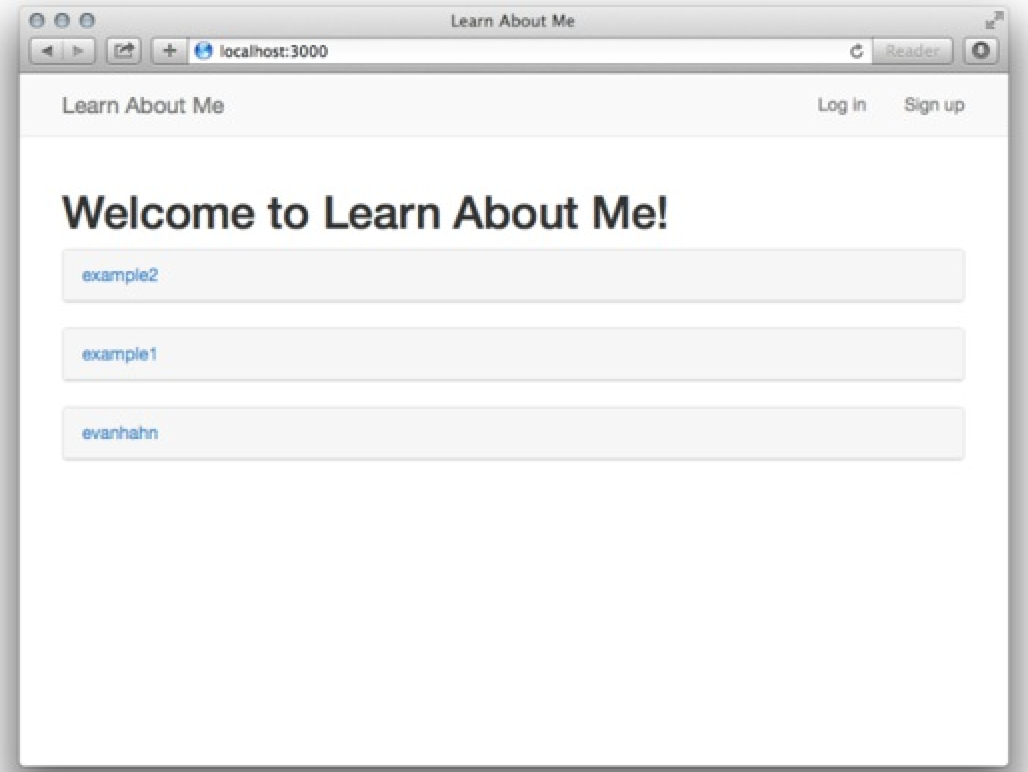
#1 body-parser populates req.body, which we see here contains the username and password for signup. Setting extended to false makes the parsing simpler and more secure, for reasons we'll see in Chapter 10.
#2 We call findOne to just return one user. We want a match on usernames here.
#3 If we find a user, we should bail out because that username already exists.
#4 Create a new instance of the User model with the username and password.
#5 Save the new user to the database and continue onto the next request handler (below).
#6 When we're all done, we should authenticate the user.

The code above effectively saves new users to our database! Let's add a user interface to this by creating `views/signup.ejs`:

Listing 8.16 views/signup.ejs

```
<% include _header %>

<h1>Sign up</h1>

<form action="/signup" method="post">
  <input name="username" type="text" class="form-control" placeholder="Username"
required autofocus>
  <input name="password" type="password" class="form-control" placeholder="Password"
required>
  <input type="submit" value="Sign up" class="btn btn-primary btn-block">
</form>

<% include _footer %>
```

Now, when you submit this form, it'll talk to the server code and sign up a new user! Start up the server with `npm start` and go to the sign up page (at `localhost:3000/signup`). Create a few accounts and you'll see them appear on the homepage!



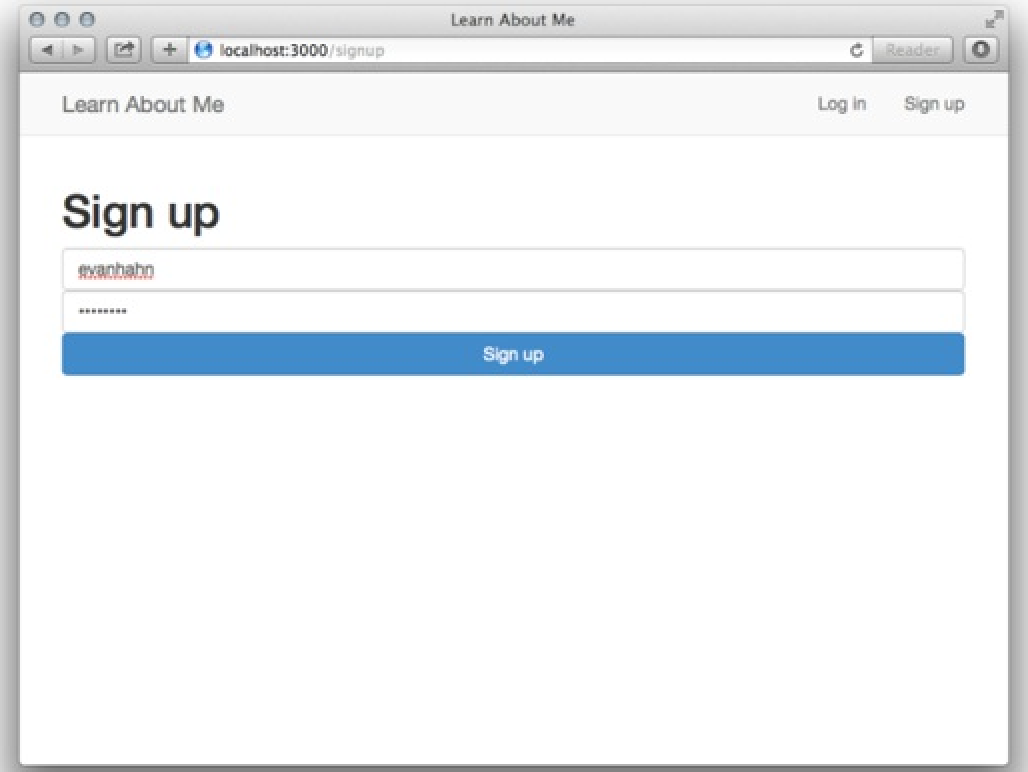
Welcome to Learn About Me!

example2

example1

evanhahn

Figure 8.3 An early LAM homepage, after creating a few users



Sign up

evanhahn

•••••••••

Sign up

Figure 8.4 The LAM signup page

The last bit of business before we have to code logging in and logging out is the viewing of profiles. We'll add just one more route for that, and that'll look like this:

Listing 8.17 The profiles route (in routes.js)

```
…

router.get("/users/:username", function(req, res, next) {
  User.findOne({ username: req.params.username }, function(err, user) {
    if (err) { return next(err); }
    if (!user) { return next(404); }
    res.render("profile", { user: user });
  });
});
…
```

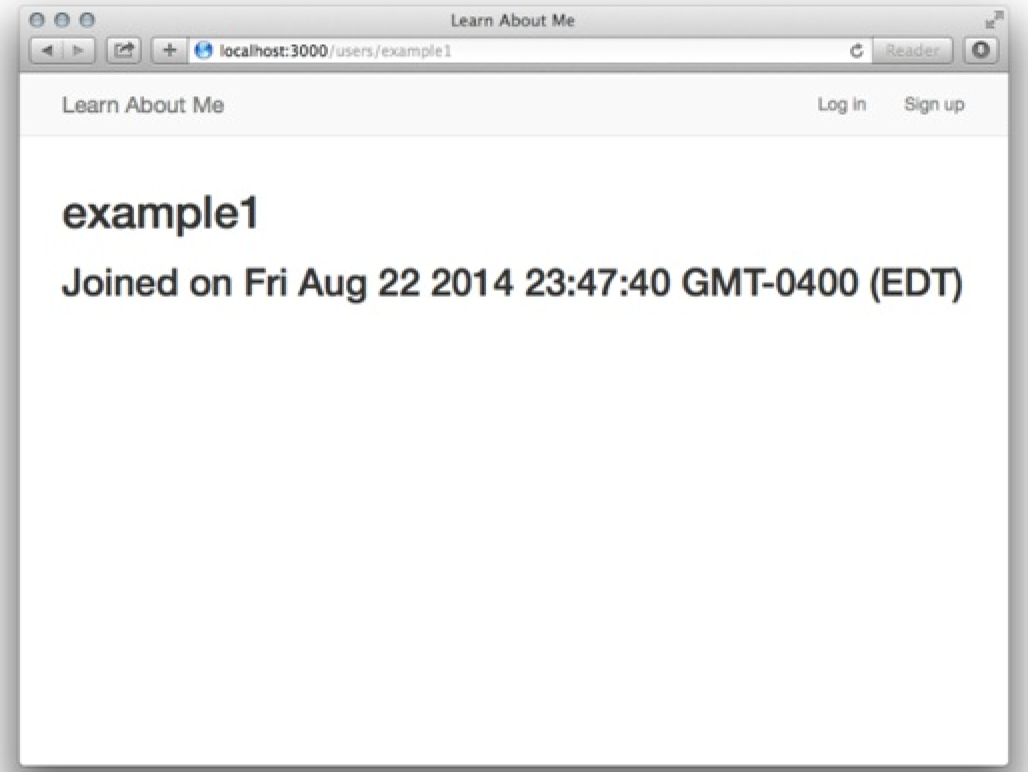
Once again, we'll be using `findOne`, but in this case we'll actually pass the user we find into the view. Speaking of, `profile.ejs` will look something like this:

Listing 8.18 views/profile.ejs

```
<% include _header %>

<% if ((currentUser) && (currentUser.id === user.id)) { %>
  <a href="/edit" class="pull-right">Edit your profile</a>
<% } %>

<h1><%= user.name() %></h1>
<h2>Joined on <%= user.createdAt %></h2>

<% if (user.bio) { %>
  <p><%= user.bio %></p>
<% } %>

<% include _footer %>
```

#1 This references currentUser, a variable that will appear once we add login and logout. For now, this will always evaluate to false.

Now we can view user profiles! Check it out in Figure 8.5:



example1

Joined on Fri Aug 22 2014 23:47:40 GMT-0400 (EDT)

Figure 8.5 The LAM profile page

Now we can create and view user profiles. Next, we'll need to add login and logout so that users can edit their existing profiles. Let's see how that works!

8.3 Authenticating users with Passport

In this chapter, we've been creating "Learn About Me", a website that lets users create and browse profiles. We've implemented the homepage, the "view profile" page, and even signup!

But right now, our app knows nothing "special" about our User model. They have no authentication, so they might as well be Cake models or Burrito models -- you can view and create them just like you could another object. We'll want to implement user authentication. We'll need a login page, the notion of a currently logged-in user (which you've seen as currentUser in a few places), and the actual verification of passwords.

For this, we'll choose Passport. To quote its documentation, "Passport is authentication middleware for Node. It is designed to serve a singular purpose: authenticate requests." We'll be dropping this middleware into our application, writing a little code to wire up our users, and we'll be in business! Passport takes away a lot of the headache for us.

It's important to remember that Passport doesn't dictate how you authenticate your users; it's only there to provide helpful boilerplate code. It's like Express in that way. In this chapter, we'll look at how to use Passport to authenticate users stored in a MongoDB database, but Passport supports authentication with providers like Facebook, Google, Twitter, and over 100 more. It's extremely modular and powerful!

8.3.1 Setting up Passport

When setting up Passport, you'll need to do three things:

1. Set up the Passport middleware; this is pretty easy.
2. Tell Passport how to serialize and deserialize users. This is a short amount of code that effectively translates a user's session into an actual user object.
3. Tell Passport how to authenticate users. In this case, this is the bulk of our code, which will instruct Passport how to talk to our Mongo database.

Let's get started.

SETTING UP THE PASSPORT MIDDLEWARE

In order to initialize Passport, you'll need to set up three official Express middlewares, a third-party middleware, and then two Passport middlewares. For your reference, they are:

1. `body-parser`
2. `cookie-parser`
3. `express-session`
4. `connect-flash`
5. `passport.initialize`

6. passport.session

We've already included some of these middlewares: body-parser, cookie-parser, express-session, and connect-flash. The first one is for parsing HTML forms. cookie-parser and express-session handle user sessions; the former is for parsing cookies from browsers and the latter is for storing sessions across different browsers. We also use connect-flash for showing error messages.

After that, make sure you require Passport and then you'll use two middleware functions it provides. Put these at the top of your application (and make sure you require them, too):

Listing 8.19 Setting up the middleware for Passport (in app.js)

```
var bodyParser = require("body-parser");
var cookieParser = require("cookie-parser");
var flash = require("connect-flash");
var passport = require("passport");
var session = require("express-session");

…

app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(session({
  secret: "TKRv0IJs=HYqrvagQ#&!F!%V]Ww/4KiVs$s,<<MX",    #1
  resave: true,                                           #2
  saveUninitialized: true                                 #3
}));
app.use(flash());

app.use(passport.initialize());
app.use(passport.session());

…
```

#1 The session needs something called a "session secret", which allows each session to be encrypted from the clients. This deters hackers from hacking into users' cookies. It needs to be a bunch of random characters (not necessarily what I have above!).

#2 The session middleware requires this option to be set, which forces the session

to be updated even when it hasn't been modified.
#3 saveUninitialized is another required option. This also resets sessions but resets ones that are uninitialized.

Once you've set that up, you'll be ready to move on to the next step: telling Passport how to extract users from the session.

SERIALIZING AND DESERIALIZING USERS

Passport needs to know how to serialize and deserialize users. In other words, we'll need to translate a user's session into an actual user object, and vice-versa. Passport's documentation does a better job describing it than I could:

> *In a typical web application, the credentials used to authenticate a user will only be transmitted during the login request. If authentication succeeds, a session will be established and maintained via a cookie set in the user's browser.*
>
> *Each subsequent request will not contain credentials, but rather the unique cookie that identifies the session. In order to support login sessions, Passport will serialize and deserialize* `user` *instances to and from the session.*

To keep our code separated, we'll be defining a new file called `setuppassport.js`. This file will export a single function that will, not surprisingly, set up this Passport stuff. Create `setuppassport.js` and require it from `app.js`:

Listing 8.20 Requiring and using Passport setup (in app.js)

```
…

var setUpPassport = require("./setuppassport");

…

var app = express();
mongoose.connect("mongodb://localhost:27017/test");
setUpPassport();

…
```

Now, let's fill in our Passport setup.

Because all of our user models have a unique `_id` property, we'll use that as our "translation". First, make sure you `require` your User model. Next, instruct Passport how to serialize and deserialize users from their ID. This code can be placed before or after the Passport middleware; place it where you'd like!

Listing 8.21 Serializing and deserializing users (in setuppassport.js)

```
var passport = require("passport");

var User = require("./models/user");

module.exports = function() {

  passport.serializeUser(function(user, done) {   #1
    done(null, user._id);                         #1
  });

  passport.deserializeUser(function(id, done) {   #2
    User.findById(id, function(err, user) {       #2
      done(err, user);                            #2
    });
  });
```

#1 serializeUser should turn a user object into an ID. We call done with no error and the user's ID.
#2 deserializeUser should turn the ID into a user object. Once we're done, we call done with any errors and the user object.

Now, once the session is dealt with, it's time to do the hard part: the actual authentication.

THE REAL AUTHENTICATION

The final part of Passport is setting up something called a strategy. Some strategies include authentication with sites like Facebook or Google; the strategy we'll use is called a *local strategy*. In short, that means the authentication is up to us, which means we'll have to write a little bit of

Mongoose code.

First, require the Passport local strategy into a variable called `LocalStrategy`:

Listing 8.22 Requiring the Passport LocalStrategy (in setuppassport.js)

```
…
var LocalStrategy = require("passport-local").Strategy;
…
```

Next, you'll need to tell Passport how to use that local strategy. Our authentication code will run through the following steps:

1. Look for a user with the supplied username.
2. If no user exists, then our user isn't authenticated; say that we're done with the message "No user has that username!"
3. If the user does exist, compare their real password with the password we supply. If the password matches, return the current user. If it doesn't, return "Invalid password."

Now, let's take that English and translate it into Passport code:

Listing 8.23 Our Passport local strategy (in setuppassport.js)

```
…

passport.use("login", new LocalStrategy(                            #1
[CA]function(username, password, done) {                            #1
  User.findOne({ username: username }, function(err, user) {        #2
    if (err) { return done(err); }
    if (!user) {                                                     #3
      return done(null, false,                                       #3
      [CA]{ message: "No user has that username!" });                #3
    }                                                                #3
    user.checkPassword(password, function(err, isMatch) {
      if (err) { return done(err); }
      if (isMatch) {
        return done(null, user);                            #4
      } else {
        return done(null, false,
        [CA]{ message: "Invalid password." });              #5
      }
    });
  });
}));
…
```

#1 This is how we tell Passport to use a local strategy.
#2 Use a MongoDB query we've seen before to get one user.
#3 If there is no user with the supplied username, return false with an error message. Call the checkPassword method we defined earlier in our User model.
#4 If it is a match, return the current user with no error.
#5 If it's not a match, return false with an error message.

As you can see, you instantiate a `LocalStrategy`. Once you do that, you call the done callback whenever you're done! You'll return the user object if it's found, and `false` otherwise.

THE ROUTES AND THE VIEWS

Finally, let's set up the rest of the views. We'll still need:

- Logging in
- Logging out
- Profile editing (when you're logged in)

Let's start with logging in. The GET route will be really straight-forward, and just render the view:

Listing 8.24 GET /login (in routes.js)

```
…

router.get("/login", function(req, res) {
  res.render("login");
});
…
```

And this is what the view, at `login.ejs`, will look like. It'll just be a simple form accepting a username and password, and then sending a POST request to `/login`:

Listing 8.25 views/login.ejs

```
<% include _header %>

<h1>Log in</h1>

<form action="/login" method="post">
  <input name="username" type="text" class="form-control"
  [CA]placeholder="Username" required autofocus>
  <input name="password" type="password" class="form-control"
  [CA]placeholder="Password" required>
  <input type="submit" value="Log in" class="btn btn-primary btn-block">
</form>

<% include _footer %>
```

Next, we'll define the handler for a POST to `/login`. This will deal with Passport's authentication. Make sure to `require` it at the top of your file:

Listing 8.26 Do the login (in routes.js)

```
var passport = require("passport");

…

router.post("/login", passport.authenticate("login", {
  successRedirect: "/",
  failureRedirect: "/login",
  failureFlash: true           #1
}));
…
```

#1 This sets an error message with connect-flash if the user fails to log in.

`passport.authenticate` returns a request handler function which we pass instead one we write ourselves. This lets us redirect to the right spot, depending on whether the user successfully logged in or not.

Logging out is also trivial with Passport. All you have to do is call `req.logout`, a new function added by Passport:

Listing 8.27 Logging out (in routes.js)

```
…

router.get("/logout", function(req, res) {
  req.logout();
  res.redirect("/");
});
…
```

Passport will populate `req.user` and `connect-flash` will populate some flash values. We added this code awhile ago, but take a look at it now; because you'll likely understand it better:

Listing 8.28 Passing data to views (in routes.js)

```
…

router.use(function(req, res, next) {
  res.locals.currentUser = req.user;          #1
  res.locals.errors = req.flash("error");
  res.locals.infos = req.flash("info");
  next();
});

…
```

#1 Every view will now have access to currentUser, which pulls from req.user, which is populated by Passport.

Now all we have is the edit page, and look at this! We can log in and log out.

Next, let's make some utility middleware that ensures users are authenticated. We won't actually use this middleware yet; we'll just define it so that other routes down the line can use it. We'll call it `ensureAuthenticated`, and we'll redirect to the login page if the user isn't authenticated.

Listing 8.29 Middleware for determining if the user is authenticated (in routes.js)

```
…
```

```
function ensureAuthenticated(req, res, next) {
  if (req.isAuthenticated()) {                                   #1
    next();
  } else {
    req.flash("info", "You must be logged in to see this page.");
    res.redirect("/login");
  }
}
…
```

#1 req.isAuthenticated is a function provided by Passport.

Now, let's use this middleware to create the "Edit profile" page.

When we GET the edit page, we'll just render the view, but we want to make sure the user is authenticated before we do that. All we have to do is pass `ensureAuthenticated` to our route, and then it's business as usual. Here's how we'd do that:

Listing 8.30 GET /edit (in router.js)

```
…

router.get("/edit", ensureAuthenticated, function(req, res) {    #1
  res.render("edit");
});
…
```

#1 First, we ensure that the user is authenticated, then we run our request handler if they haven't been redirected.

As you can see, everything is as we've seen before, except we place our middleware right before our request handler.

Let's define the edit view now. This will be in `edit.ejs`, and be a simple form that allows users to change their display name and biography:

Listing 8.31 views/edit.ejs

```
<% include _header %>

<h1>Edit your profile</h1>
```

```
<form action="/edit" method="post">
  <input name="displayname" type="text" class="form-control"
  [CA]placeholder="Display name"
  [CA]value="<%= currentUser.displayName || "" %>">
  <textarea name="bio" class="form-control"
  [CA]placeholder="Tell us about yourself!">
  [CA]<%= currentUser.bio || "" %></textarea>
  <input type="submit" value="Update" class="btn
  [CA]btn-primary btn-block">
</form>

<% include _footer %>
```

Now, let's handle that form with a POST handler. This will also ensure authentication with `ensureAuthenticated`, and will otherwise update our model and save it to our MongoDB database.

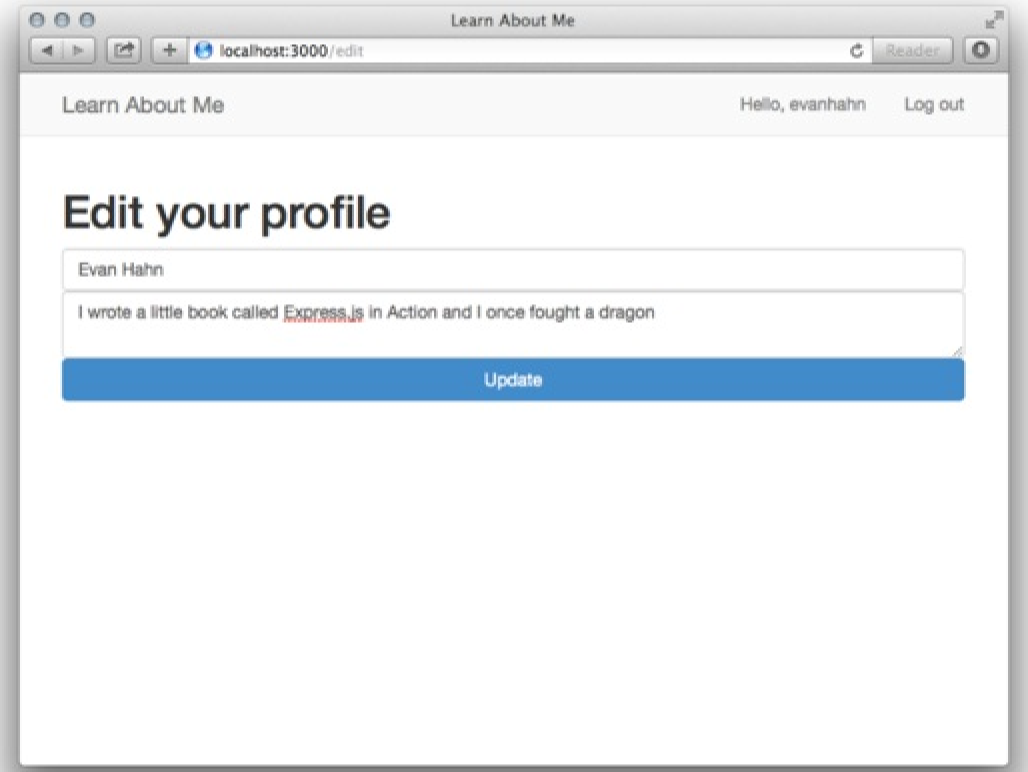
Listing 8.32 POST /edit (in routes.js)

```
…

router.post("/edit", ensureAuthenticated, function(req, res, next) {   #A
  req.user.displayName = req.body.displayname;
  req.user.bio = req.body.bio;
  req.user.save(function(err) {
    if (err) {
      next(err);
      return;
    }
    req.flash("info", "Profile updated!");
    res.redirect("/edit");
  });
});

…
```

#A Normally, this would be a PUT request, but browsers only support GET and POST in HTML forms.

There's nothing fancy here; all we do is update the user in our MongoDB database. Remember that Passport populates `req.user` for us.

Suddenly, we have our profile editor!



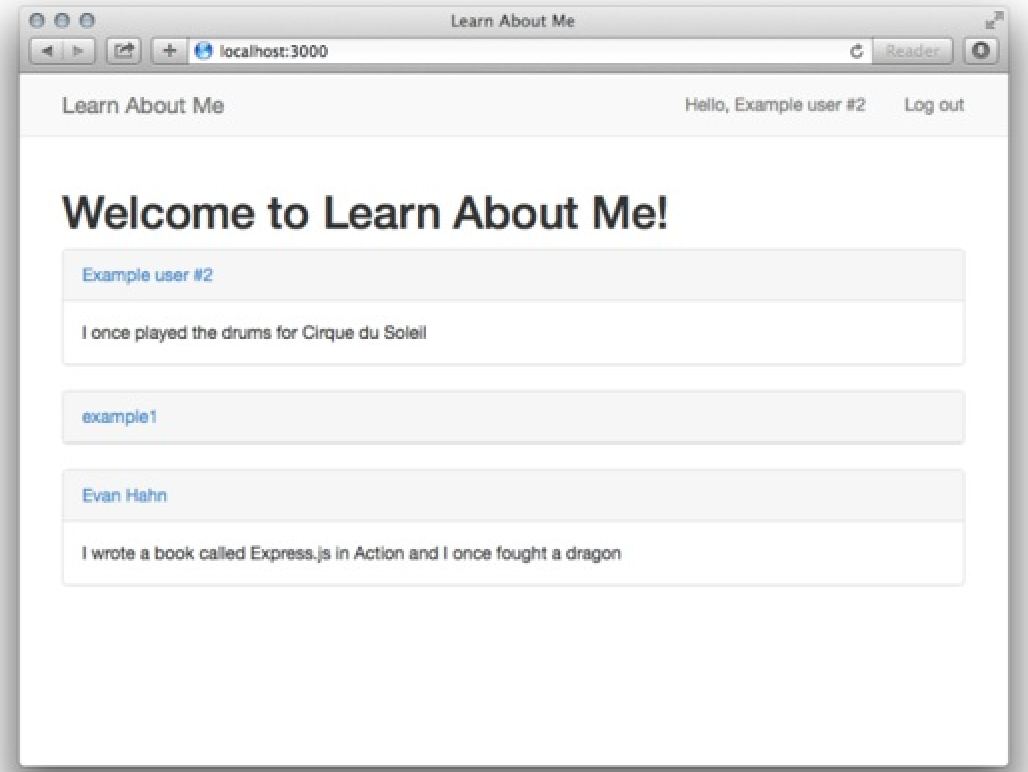
Edit your profile

Evan Hahn

I wrote a little book called Express.js in Action and I once fought a dragon

Figure 8.6 Profile editor

Now that we can edit profiles, create some fake users and edit their profiles. Check out Learn About Me, our mostly-finished app!



Welcome to Learn About Me!

Example user #2

I once played the drums for Cirque du Soleil

example1

Evan Hahn

I wrote a book called Express.js in Action and I once fought a dragon

Figure 8.7 The LAM homepage

And now you have a real app!

8.4 Summary

In this chapter you learned:

- How MongoDB works: it's a database that lets you store JavaScript-style objects
- How to use Mongoose, an official MongoDB library for controlling the database with Node
- How to securely create user accounts using bcrypt
- How to use Passport for user authentication

9 Testing Express Applications

Writing reliable code can be difficult. Even small software can have be too complex for one person, which can create bugs. Developers have come up with a number of tricks to try to squash these errors. Compilers and syntax checkers automatically scan your code for potential bugs; peer code reviews let other people look at what's written to see if they can spot errors; style guides can keep teams of developers on the same page. These are all helpful tricks we play that keep our code more reliable and bug-free.

Another powerful way to tackle bugs is with *automated testing*. Automated testing lets us codify (literally!) how we want our software to behave, and lets us say “My code works!” with much more confidence. It lets us refactor code without worrying if we broke something, and gives us easy feedback about where our code fails.

We want these benefits for our Express applications! By the end of this chapter, you'll:

- Understand the motivation for testing at a high level
- Understand the different types of testing
- Be able to do test-driven development, understanding and using the red-green-refactor model of development
- Write, run, and organize tests for general Node.js code to make sure your functions and models work as intended (using tools called Mocha and Chai)
- Test your Express applications to make sure your servers are behaving as they should (with a module called Supertest)
- Test HTML responses to make sure your views are generating the correct HTML (using a jQuery-like module called Cheerio)

Let's get started putting these components together.

9.1 What is testing and why is it important?

It should come as no surprise that there is often a disconnect between how you envision your code behaving and how it actually behaves. No programmer has ever written bug-free code 100% of the time; this is part of our profession.

If we were writing a simple calculator, for example, we know in our heads that we want it to do addition, subtraction, multiplication, and division. We can test these by hand every time we make a change—dafter making this change, does 1 plus 1 still equal 2? Does 12 divided by 3 still equal 4?—but this can be tedious and error-prone.

We can write automated tests, which effectively puts these desires into code. We write code that says "make sure, with our calculator, that 1 + 1 = 2, and that 12 ÷ 3 = 4". This is effectively a specification for your program, but it's not written in English—it's written in code for the computer, which means that you can automatically verify it. "Testing" is usually short for "automated testing", and it's simply when test code is run that verifies your "real" code.

This automatic verification has a number of advantages.

Most importantly, you can be much more confident about your code's reliability. If you've written a rigorous specification that a computer can automatically run against your program, you can be much more confident about its correctness once you've written it.

It's also really helpful when you want to change your code. A common problem is that you have a functioning program, but you want some part of it to be rewritten (perhaps to be optimized or cleaned up). Without tests, you'll have to manually verify that your old code behaves like the new code. With automated tests, you can be confident that this refactoring doesn't break anything.

Automated testing is also a lot less tedious. Imagine if, every time you wanted to test your calculator, you had to make sure that 1 + 1 = 2, 1 − 1 = 0, 1 − 3 = -2...et cetera. It'd get old pretty fast! Computers are fantastic at handling tedium like this.

In short: we write tests so we can automatically verify that our code (probably) works.

9.1.1 Test-driven development

Imagine you're writing a little JavaScript that resizes images to proper dimensions, a common task in web applications. When passed an image and dimensions, your function will return the image resized to those dimensions. Perhaps your boss has assigned this task, or perhaps it's your own impetus, but in any case, the specifications are pretty clear.

Let's say that I've convinced you to write automated tests for this; the paragraphs above have moved you. When do you write the tests? You could write the image resizer and *then* write the tests, but you could also switch things up and write the tests *first*.

Writing tests first has a number of advantages.

When you write tests first, you're literally codifying your specification. When you're finished writing your tests, you've told the computer how to ask the question: is my code finished yet? If you have any failing tests, then your code isn't conforming to the specification. If all of your tests pass, then you know that your code works as you specified. Writing the code first might mislead you and you'll write incomplete tests.

You've probably used an API that's really pleasant to work with. The code is simple and intuitive. When you write tests first, you're forced to think about how your code should work before you've even written it. This can help you design what some people call "dream code"; the easiest interface to your code. TDD can help you see the big picture about how your code should work and make for a more elegant design.

This "write tests first" philosophy is called **Test-Driven Development***, shortened to TDD. It's so named because your tests dictate how your code forms.*

TDD can really help you, but there are a few times where it can slow you down. If your specifications are unclear, you could spend a bunch of time writing tests, only to realize that you don't actually want to implement what you set out to! Now you have all of these useless tests and some wasted time.

TDD can limit your flexibility, especially if your specifications are a little foggy.

And if you're not writing tests at all, then TDD is contrary to your very philosophy and you won't write tests at all!

Some folks are TDD for all their development—test first or go home. Others are hugely against it. It's not a silver bullet nor is it a deadly poison; decide whether TDD is right for you and your code. We'll be using some TDD in this chapter, but don't take that as an unconditional endorsement. It's good for some situations and not so good for others.

HOW TDD WORKS: RED, GREEN, REFACTOR

The TDD cycle usually works in three repeating steps, called red, green, refactor, as shown in Figure 9.1.

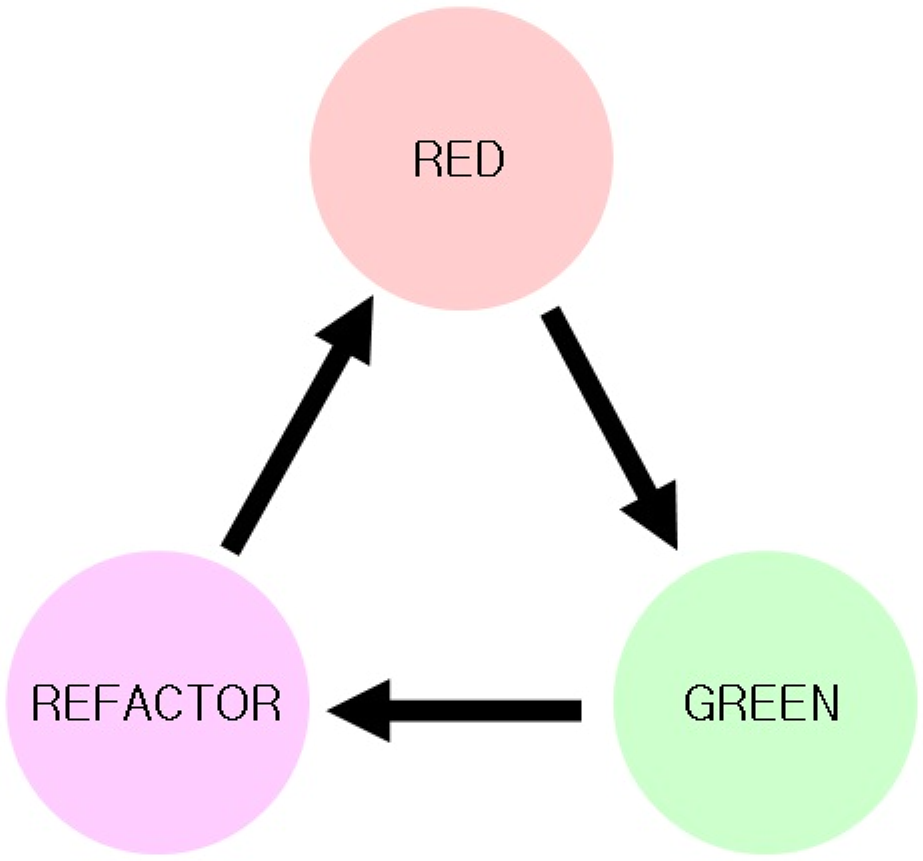


Figure 9.1 The repeating red-green-refactor cycle of TDD.

1. Step 1 is the "red" step. Because it's TDD, you write your tests first. When you write these tests before you write any of the real code, none of your tests will pass—how could they when no real code has been written? During the red step, you write all of your tests and run them to

watch them all fail. This step is so named for the red color that you usually see when you have a failing test.

2. Step 2 is the “green” step. Now that you’ve written all of your tests, you begin to “fill in” the real code to satisfy all the tests. As you make progress, your tests will slowly go from red (failing) to green (passing). Like the previous step, it’s called the “green” step because you typically see green for a passing test. Once you’re all green (all of your tests pass), you’re ready for the final step.
3. Step 3 is the “refactor” step. If all of your tests are green, that means all of your code works, but it might not be perfect. Perhaps one of your functions is slow or you’ve chosen bad variable names. Like a writer cleaning up a draft of a book, you go back and clean up the code. Because you have all of your tests, you can refactor without worrying that you’re breaking some unforeseen part of your code.
4. Step 4 is to repeat the process. You probably haven’t written all of your code for the project, so go back to step 1 and write some tests for the next part.

Here’s how we might use red-green-refactor for our image resizer:

- First, the “red” step. We’d write some of our tests. For example: if we pass it a JPEG image, our function should return a JPEG image; if we pass it a PNG image, our function should return a PNG image. *These tests aren’t complete*, but it’s a good starting point.
- Next, the “green” step. Now that we have some of our tests, we’ll fill in the code to make our tests pass. Note that we haven’t written any tests that say that we should resize the image, only that we should return the same file type. So we don’t write the image resizing yet! We simply return the image and all of our tests can pass.
- Now for the refactor step.

9.1.2 Cardinal rule: when in doubt, test

In short, you can almost never have too many tests.

As you can likely imagine, successful tests don't necessarily mean that you code works. For example, if you were testing a function, you could test that the function is a function. That's a very valid thing to test, but if it's your only test, then you might be misled into thinking that your code works when all of your tests succeed.

Because of that, you want to test as much of your code as you can. You want to poke at every (reasonable) nook and cranny of your software to make sure it performs as you expect. The more passing tests you have, the more you approach certainty that your code works as you expect. You can never be 100% sure—something might break somewhere along the line that you didn't think of—but if you've thrown everything you can possibly think of at your code, it's probably working.

Code Coverage

Testing can make you more confident about your code, but it’s just one method. As we discussed at the beginning of the chapter, there are plenty of methods like peer reviews and code linters. An extension of testing to further boost your confidence is the idea of code coverage.

Code coverage tools see how much of your code is “covered” by your tests. You could imagine writing 10 passing tests for your code but completely ignoring one of your functions which is totally broken! Code coverage tools tell you what parts of your code are untouched by tests and therefore untested. In the Node.js world, the prevailing code coverage tool seems to be Istanbul. We won’t cover it here, but if you’re looking for even more confidence, take a look at Istanbul.

Lost time is the only reason *not* to write tests. This is both lost time for the computer—some tests can be computationally expensive—and lost time for you as a human being—it takes time to type the tests!

9.2 Introduction to the Mocha testing framework

Just like it's possible to write web servers with only Node.js, it's possible to write tests with only Node.js. We could create a file that checked a bunch of conditions to make sure everything was working as normal, and then we could output the results with `console.log`. Like Express, we might find this “raw” method to be verbose and we might find ourselves having to write a lot of boilerplate code just to write tests.

Mocha is a testing framework that helps to reduce some of this headache. (It's written by the original creator of Express, by the way.) It gives you a nice syntax for organizing your tests, and has a several other features like asynchronous test support and easy-to-read output. It's not specifically tied to Express, so you can use it to test Express applications, JavaScript functions, database models, and anything else that runs inside the Node runtime.

Before we start testing Express applications, let's start by testing a simple function to see how it's done.

Imagine we want to write a function called `capitalize` that capitalizes the first character of a string and makes the rest of the string lowercase. For example, `"hello WORLD"` would become `"Hello world"`.

9.2.1 How does Node.js testing work?

Testing in Node.js applications has three major parts: the “real” code (written by you), the testing code (written by you), and the test runner (usually a third-party module, probably not written by you).

1. The “real” code is whatever you want to test. This might be a function, or a database model, or an Express server. In a Node.js context, this is anything that assigns anything to `module.exports`.
2. The test code tests your “real” code. These will `require` whatever you want to test and then start asking questions about it. Does the function return what it should return? Do your objects behave as they should behave?
3. The test runner is an executable that runs on your computer. This is an

executable that looks as your test code and runs it. Test runners will commonly print out things like “these tests succeeded, these tests failed and here's how” and “the tests took 100 milliseconds to run” and things like that. We'll be using Mocha in this chapter, but you might've used Jasmine or Qunit in your JavaScript career. You might've used Rspec or Junit in another life.

Both the real code and your test code live in the same repository. We'll also define Mocha (our test runner) as a dependency, and we'll install it locally to our repository.

9.2.2 Setting up Mocha and the Chai assertion library

Let's take a stab at writing a first version of this. Create a new directory and create one file inside: `capitalize.js`, then put the following inside:

Listing 9.1 A first version of the capitalize function (in capitalize.js)

```
function capitalize(str) {
  var firstLetter = str[0].toUpperCase();
  var rest = str.slice(1).toLowerCase();
  return firstLetter + rest;
}

module.exports = capitalize;
```

If we just eyeball the code, it looks like it *should* work, but let's write some tests to become more confident about that.

Create a `package.json` file in the same directory, which should contain the following:

Listing 9.2 The package.json for the capitalize function

```
{
  "private": true,
  "devDependencies": {
    "chai": "^1.9.2", #A
    "mocha": "^2.0.1" #A
  },
  "scripts": {
    "test": "mocha"  #B
  }
}
```

#A As always, your version numbers may vary.
#B When you type “npm test”, this will run Mocha to run your tests.

We're using two modules here: Mocha and Chai.

Mocha is a testing framework. If you've ever used other JavaScript testing frameworks like Jasmine, this should be familiar. At the end of the day, it's the thing that actually runs your tests. It's the syntax you use to say "here's what I'm testing, let me set it up, here's where I test thing A, here's where I test thing B”, et cetera.

Chai is an assertion library. While Mocha lays out the tests, Chai (almost literally) says "I expect the `helloWorld` function to return `'hello world'`". The actual syntax is `expect(helloWorld()).to.equal("hello world")`, which reads a lot like the previous English. If `helloWorld` works and returns “hello world”, your tests will pass. If it doesn't return “hello world”, an error will appear, telling you that things aren't as you expect.

There are a number of assertion libraries (including one built into Node), but at the end of the day, Mocha waits for an assertion library to throw an error. If no error is thrown, the test passes. If an error is thrown, the test fails. That’s why we use Chai—it’s a nice way to throw errors when our tests fail.

The distinction between Mocha and Chai is important. Mocha is the test runner, so there’s an actual executable that runs (you don’t ever type `node my_tests.js` nor do you ever `require` it). Mocha injects some global variables into your code—as we’ll see, these globals exist to structure each of your tests. Inside of each of these tests, you use Chai to actually test your code. When we test our capitalization library, we’ll use Mocha to break up our tests into pieces like “the capitalization library capitalizes single words” and “the capitalization library doesn’t break if you pass it the empty string”. At the Chai level, we’ll actually call our capitalization library and make sure that our module’s output matches what we expect.

9.2.3 What happens when we run our tests

As you might expect, we'll actually want to *run* these tests written with Mocha and Chai in order to make sure that our code works. How do we do this?

First, as you can see above, we've defined the test script in our `package.json`. This allows us to type `npm test` into the command line. This runs Mocha which in turn runs our tests, as you can see in Figure 9.2.

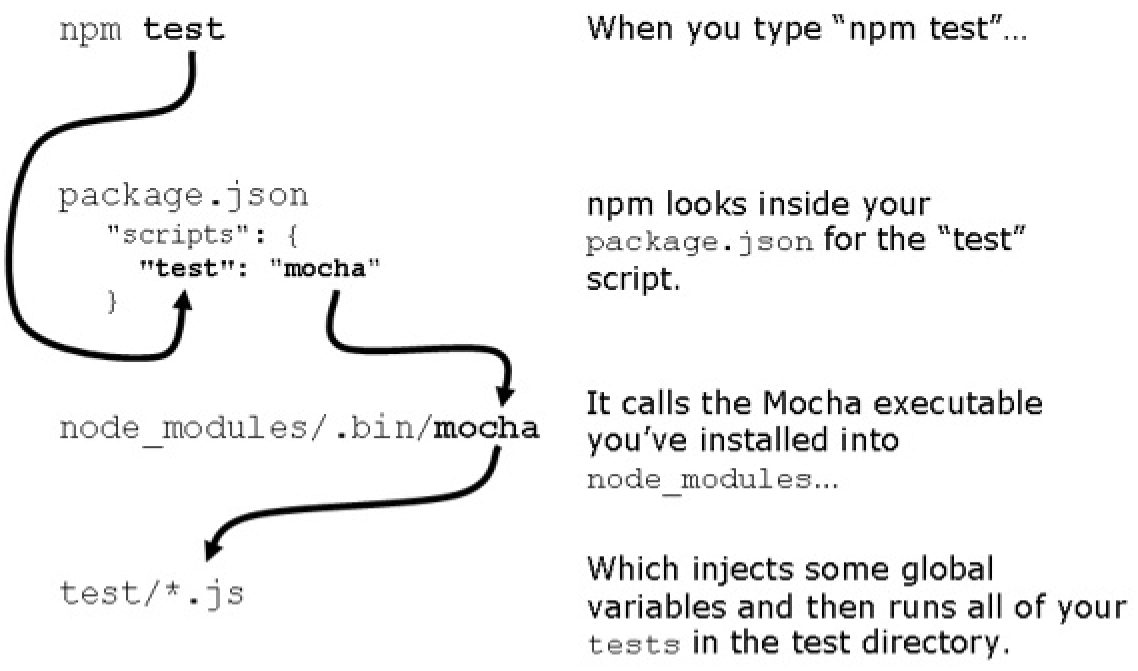

npm test
When you type "npm test"…
package.json
"scripts": {
"test": "mocha"
}
npm looks inside your package.json for the "test" script.
node_modules/.bin/mocha
It calls the Mocha executable you've installed into node_modules…
test/*.js
Which injects some global variables and then runs all of your tests in the test directory.

Figure 9.2 What happens when we type "npm test" into our command line.

Now we've set everything up. It's time to start writing some tests.

9.2.4 Writing your first test with Mocha and Chai

Now that we've written a first version of our capitalization function, let's write a test to see if it works!

Create a folder called `test` in the root of your project; this is where your test code will live. Inside that directory, create a file for testing our capitalization; I simply called mine `capitalize.js`. Put the following inside:

Listing 9.3 Our first test for capitalize (in test/capitalize.js)

```
var capitalize = require("../capitalize");  #A

var chai = require("chai");  #B
var expect = chai.expect;    #B

describe("capitalize", function() {  #C

  it("capitalizes single words", function() {  #D
    expect(capitalize("express")).to.equal("Express");  #E
    expect(capitalize("cats")).to.equal("Cats");        #E
  });

});
```

#A First, require our function that we're going to test.
#B Require Chai and then use the “expect” property, which we'll use to make assertions in our tests.
#C This is called a “suite”, and describes a series of specifications in the same topic. This is at the Mocha level.
#D This is a specification, and it has a title and some code to run. This is at the Mocha level.
#E Do the actual assertions; make sure our code actually does what we expect! This is at the Chai level.

So what's going on here?

First, we're requiring our module so that we can test it. Next, we're requiring Chai and using its `expect` property so that we can use it to make assertions later on. (Chai has two other assertion styles, but we'll stick to this one for now.)

Next, we `describe` a “suite” of tests. This is basically a component of your application; this could be a class or just a slew of functions. This suite is called "capitalize"; it's English, not code. In this case, this suite describes the capitalization function.

Inside of this suite, we define a test (we'll add more in a moment). It's a JavaScript function that says what some piece of your program should do. It says it in plain English ("it capitalizes single words") and in code. For each suite, you can have a number of tests for any number of tests you want to do.

Finally, inside of the test, we `expect` the result of `capitalize("express")` to equal `"Express"`, and the same capitalization should happen for `"cats"`.

With respect to our code, running `npm test` goes through a flow like in Figure 3:

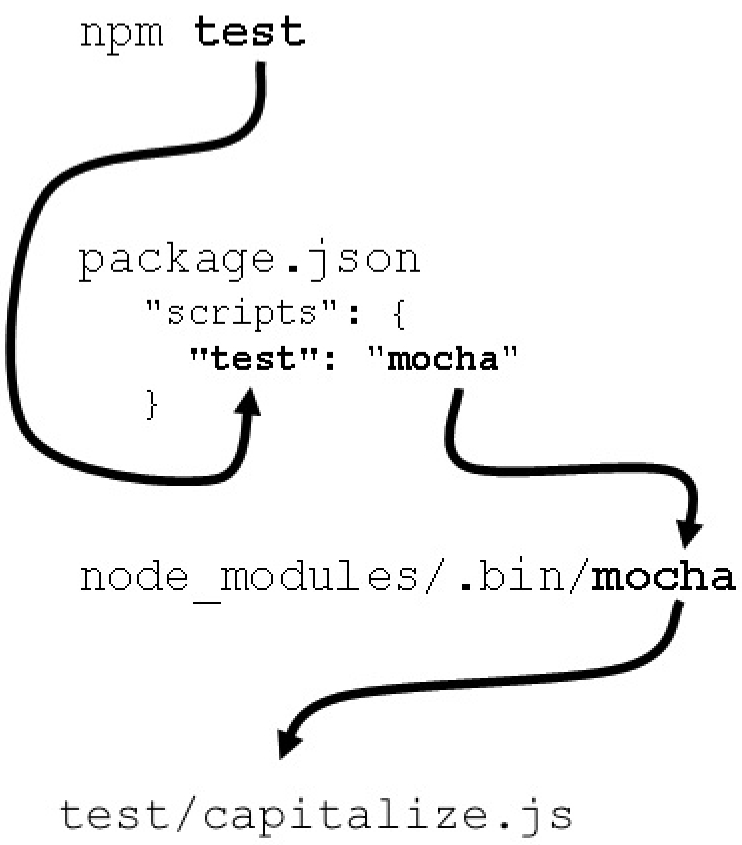


Figure 9.3 Typing npm test will go through this flow and ultimately end up running the code inside test/capitalize.js.

If you go to the root of your project and type `npm test`, you'll see something like the following output:

```
capitalize
  ✓ capitalizes single words

1 passing (9ms)
```

That means we've run one test, and it passes! Congratulations—you've written your first test. We don't know that everything works 100%, but we know that it properly capitalizes the first letter of two different words.

We're not out of the woods yet; there are more tests to write to become more confident that our code works.

9.2.5 Adding more tests

We've written a single test so far, and it's showed us that our code isn't totally broken. But we don't know if it works on more complex inputs. What would happen if you passed it a string with no letters? What about an empty string? We can see that we're capitalizing the first letter, but are we lowercasing the rest of the string? Let's add some more tests to test the “unhappy paths”.

Let's start by adding another relatively simple test: does it make the rest of the string lowercase? We'll leave everything from before and we'll add a new test to `test/capitalize.js`:

Listing 9.4 Another test for capitalize (in test/capitalize.js)

```
// …

describe("capitalize", function() {

  it("capitalizes single words", function() { /* … * / });

  it("makes the rest of the string lowercase", function() {  #A
    expect(capitalize("javaScript")).to.equal("Javascript"); #B
  });

});
```

#A Our new tests will make sure it “makes the rest of the string lowercase”.
#B We expect the capitalization of “javaScript” to equal “Javascript”.

You can run your tests with `npm test` (or just `npm t` for short), and you should see something like this:

```
capitalize
  ✓ capitalizes single words
  ✓ makes the rest of the string lowercase

2 passing (10ms)
```

Cool! Now we're more confident that we're capitalizing the first letter and lowercasing the rest of the string. But we're not out of the woods yet.

What about adding a test for the empty string? Capitalizing the empty string should just return the empty string, right? Let's write a test to see if that happens.

Listing 9.5 Testing capitalization of the empty string (in test/capitalize.js)

```
// …

describe("capitalize", function() {

  // …

  it("leaves empty strings alone", function() {
    expect(capitalize("")).to.equal("");
  });

});
```

Run `npm test` again to run this new test (and all the others). You should see something like the following output:

```
capitalize
  ✓ capitalizes single words
  ✓ makes the rest of the string lowercase
  1) leaves empty strings alone

2 passing (10ms)
1 failing

1) capitalize leaves empty strings alone:
   TypeError: Cannot call method 'toUpperCase' of undefined
    at capitalize (/path/to/capitalizeproject/capitalize.js:2:28)
    …
```

Uh oh! Looks like we have a red/failing test. Let's look at it to see what's wrong.

First, we can see that the error occurs when we run the “leaves empty strings alone" test. The error is a `TypeError`, and it's telling us that we can't call `toUpperCase` on `undefined`. We can also see a stack trace, which starts on line 2 of`capitalize.js`. Here's the line that's causing the error:

```
var firstLetter = str[0].toUpperCase();
```

Looks like `str[0]` is undefined when we pass the empty string, so we'll need to make sure it's defined. Let's replace the use of square brackets with the `charAt` method. Our new-and-improved function should look like this:

Listing 9.6 The new capitalize.js

```
function capitalize(str) {
  var firstLetter = str.charAt(0).toUpperCase();   #A
  var rest = str.slice(1).toLowerCase();
  return firstLetter + rest;
}

module.exports = capitalize;
```

#A Check out this new-and-improved line!

Re-run our tests with `npm test` and you should see everything green!

```
capitalize
  ✓ leaves empty strings alone
  ✓ capitalizes single words
  ✓ makes the rest of the string lowercase

3 passing (11ms)
```

We can add a few more tests to make sure our code is robust. We'll add a test that doesn't try to capitalize any letters. We'll also make sure it properly capitalizes multi-word strings. We should also make sure it leaves a string alone if it's already properly capitalized. These new tests should pass with the code we already have.

Listing 9.7 Some new tests for capitalization (in test/capitalize.js)

```
// …
```

```
it("leaves strings with no words alone", function() {
  expect(capitalize("  ")).to.equal("  ");
  expect(capitalize("123")).to.equal("123");
});

it("capitalizes multiple-word strings", function() {
  expect(capitalize("what is Express?")).to.equal("What is express?");
  expect(capitalize("i love lamp")).to.equal("I love lamp");
});

it("leaves already-capitalized words alone", function() {
  expect(capitalize("Express")).to.equal("Express");
  expect(capitalize("Evan")).to.equal("Evan");
  expect(capitalize("Catman")).to.equal("Catman");
});

// …
```

Run `npm test` and you should see our tests pass.

Finally, we'll try to throw one more curveball at our function:
the `String` object. Every JavaScript style guide will warn you against using the `String` object—it's bad news that can cause unexpected behavior, like they say about `==` or `eval`. It's possible that you don't even know about this feature of JavaScript, which is for the best, because you should never use it.

Unfortunately, there are inexperienced programmers out there (and others are, sadly, fools). Some of them might be using your code. You could argue that bugs are their fault, but you could also argue that *your* code shouldn't be the problem. That's why we should test our function with the `String` object, just in case. Let's write one last test that uses the `String` object.

Listing 9.8 Testing with the String object

```
// …

it("capitalizes String objects without changing their values",
[CA]function() {
  var str = new String("who is JavaScript?");
  expect(capitalize(str)).to.equal("Who is javascript?");
  expect(str.valueOf()).to.equal("who is JavaScript?");  #A
});
```

```
// …
```

#A str.valueOf() converts the String object to a "normal" string.

We've got seven tests for our little capitalization function; run `npm test` one last time to make sure they all pass!

```
capitalize
  ✓ leaves empty strings alone
  ✓ leaves strings with no words alone
  ✓ capitalizes single words
  ✓ makes the rest of the string lowercase
  ✓ capitalizes multiple-word strings
  ✓ leaves already-capitalized words alone
  ✓ capitalizes String objects without changing their values

7 passing (13ms)
```

Look at us! We're now pretty sure our capitalization function works, even when passed a variety of odd strings.

9.2.6 More features of Mocha and Chai

So far, we've only seen how we can use Mocha and Chai to test equality. Effectively, we've used a glorified equality operator. But these two modules can do much more than that. We won't go through all of the options here, but we'll look at a couple of examples.

RUNNING CODE BEFORE EACH TEST

It's common to run setup code before you actually run your assertions. Perhaps you're defining a variable to be manipulated or spooling up your server. If you're doing this setup across many tests, Mocha has the `beforeEach` function to help reduce the amount of repeated code.

For example, let's say we've made a User model and we want to test it. In every single test, we're creating a User object and we want to test it. Here's how we might do that:

Listing 9.9 Using Mocha's beforeEach feature

```
describe("User", function() {

  var user;
  beforeEach(function() {              #A
    user = new User({                  #A
      firstName: "Douglas",            #A
      lastName: "Reynholm",            #A
      birthday: new Date(1975, 3, 20) #A
    });                                #A
  });                                  #A

  it("can extract its name", function() {
    expect(user.getName()).to.equal("Douglas Reynholm");
  });

  it("can get its age in milliseconds", function() {
    var now = new Date();
    expect(user.getAge()).to.equal(now - user.birthday);
  });

});
```

#A This code is run before every single test, so that the user is defined inside of every test.

The code above tests some of the functionality of an imaginary User object, but it doesn't have code to redefine an example User object inside of every test (inside of every `it` block); it defines them in a `beforeEach` block, which redefines the user before running each test.

TESTING FOR ERRORS

If we pass a string to our capitalization function, everything should work normally. But if we pass a non-string, like a number or `undefined`, we want our function to throw some kind of error. We can use Chai to test this.

Listing 9.10 Using Chai to test for errors

```
// …

it("throws an error if passed a number", function() {
  expect(function() { capitalize(123); }).to.throw(Error);
});
```

```
// …
```

This will test that calling `capitalize` with `123` throws an error. The only tricky bit is that we have to wrap it in a function. This is because we don't want our test code to create an error—we want that error to be caught by Chai.

REVERSING TESTS

We might want to test that a value equals another value or that a function throws an error, but we might also want to test that a value *doesn't* equal another value or that a function *doesn't* throw an error. In the spirit of Chai's almost-readable-as-English syntax, we can use `.not` to reverse our test.

Let's say that we want to make sure that capitalizing "foo" doesn't equal "foo". This is a bit of a contrived example, but we might want to make sure that our capitalization function does *something*.

Listing 9.11 Negating tests

```
// ...

it("changes the value", function() {
  expect(capitalize("foo")).not.to.equal("foo");   #A
});

// …
```

#A Notice the .not in there; that's reversing our condition.

We've only begun to scratch the surface of what Chai can do. For more of its features, check out the documentation at http://chaijs.com/api/bdd/.

9.3 Testing Express servers with Supertest

The techniques above are useful for testing "business logic" like model behavior or utility functions. These are often called "unit tests"; they test discrete units of your app. But you might also want to test the routes or middleware of your Express applications. You might want to make sure that your API endpoints are returning the values they should, or that you're serving static files, or a number of other things. These are often called "integration tests" because they test the integrated system as a whole, rather than individual pieces in isolation.

We'll use Supertest to accomplish this. Supertest spools up our Express server and sends requests to it. Once the requests come back, we can make assertions about the response. For example, we might want to make sure that we get an HTTP 200 status code when we send a GET request to the homepage. Supertest will send that GET request to the homepage and then, when we get the response, make sure it had 200 as its HTTP status code. We can use this to test and middleware or route that we define in our application.

Most browsers send a header called `User-Agent` that identifies the type of browser to the server. This is often how websites serve mobile versions of sites to you if you're on your phone: a server can see that you're on a mobile device and send you a different version of the page.

Let's build “What's My User Agent?”, a simple application for getting the User Agent string of your users. It will support a “classic” HTML view when you visit it in a browser. You'll also be able to get the user's User Agent as plain text. There will be just one route for these two responses. If a visitor comes to the root of your site (at `/`) and doesn't request HTML (as most web browsers would), they'll be shown their User Agent as plain text. If they visit the same URL but their`Accepts` header mentions HTML (like web browsers do), they'll be given their User Agent as an HTML page.

Create a new directory for this project, and create a package file in the folder:

Listing 9.12 package.json for “What's my User Agent?”

```
{
  "name": "whats-my-user-agent",
  "private": true,
  "scripts": {
    "start": "node app",
    "test": "mocha"
  },
  "dependencies": {
    "ejs": "^1.0.0",        #A
    "express": "^4.10.1"
  },
  "devDependencies": {
    "mocha": "^2.0.1",
    "cheerio": "^0.17.0",  #B
    "supertest": "^0.14.0" #C
  }
}
```

#A We'll use EJS to render the HTML page, as we've used before.
#B Cheerio lets us parse the rendered HTML for testing. We'll use this to make sure the User Agent string is properly inserted into our HTML.
#C Supertest lets us spool up Express servers and test them. We'll use Supertest to test both of our application's routes.

In the previous examples, we wrote our code and *then* wrote the tests. In this example, we'll flip it around and do test-driven development. We know what we want our application to do, so we can write the tests right now without worry about *how* we implement it. Our tests will fail at first, because we won't have written any "real" code! After our tests are written, we'll go back and "fill in" the application to make our tests pass.

The TDD approach isn't always the best; sometimes you aren't quite sure what your code should look like, so it'd be a bit of a waste to write tests. There are huge flame wars online about the pros and cons of TDD; I won't reiterate them here, but we'll try TDD for this example.

We will write tests for the two major parts of this application:

1. The plain text API
2. The HTML view

Let's start by testing the plain text API.

9.3.1 Testing a simple API

Because it's the simplest, we'll start by testing the plain text API.

In plain English, this test will need to send a request to our server at the `/` route, so the server knows that we want plain text in the first place. We'll want to assert that (1) the response is the right User Agent string (2) the responses come back as plain text. Let's codify this English into Mocha tests.

Create a folder called `test` for all your tests, and create a file for testing the plain text API; I called mine `txt.js`. Inside, put the following skeleton:

Listing 9.13 Skeleton of plain text tests (in test/txt.js)

```
var app = require("../app");  #A

describe("plain text response", function() {

  it("returns a plain text response", function(done) {  #B
    // ...
  });

  it("returns your User Agent", function(done) {  #B
    // ...
  });

});
```

#A We'll require our app, because that's what will be testing. We'll put it inside app.js in the root of our project (but because this is TDD, we haven't actually done this yet).
#B There will be two tests. One makes sure we get a plain text response, and another makes sure we get the correct User Agent string.

So far, this is just a skeleton, but it's not too different from what we had before when we were testing our capitalization module. We're requiring our app (which we haven't written yet!), describing a suite of tests (plain text mode, in this case), and then defining two tests.

Let's fill in the first test, to make sure that our application returns a plain text response. Remember: what we're testing doesn't exist yet. We're going to write the tests, watch them fail, and then “fill in” the real code to make our tests pass.

Our first test will need to make a request to the server, making sure to set the `Accept` header to `text/plain`, and once it gets a response from the server, our test should ensure that it comes back as `text/plain`. The Supertest module will help us with this, so `require` it at the top of your file.

Then we’ll use Supertest to make requests to our server and see if it gives us the response we want.

Listing 9.14 Using Supertest to check the response (in test/txt.js)

```
var supertest = require("supertest");

// …

it("returns a plain text response", function(done) {  #A
  supertest(app)  #B
    .get("/")     #B
    .set("User-Agent", "my cool browser")  #B
    .set("Accept", "text/plain")  #B
    .expect("Content-Type", /text\/plain/)  #C
    .expect(200)  #C
    .end(done);   #A
});

// …
```

#A When running asynchronous tests like these, our function is passed a callback. We call that callback when we're all done running our code.
#B Supertest builds up the request. We're testing our app, visiting the “/” URL, and setting two HTTP headers: one for the User Agent and one for the types of content we accept.
#C Supertest then checks the response, making sure the Content-Type matches “text/plain” and that we get a status code of 200.

Notice how we use Supertest to test our application. It's not quite like Chai in that it reads like English, but it should be pretty straightforward. Here's what we're doing with Supertest, line by line:

1. We wrap our app up by calling `supertest` with `app` as an argument. This returns a Supertest object.
2. Next, we call `get` on that Supertest object with the route we want to request; in this case, we want the application's root (at “/”).
3. Next, we set some options on this request; in this case, we're setting the HTTP `Accept` header to `text/plain` and the User-Agent header to "my cool browser". We call `set` multiple times because we want to set

multiple headers.

4. In the first call to `expect`, we say "I want the Content-Type to match 'text/plain'". Notice that this is a regular expression, not a string. We want to be a little flexible here; the Content-Type could be "text/plain", or it could be "text/plain; charset=utf-8" or something like that. We care to test for the plain text content type, but not for the specific character set because it's just ASCII in this case, which is the same in most character encodings.
5. In the second call to `expect`, we're making sure we get the HTTP status code of 200, meaning "OK". You could imagine writing a test for a nonexistent resource, where you'd expect the status code to be 404, or any of the other many HTTP status codes.
6. Finally, we call `end` with `done`. `done` is a callback function passed to us by Mocha which we use to signal that asynchronous tests (like this one) are all done.

Next, let's fill in our second test to make sure that our application is returning the right User Agent. It'll look pretty similar to the above, but we'll actually test the response body. Let's fill in our second test:

Listing 9.15 Testing that our app returns the right User Agent string (in test/txt.js)

```
// …

it("returns your User Agent", function(done) {
  supertest(app)   #A
    .get("/")      #A
    .set("User-Agent", "my cool browser")   #A
    .set("Accept", "text/plain")   #A
    .expect(function(res) {    #B
      if (res.text !== "my cool browser") {   #B
        throw new Error("Response does not contain User Agent");   #B
      }   #B
    })    #B
    .end(done); #C
});

// …
```

#A This request setup is the same as before.

#B We call expect with a function that throws an error if we don't get the right User Agent string.

#C Once again, we call "done" when we're done.

The first three lines of this test and the last line should look similar to before; we set up Supertest to test our app, and when we're done testing things, we call `done`.

The middle part calls `expect` with a function this time. This function throws an error if `res.text` (the text that our application returns) isn't equal to the `User-Agent` header we passed in. If it *is* equal, then the function simply finishes with no fuss.

One last thing: we've got some duplicate code here. In this test, we're always making the same request to our server: the same application, the same route, and the same headers. What if we didn't have to repeat ourselves? Enter Mocha's`beforeEach` feature:

Listing 9.16 Reducing repetition in our code with beforeEach (in test/txt.js)

```
// …

describe("plain text response", function() {

  var request;
  beforeEach(function() {  #A
    request = supertest(app)  #A
      .get("/")  #A
      .set("User-Agent", "my cool browser")  #A
      .set("Accept", "text/plain");  #A
  });  #A

  it("returns a plain text response", function(done) {
    request  #B
      .expect("Content-Type", /text\/plain/)
      .expect(200)
      .end(done);
  });

  it("returns your User Agent", function(done) {
    request  #B
```

```
      .expect(function(res) {
        if (res.text !== "my cool browser") {
          throw new Error("Response does not contain User Agent");
        }
      })
      .end(done);
  });

});
```

#A We can use beforeEach to run the same code before every test in this describe block. In this case, we're reassigning the request variable to a new Supertest object.
#B We can use the variable in tests without repeating ourselves.

As you can see, we're using `beforeEach` to remove repeated code. The benefits of this really start to show as you have many tests with the same setup every time.

Now that we've written our two tests, let's run them with `npm test` as a sanity check. Because we haven't even made the file where our app will live, you should get an error that contains something like "Cannot find module '../app'". This is exactly what we expect at this point: we've written the tests but no real code, so how in the world could our tests pass? This is the "red" step in the red-green-refactor cycle.

You can make the errors a little better by creating `app.js` in the root of your project and putting a skeleton Express app inside, like this:

Listing 9.17 Skeleton of app.js

```
var express = require("express");

var app = express();

module.exports = app;
```

Your tests will still fail when running `npm test`. Your errors might look something like this:

```
  html response
```

```
    1) returns an HTML response
    2) returns your User Agent

  plain text response
    3) returns a plain text response
    4) returns your User Agent

  0 passing (68ms)
  4 failing

  1) html response returns an HTML response:
     Error: expected 200 "OK", got 404 "Not Found"
       ...

  2) html response returns your User Agent:
     TypeError: Cannot read property 'trim' of null
       ...

  3) plain text response returns a plain text response:
     Error: expected "Content-Type" matching /text\/plain/, got "text/html;
charset=utf-8"
       ...

  4) plain text response returns your User Agent:
     Error: Response does not contain User Agent
       ...
```

No doubt, these are errors. But these errors are already leagues better than "cannot find module". We can see that real things are being tested here.

Let's write our application to make these tests go from red (failing) to green (passing).

9.3.2 Filling in the code for our first tests

Now that it's time to write some “real” code, put the following inside `app.js` in the root of your project:

Listing 9.18 First draft of app.js

```
var express = require("express");

var app = express();

app.set("port", process.env.PORT || 3000);
app.get("/", function(req, res) { #A
  res.send(req.headers["user-agent"]); #A
}); #A

app.listen(app.get("port"), function() {
  console.log("App started on port " + app.get("port"));
});

module.exports = app; #B
```

#A We write some code to return the User-Agent header.
#B Export the app for testing.

The last line is the only thing that might seem new: we export the app. Normally, when you're just running a file (like `node app.js`), you don't need to export the app because you don't think of it as a module. But when you're testing the application, you'll need to export it so that the outside world can poke at it and test it.

If you run `npm test` now, you'll see something like the following output:

```
plain text response
  1) returns a plain text response
  ✓□returns your User Agent

1 passing (29ms)
1 failing

1) plain text response returns a plain text response:
   Error: expected "Content-Type" matching /text\/plain/, got "text/html; charset=utf-
8"
     at Test.assert …
     …
```

This is good! We're not all the way done because only half of our tests pass, but it looks like we're returning the right User Agent. Add just one more line to make all of our tests pass:

Listing 9.19 Making app.js return plain text

```
// …
```

```
app.get("/", function(req, res) {
  res.type("text");  #A
  res.send(req.headers["user-agent"]);
});

// …
```

#A The new line: make sure the Content-Type is some variant of plain text.

Now, when you run `npm test`, you'll see all of your tests pass!

```
plain text response
  ✓ returns a plain text response
  ✓ returns your User Agent

2 passing (38ms)
```

This is great; we're now returning the plain text responses we desire. Now we’re done with the “green” step in the red-green-refactor cycle. In this case the final refactor step is simple: we don’t have to do anything. Our code is so short and sweet that it doesn’t really need much of a clean up yet.

But wait, didn't we also want to return HTML responses, too? Our tests shouldn't be passing yet, should they? You're right, wise reader. Let's write some more tests and go back to the “red” step.

9.3.3 Testing HTML responses

As we've seen, if the user requests plain text, then they'll get plain text. But if they want HTML, they *should* get HTML, but they're just getting plain text right now. To fix this “the TDD way”, we'll write some tests to make sure the HTML stuff works, we'll watch those tests fail, and then we'll fill in the rest of the code.

Create `test/html.js` which will hold the tests for the HTML part of our server. The skeleton for this file will look pretty similar to what we've seen in the plain text part of our tests, but the “innards” of one of them will look pretty different. Here's the skeleton of the HTML tests:

Listing 9.20 Testing our HTML responses (in test/html.js)

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

var supertest = require("supertest");

describe("html response", function() {

  var request;
  beforeEach(function() {
    request = supertest(app)  #A
      .get("/")  #A
      .set("User-Agent", "a cool browser")  #A
      .set("Accept", "text/html");  #A
  });

  it("returns an HTML response", function(done) {
    // …
  });

  it("returns your User Agent", function(done) {
    // …
  });

});
```

#A This beforeEach is very similar to before, but we're requesting text/html instead of text/plain.

So far, this should look a lot like the code we had from our plain text tests. We're requiring the app and Supertest; we're doing some test setup in a `beforeEach` block; we're making sure we're getting HTML back and also the right User Agent.

The first test in this file also looks pretty darn similar to the first one we wrote in the other file. Let's fill it in now:

Listing 9.21 Testing for an HTML response (in test/html.js)

```
// …

it("returns an HTML response", function(done) {
  request
    .expect("Content-Type", /html/)
    .expect(200)
    .end(done);
});
```

```
// …
```

This is very similar to before. We're testing for a response that contains “html” and we want the HTTP status code to be 200.

The next test is where things get pretty different.

First, let's write the code to get the HTML response from the server. This should look pretty similar to what we've seen before:

Listing 9.22 Getting the HTML response (in test/html.js)

```
// …

it("returns your User Agent", function(done) {
  request
    .expect(function(res) {
      var htmlResponse = res.text;
      // …
    })
    .end(done);
});

// …
```

But now it's time to do something with that HTML. We don't just want the User Agent string to show up somewhere in the HTML. We want it to show up inside a *specific* HTML tag. Our response will look something like this:

Listing 9.23 What we might be looking for in our HTML responses

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
  <h1>Your User Agent is:</h1>
  <p class="user-agent">Mozilla/5.0 (Windows NT 6.1; WOW64; rv:28.0) Gecko/20100101
Firefox/36.0</p>
</body>
</html>
```

We don't care too much about most of this HTML; the thing we care to test is inside something with the class `user-agent`. How do we get it out?

Enter Cheerio, the final dependency from our list of devDependencies. In short, Cheerio is jQuery for Node. That might sound silly—why would you need to deal with the DOM in an environment that doesn't have a DOM?—but it's exactly what we need here. We need to be able to look through the HTML and find the User Agent inside. If we were in the browser, we could use jQuery to do this. Because we're in Node, we'll use Cheerio, which will be very familiar to anyone who knows jQuery. We'll use Cheerio to parse the HTML, find where the User Agent should be, and make sure that it's valid.

Start by requiring Cheerio at the top of your test file, and then we'll use Cheerio to parse the HTML we get from our server.

Listing 9.24 Parsing HTML with Cheerio (in test/html.js)

```
// …

var cheerio = require("cheerio");

// …

it("returns your User Agent", function(done) {
  request
    .expect(function(res) {
      var htmlResponse = res.text;
      var $ = cheerio.load(htmlResponse);  #A
      var userAgent = $(".user-agent").html().trim();  #B
      if (userAgent !== "a cool browser") {  #C
        throw new Error("User Agent not found");  #C
      }  #C
    })
    .end(done);
});

// …
```

#A Initialize a Cheerio object from our HTML.
#B Get the User Agent from the HTML. This should look a lot like jQuery.
#C Test for an User Agent just like before.

Here, we use Cheerio to parse our HTML and make sense of it like we do with jQuery. Once we've parsed the HTML and gotten the value we want, we run our tests just like before! Cheerio makes parsing HTML easy, and you can use it to test HTML responses.

Now that we've written our two tests, we can run `npm test`. We should see our plain text tests pass as before, but our new HTML tests will fail because we haven't written the code yet—this is the “red” step. Let's make those tests pass.

If you've been following along so far, the code for this shouldn't be too crazy. We'll make some changes to our request handler, and render an EJS view which will contain the User Agent as our test expects.

First, let's make some modifications to `app.js`. We'll set up EJS as our view engine and then render the HTML view when the client wants HTML.

Listing 9.25 Filling in app.js to support HTML responses

```
var express = require("express");
var path = require("path");

var app = express();

app.set("port", process.env.PORT || 3000);

var viewsPath = path.join(__dirname, "views"); #A
app.set("view engine", "ejs");                 #A
app.set("views", viewsPath);                   #A

app.get("/", function(req, res) {
  var userAgent = req.headers["user-agent"] || "none";

  if (req.accepts("html")) {              #B
    res.render("index", { userAgent: userAgent });  #B
  } else {
    res.type("text");  #C
    res.send(userAgent);  #C
  }
});

// …
```

#A Set up our views with EJS and make sure we're using the “views” directory.

#B If the request accepts HTML, render the “index” template (which we'll define in a moment).
#C Otherwise, send the User Agent string as plain text as we've done before.

This code shouldn't be too wild if you've seen views before. We're setting up EJS as our view engine, assigning a path to it, and then rendering a view if the user requests it.

The last thing we'll need to do is define the EJS view. Create `views/index.ejs` and put the following code inside:

Listing 9.26 views/index.ejs

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <style>
  html {
    font-family: sans-serif;
    text-align: center;
  }
  </style>
</head>
<body>
  <h2>Your User Agent is:</h2>
  <h1 class="user-agent">
    <%= userAgent %>
  </h1>
</body>
</html>
```

It's time for the big moment. Run all of your tests with `npm test`, and you should see a sea of positivity:

```
html response
  ✓ returns an HTML response
  ✓ returns your User Agent

plain text response
  ✓ returns a plain text response
  ✓ returns your User Agent

4 passing (95ms)
```

All of your tests pass! It’s all green! Happy days! Now you know how to test an application with Mocha, Chai, Supertest, and Cheerio.

The biggest takeaway from this chapter isn't a series of tools: it's the fact that through testing, you can be much more confident about your application's behavior. When we write code, we want our code to work as we intend. That is often hard to do, but with testing, we can be a little surer that things work as we intend.

9.4 Summary

In this chapter, we've learned:

- What testing is and how it helps us to be more confident about our code's behavior
- Different ways to test and common practices, like test-driven development and “test as much as possible”
- How to run tests in Node.js with Mocha and Chai
- How to use Mocha and Supertest to test “real” Express servers
- How to test HTML responses with Cheerio

10 Security

In Chapter 8, I told you that I had three favorite chapters. The first was Chapter 3, where I discussed the foundations of Express in an attempt to give you a solid understanding of the framework. The second favorite was Chapter 8, where your applications used databases to become "more real". Welcome to my final favorite: the chapter about security.

I probably don't have to tell you that computer security is important, and becoming more so by the day. You've surely seen news headlines about data breaches, cyber-warfare, and "hacktivism". As our world moves more and more into the digital sphere, our digital security becomes more and more important.

Keeping your Express applications secure should (hopefully) be important—who *wants* to be hacked? In this chapter, we'll discuss the various ways your applications could be subverted and how to defend yourself. **More specifically, we'll talk about**:

- How the "security mindset" can help you spot security holes
- Keeping your code bug-free (insofar as possible!)
- Protecting your users against cross-site scripting, cross-site request forgery, and man-in-the-middle attacks
- Handling the inevitable server crash
- Auditing your third-party code
- Various minor security tactics; fixing browser bugs, preventing "clickjacking", etc.

This chapter doesn't have as much of a singular flow as the others. We'll find ourselves exploring a topic and then jumping to another, and while there may be some similarities, most of these attacks are relatively disparate.

10.1 The security mindset

Famous security technologist Bruce Schneier describes something that he calls the “security mindset”:

> Uncle Milton Industries has been selling ant farms to children since 1956. Some years ago, I remember opening one up with a friend. There were no actual ants included in the box. Instead, there was a card that you filled in with your address, and the company would mail you some ants. My friend expressed surprise that you could get ants sent to you in the mail.
> I replied: "What's really interesting is that these people will send a tube of live ants to anyone you tell them to."
> Security requires a particular mindset. Security professionals -- at least the good ones -- see the world differently. They can't walk into a store without noticing how they might shoplift. They can't use a computer without wondering about the security vulnerabilities. They can't vote without trying to figure out how to vote twice. They just can't help it.
>
> — “The Security Mindset” by Bruce Schneier,
> at https://www.schneier.com/blog/archives/2008/03/the_security_mi_1.htr

---

Bruce Schneier isn’t advocating that you should steal things and break the law! But the best way to secure yourself is to *think* like an attacker—how could someone subvert a system? How could someone abuse what they’re given? If you can think like an attacker and seek out loopholes in your own code, then you can figure out how to close those holes and make your application more secure.

This chapter can’t possibly cover every security vulnerability out there. Between the time I write this and the time you read this, there will likely be a new attack vector that *could* affect your Express applications. Thinking like an attacker will help you defend your applications against the endless onslaught of possible security flaws.

Just because I'm not going through *every* security vulnerability doesn't mean I won't go through the common ones. Read on!

10.2 Keeping your code as bug-free as possible

At this point in your programming career, you've likely realized that most bugs are bad and that you should take measures to prevent them. It should come as no surprise that many bugs can cause security vulnerabilities. For example, if a certain kind of user input can crash your application, a hacker could simply flood your servers with those requests and bring the service down for everyone. We don't want that!

There are a variety of methods you can use to keep your Express applications bug-free, and therefore less susceptible to attacks. In this section, I won't cover the general principles for keeping your software bug-free, but here are a few to keep in mind:

- Testing is terribly important. We discussed testing in the previous chapter.
- Code reviews can be quite helpful. More eyes on the code almost certainly means fewer bugs.
- Don't reinvent the wheel. If someone has made a library that does what you want, you should probably use the library. Make sure the library is well-tested and reliable!
- Stick to good coding practices. We'll go over some Express and JavaScript-specific issues, but you should make sure your code is well-architected and clean.

We'll talk about Express-specifics in this section, but the things above are hugely helpful in preventing bugs, and therefore in preventing security issues.

10.2.1 Enforcing good JavaScript with JSHint

At some point in your JavaScript life, you've probably heard of *JavaScript: The Good Parts*. If you haven't, it's a famous book by Douglas Crockford, the inventor of JSON (or the "discoverer", as he calls it). It carves out a subset of the language that's deemed "good", and the rest is discouraged.

For example, Crockford discourages the use of the double-equals operator (`==`) and instead recommends sticking to the triple-equals operator (`===`). The double-equals operator does type coercion, which can get complicated and can introduce bugs, while the triple-equals operator works pretty much how you'd expect.

In addition, there are a number of common pitfalls that befall JavaScript developers that aren't necessarily the language's fault. To name a few: missing semicolons, forgetting the `var` statement, and misspelling variable names.

If there were a tool that enforced good coding style *and* a tool that helped you fix errors, would you use them? What if they were *just one tool*? I'll stop you before your imagination runs too wild: there's a tool called JSHint.

JSHint looks at your code and points out what it calls "suspicious usage". It's not *technically* incorrect to use the double-equals operator or to forget `var`, but it's likely to be an error.

To install JSHint, you'll install it globally with `npm install jshint -g`. Now, if you type `jshint myfile.js`, JSHint will look at your code and alert you to any suspicious usage or bugs. For example, take a look at this file:

Listing 10.1 A JavaScript file with a bug

```
function square(n) {
  var result n * n;   #A
  return result;
}
square(5);
```

#A Note the missing = sign here.

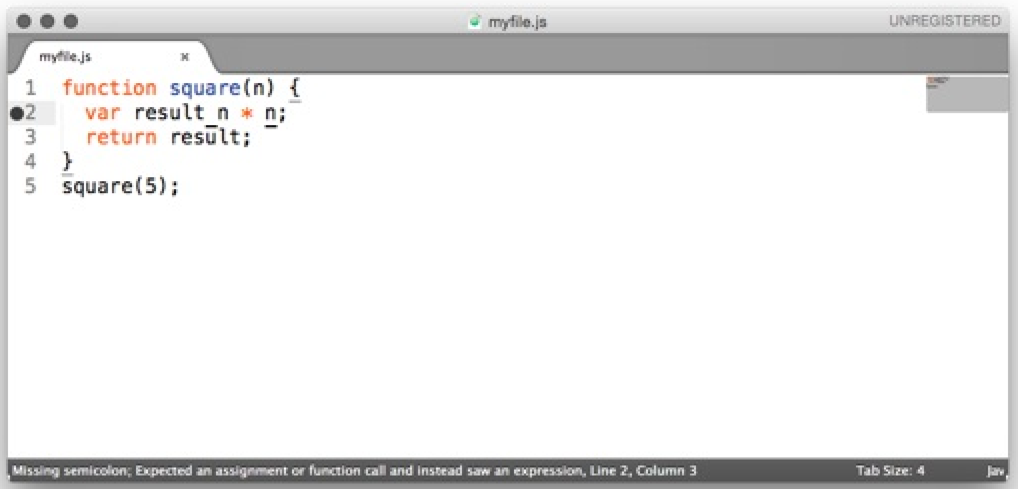
Notice that the second line has an error: we are missing an equals sign. If we run JSHint on this file (with `jshint myfile.js`), we'll see the following output:

```
myfile.js: line 2, col 13, Missing semicolon.
myfile.js: line 3, col 18, Expected an assignment or function call and instead saw an
expression.

2 errors
```

If we see this, we know that something's wrong! We can go back and add the equals sign, and then JSHint will stop complaining.

In my opinion, JSHint works best when integrated with your editor of choice.

Visit the JSHint download page at http://jshint.com/install/ for a list of editor integrations. Now, you'll see the errors before you even run the code!



```
1  function square(n) {
2    var result n * n;
3    return result;
4  }
5  square(5);
```

```
Missing semicolon; Expected an assignment or function call and instead saw an expression, Line 2, Column 3    Tab Size: 4    Jav
```

Figure 10.1 JSHint integration in the Sublime Text editor. Notice the error on the left side of the window and the message at the bottom in the status bar.

JSHint has saved me a *ton* of time when working with JavaScript and has fixed countless bugs. I know some of those bugs have been security holes.

10.2.2 Halting after errors happen in callbacks

Callbacks are a pretty important part of Node. Every middleware and route in Express uses them, not to mention…well, nearly everything else! Unfortunately, people make a few mistakes with callbacks, and these can create bugs.

See if you can spot the error in this code:

```
fs.readFile("myfile.txt", function(err, data) {
  if (err)  {
        console.error(err);
  }
  console.log(data);
});
```

In this code, we're reading a file and outputting its contents with `console.log` if everything worked. But if it *didn't* work for some reason, we output the error…and then just continue on to try to output the file's data!

If there's an error, we should be halting execution. For example:

```
fs.readFile("myfile.txt", function(err, data) {
  if (err)  {
    console.error(err);
    throw err;  #A
  }
  console.log(data);
});
```

#A If there's an error, we'll never continue to the rest of the code, because there's been an error!

It's usually important to *stop* if there's any kind of error. You don't want to be dealing with errant results—this can cause your server to have buggy behavior.

10.2.3 Perilous parsing of query strings

It's very common for websites to have query strings. For example, almost every search engine you've ever used features a query string of some sort. A search for "crockford backflip video" might look something like this:

```
http://mysearchengine.com/search?q=crockford+backflip+video
```

In Express, you can grab the query by using `req.query`, like so:

Listing 10.2 Grabbing req.query (note: contains bugs!)

```
app.get("/search", function(req, res) {
  var search = req.query.q.replace(/\+/g, " ");  #A
  // ...do something with the search...
});
```

#A This variable will now contain the string "crockford backflip video".

This is all well and good, unless the input isn't exactly as you expect. For example, if a user visits the `/search` route with no query named `q`, then we'll be calling `.replace` on an undefined variable! This can crash our server.

You'll always want to make sure that your users are giving you the data you expect, and if they aren't, you'll need to do *something* about it. One simple option is to just provide a default case, so if they don't give anything, assume the query is empty. For example:

Listing 10.3 Don't assume your queries exist (note: still contains bugs!)

```
app.get("/search", function(req, res) {
  var search = req.query.q || "";    #A
  var terms = search.split("+");
  // ...do something with the terms...
});
```

#A Now, if req.query.q is undefined, we'll fall back to non-errant behavior. Alternatively, you could redirect if nothing has been typed, or give some other behavior.

This fixes one important bug: if we're expecting a query string isn't there, we don't have undefined variables.

But there's another important gotcha with Express's parsing of query strings. In addition to the variables potentially being undefined, variables can also be of the wrong type (but still be defined)!

If a user visits `/search?q=abc`, then `req.query.q` will be a string. It'll still be a string if they visit `/search?q=abc&name=douglas`. But if they specify the `q` variable twice, like this:

```
/search?q=abc&q=xyz
```

…then `req.query.q` will be the array `["abc", "xyz"]`. Now, if you try to call `.replace` on it, it'll fail again because that method isn't defined on arrays. Oh no!

Personally, I think that this is a design flaw of Express. This behavior should be allowed, but I don't think that it should be enabled by default. Until they change it (and I'm not sure they have plans to), you'll need to assume that your queries could be arrays.

To solve this problem (and others), I wrote the `arraywrap` package (at https://www.npmjs.org/package/arraywrap). It's a very small module; the whole thing is only 11 lines of code. It's a function that takes one argument. If the argument isn't already an array, it wraps it in an array. If the argument is an array, it just returns the argument and does nothing.

You can install it with `npm install arraywrap --save` and then you can use it to coerce *all* of your query strings to arrays, like so:

Listing 10.4 Don't assume your queries aren't arrays

```
var arrayWrap = require("arraywrap");

// …

app.get("/search", function(req, res) {
  var search = arrayWrap(req.query.q || "");   #A
  var terms = search[0].split("+");
  // ...do something with the terms...
});
```

#A Now things work if we supply a single variable, supply no variables, or supply multiple variables.

Now, if someone gives you more queries than you expect, you just take the first one and ignore the rest. Alternatively, you could detect if the query was an array and do something different there.

This brings us to a big point of the chapter: *never trust user input*. Assume that every route will be broken in some way. Assume your users *could* give you bad data or no data at all!

10.3 Protecting your users

Governments have had their sites defaced; Twitter had a kind of "tweet virus"; bank account information has been stolen. Even products who aren't dealing with particularly sensitive data can still have passwords leaked—Sony and Adobe have been caught up in such scandals. If you site has users, you'll want to be responsible and protect them.

There are a number of things you can do to protect your users from harm, and we'll look at those in this section.

10.3.1 Using HTTPS

In short, you want to use HTTPS instead of HTTP. It helps protect your users against all kinds of attacks. Trust me—you want it!

There are two pieces of Express middleware that you'll want to use with HTTPS. One will force your users to use HTTPS and the other will keep them there.

FORCE USERS TO HTTPS

The first middleware we'll look at is `express-enforces-ssl`. As the name suggests, it enforces SSL (HTTPS). Basically, if the request is over HTTPS, it continues onto the rest of your middleware and routes. If not, it redirects to the HTTPS version.

To use this module, you'll need to do two things.

1. Most of the time, when you deploy your applications, your server isn't directly connecting to the client. If you're deployed to the Heroku cloud platform (as we'll see in Chapter 11), Heroku servers sit "between" you and the client. To tell Express about this, we'll need to enable the "trust proxy" setting.
2. Call the middleware!
3. Make sure you `npm install express-enforces-ssl`, and then:

Listing 10.5 Enforcing HTTPS in Express

```
var enforceSSL = require("express-enforces-ssl");
// ...
app.enable("trust proxy");
app.use(enforceSSL());
```

There's not much more to this module, but you can see more at https://github.com/aredo/express-enforces-ssl.

KEEP USERS ON HTTPS

Once your users are on HTTPS, we'll want to tell them to avoid going back to HTTP. New browsers support a feature called HTTP Strict Transport Security (shortened to HSTS). It's a simple HTTP header that tells browsers to stay on HTTPS for a period of time.

For example, if you want to keep your users on HTTPS for one year, you'd set the following header:

Listing 10.6 Sticking to HTTPS for one year

```
Strict-Transport-Security: max-age=31536000  #A
```

#A There are approximately 3,1536,000 seconds in a year.

You can also enable support for subdomains. If you own *slime.biz*, you'll probably want to enable HSTS for *cool.slime.biz.*

To set this header, we'll meet Helmet. Helmet is a module for setting helpful HTTP security headers in your Express applications. As we'll see throughout the chapter, it has various headers it can set. We'll start with its HSTS functionality.

First, as always, `npm install helmet` in whatever project you're working on. I'd also recommend installing the `ms` module, which translates human-readable strings (like `"2 days"`) into 172,800,000 milliseconds. Now you can use the middleware!

Listing 10.7 Using Helmet's HSTS middleware

```
var helmet = require("helmet");
var ms = require("ms");
// ...
app.use(helmet.hsts({
  maxAge: ms("1 year"),
  includeSubdomains: true
}));
```

Now, HSTS will be set on every request!

Why can't we just use HSTS?

This header is only effective if your users are *already* on HTTPS, which is why we need `express-enforces-ssl`.

10.3.2 Preventing cross-site scripting attacks (XSS)

I probably shouldn't say this, but there are a lot of ways you could steal my money. You could beat me up and rob me, you could threaten me, or you could pickpocket me. If you were a hacker, you could also hack into my bank and wire a bunch of my money to you (and of all the options we listed, this is the one I most prefer).

If you could get control of my browser, even if you didn't know my password, you could still get my money. You could wait for me to log in, and then take control of my browser. You'd tell my browser to go to the "wire money" page on my bank and take a large sum of money. If you were clever, you could hide it so that I'd never even know it happened (until, of course, all of my money was gone).

But how would you get control of my browser? Perhaps the most popular way would be through use of a cross-site scripting attack, also known as an XSS attack.

Imagine that, on my bank's homepage, I can see a list of my contacts and their names.

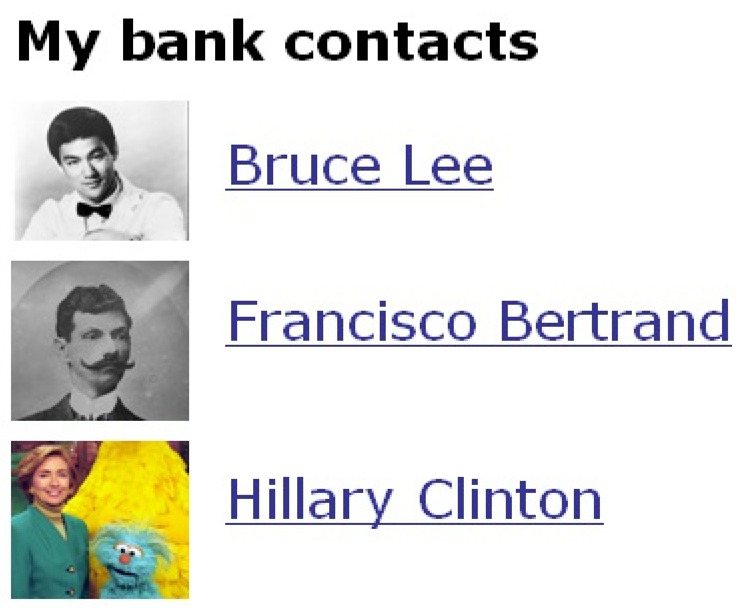


Figure 10.2 A fictional list of my bank contacts.

Users have control over their names. Bruce Lee can go into his settings and change his name to "Bruce Springsteen" if he wants to. But what if he changed his name to this:

```
Bruce Lee<script>transferMoney(1000000, "bruce-lee's-account");
</script>
```

The list of contacts would still show up the same, but now my web browser will also execute the code inside the `<script>` tag! Presumably, this will transfer a million dollars to Bruce Lee's account, and I'll never be the wiser. Bruce Lee could also add `<script src="http://brucelee.biz/hacker.js"></script>` to his name. This script could send data (like login information, for example) to *brucelee.biz*.

There's one big way to prevent XSS: never blindly trust user input.

ESCAPING USER INPUT

When you have user input, it's almost always possible that they'll enter something malicious. In the example above, you could set your name to contain `<script>` tags, causing XSS issues. We can sanitize or "escape" user input, so that when we put it into our HTML, we aren't doing anything unexpected.

Depending on where you're putting the user input, you'll sanitize things differently. As a general principle, you'll want to sanitize things as much as you can, and always keep the context in mind.

For example, if you're putting some user content inside of HTML tags, you'll want to make sure that it can't define any HTML tags. You'll want this kind of string:

> Hello, `<script src="http://evil.com/hack.js"></script>` world.

To become something like this:

```
Hello, &lt;script src="http://evil.com/hack.js"&gt;&lt;/script&gt;world.
```

By doing that, the script tags will be rendered useless.

This kind of escaping (and more) is handled by most templating engines for you. In EJS, simply use the default `<%= myString %>` and *don't* use the `<%- userString %>`. In Jade, this escaping is done by default. Unless you're certain that you don't want to sanitize something, make sure to use the "safe" version whenever you're dealing with user strings.

If you *know* that the user should be entering a URL, you'll want to do more than just escaping; you'll want to do your best to validate that something is a URL. You'll also want to call the built-in `encodeURI` function on a URL to make sure it's safe.

If you're putting something inside of an HTML attribute, you'll want to make sure your users can't put quotation marks, for example. Unfortunately, there isn't a one-size-fits-all solution for sanitizing user input; the way you sanitize depends on the context. But you should *always* sanitize user input as much as you can.

You can also escape the input before you ever put it *into* your database. In the examples above, we're showing how to sanitize things whenever we're displaying them. But if you know that your users should enter homepages on their user profiles, it's also useful to sanitize that before you ever store it in the database. If I enter "hello, world" as my homepage, the server should give an error. If I enter http://evanhahn.com as my homepage, that should be allowed, and put into the database. This can have security benefits *and* user interface benefits.

MITIGATING XSS WITH HTTP HEADERS

There's one other way to help mitigate XSS, but it's quite small, and that's through the use of HTTP headers. Once again, we'll break out Helmet.

There's a simple security header called `X-XSS-Protection`. It can't protect against all kinds of XSS, but it can protect against what's called "reflected XSS". The best example of reflected XSS is on an insecure search engine. On every search engine, when you do a search, your query appears on the screen (usually at the top). If you search for "candy", the word "candy" will appear at the top, and it'll be part of the URL:

```
https://mysearchengine.biz/search?query=candy
```

Now imagine you're searching "`<script src="http://evil.com/hack.js"></script>`". The URL might look something like this:

```
https://mysearchengine.biz/search?query=<script%20src="http://evil.com/hack.js">
</script>
```

Now, if this search engine puts that query into the HTML of the page, you've injected a script into the page! If I send this URL to you and you click the link, I can take control and do malicious things.

The first step against this attack is to *sanitize the user's input*. After that, you can set a the `X-XSS-Protection` header to keep some browsers from running that script should you make a mistake. In Helmet, it's just one line:

Listing 10.8 Using Helmet to set the X-XSS-Protection header

```
app.use(helmet.xssFilter());
```

Helmet also lets you set another header called Content Security Policy. Frankly, Content Security Policy could be its own chapter. Check out the HTML5 Rocks guide at http://www.html5rocks.com/en/tutorials/security/content-security-policy/for more information, and once you understand it, use Helmet's `csp` middleware.

Neither of these Helmet headers are anywhere near as important as sanitizing user input, so do those first!

10.3.3 Cross-site request forgery (CSRF) prevention

Imagine that I'm logged into my bank. You *want* me to transfer a million dollars into your account, but you aren't logged in as me. (Another challenge: I don't have a million dollars.) How can you get me to send you the money?

THE ATTACK

On the bank site, there's a "transfer money" form. On this form, one types the amount of money and the recipient of the money, and then they hit "Send". Behind the scenes, a POST request is being made to a URL. The bank will make sure my cookies are correct, and if they are, it'll wire the money.

You can make the POST request with the amount and the recipient, but you don't know my cookie and you can't guess it; it's a long string of characters. So what if you could make *me* do the POST request? *You'll do this with* cross-site request forgery (shortened to CSRF, and sometimes XSRF).

To pull off this CSRF attack, you'll basically have me submit a form without knowing it. Imagine that you've made a form like this:

Listing 10.9 A first draft of a hacker form

```
<h1>Transfer money</h1>
<form method="post" action="https://mybank.biz/transfermoney">
  <input name="recipient" value="YourUsername" type="text">
  <input name="amount" value="1000000" type="number">
  <input type="submit">
</form>
```

Let's say that you put this in an HTML file on a page *you* controlled; maybe it's *hacker.com/stealmoney.html*. You could email me and say, "Click here to see some photos of my cat!" If I clicked on it, I would see something like this:

And if I see that, I'll get suspicious. I won't click "submit" and I'll close the window. But we can use JavaScript to automatically submit the form.

Listing 10.10 Automatically submitting the form

```
<form method="post" action="https://mybank.biz/transfermoney">
  <!-- … -->
</form>

<script>
var formElement = document.querySelector("form");
formElement.submit();
</script>
```

If I get sent to *this* page, the form will immediately submit and I'll be sent to my bank, to a page that says "Congratulations, you've just transferred a million dollars." I'll probably panic and call my bank, and the authorities can likely sort something out.

But this is progress—you're now sending money to yourself. I won't show it here, but you can completely hide this from the victim. First, you make an `<iframe>` on your page. You can then use the form `target` attribute, so that when the form submits, it submits *inside* the `iframe`, rather than on the whole page. If you make this `iframe` small or invisible (easy with CSS!), then I'd never know I was hacked until I suddenly had a million fewer dollars.

My bank needs to protect against this. But how?

OVERVIEW OF PROTECTING AGAINST CSRF

My bank already checks cookies, to make sure that it's me. You can't perform CSRF attacks without getting me to do *something*. But once the bank knows it's me, how does it know that I meant to do something and wasn't being tricked into doing something?

My bank decides this: if you're submitting a POST request to *mybank.biz/transfermoney*, you're not just doing that out of the blue. Before doing that POST, you'll be on a page that's asking you where you want to transfer your money--perhaps the URL is *mybank.biz/transfermoney_form*.

So when the bank sends you the HTML for *mybank.biz/transfermoney_form*, it's going to add a hidden element to the form: a completely random, unguessable string called a token. The form might now look like this:

Listing 10.11 Adding CSRF protections

```
<h1>Transfer money</h1>
<form method="post" action="https://mybank.biz/transfermoney">
  <input name="_csrf" type="hidden"[CA]        #A
   value="1dmkTNkhePMTB0DlGLhm">               #A
  <input name="recipient" value="YourUsername" type="text">
  <input name="amount" value="1000000" type="number">
  <input type="submit">
</form>
```

#A The value of the CSRF token will be different for every user, often every time. The above is just an example.

You've probably used thousands of CSRF tokens while browsing the web, but you haven't seen it because it's hidden from you. If you're like me and you enjoy viewing the HTML source of pages, you'll see CSRF tokens!

Now, when you submit the form and send the POST request, my bank will make sure that the CSRF token I send is the same as the one I just received. If it is, the bank can be pretty sure that I just came from the bank's website and therefore intended to send the money. If it's not, I might be being hacked—don't send the money.

In short, we need to do two things:

1. Create a random CSRF token every time we're asking users for data
2. Validate that random token every time we deal with that data

PROTECTING AGAINST CSRF IN EXPRESS

The Express team has a simple middleware that does those two tasks: `csurf` (at https://github.com/expressjs/csurf). The `csurf` middleware does two things:

1. Adds a method to the request object called `req.csrfToken`. You'll send this token whenever you send a form, for example.

2. If the request is anything other than a GET, it looks for a parameter called `_csrf` to validate the request, creating an error if it's invalid. (Technically, it skips also skips HEAD and OPTIONS requests, but those are much less common. There are also a few other places the middleware will search for CSRF tokens; consult the documentation for more.)

To install this middleware, just run `npm install csurf --save`.

The `csurf` middleware depends on some kind of session middleware and middleware to parse request bodies. If you need CSRF protections, you probably have some notion of users, which means that you're probably already using these, but`express-session` and `body-parser` do the job. Make sure you're using those before you use `csurf`. If you need an example, you can check out Chapter 8's code for `app.js`, or look at the CSRF example app at https://github.com/EvanHahn/Express.js-in-Action-code/blob/master/Chapter_10/csrf-example/app.js.

To use the middleware, simply `require` and `use` it:

Listing 10.12 Using the CSRF middleware

```
var csrf = require("csurf");
// …
app.use(csrf()); #A
```

#A Make sure to include a body parser and session middleware before this.

Once you've used the middleware, you can grab the token when rendering a view, like so:

Listing 10.13 Getting the CSRF token

```
app.get("/", function(req, res) {
  res.render("myview", {
    csrfToken: req.csrfToken()
  });
});
```

Now, inside of a view, you'll output the `csrfToken` variable into a hidden input called `_csrf`. It might look like this in an EJS template:

Listing 10.14 Showing the CSRF token in a form

```
<form method="post" action="/submit">
  <input name="_csrf" value="<%= csrfToken %>" type="hidden">
  <! -- … -->
</form>
```

And that's all! Once you've added the CSRF token to your forms, the `csurf` middleware will take care of the rest.

It's not required, but you'll probably want to have some kind of handler for failed CSRF. Define an error middleware that checks for a CSRF error. For example:

Listing 10.15 Handling CSRF errors

```
// …

app.use(function(err, req, res, next) {
  if (err.code !== "EBADCSRFTOKEN") {    #A
    next(err);                           #A
    return;                              #A
  }                                      #A
  res.status(403);   #B
  res.send("CSRF error.");
});

// …
```

#A We'll skip this handler if it's not a CSRF error.
#B Error code 403 is "Forbidden".

This error handler will return "CSRF error" if there's, well, a CSRF error. You might want to customize this error page and you might also want this to send you a message—someone's trying to hack one of your users!

You can place this error handler wherever in your error stack you'd like. If you want it to be the first error you catch, put it first. If you want it to be last, you can put it last.

10.4 Keeping your dependencies safe

Any Express application will depend on at least one third-party module: Express. If the rest of this book has shown you anything, it's that you'll be depending on *lots* of third-party modules. This has the huge advantage that you don't have to write a lot of boilerplate code, but it does come with one cost: you're putting your trust in these modules. What if the module creates a security problem?

There are three big ways that you can keep your dependencies safe:

1. Audit the code yourself
2. Make sure you're on the latest versions
3. Check against the Node Security Project

10.4.1 Auditing the code

It might sound a bit crazy, but you can often easily audit the code of your dependencies. While some modules like Express have a relatively large surface area, many of the modules you'll install are only a small number of lines, and you can understand them quickly. It's a fantastic way to learn, too!

Just as you might look through your own code for bugs or errors, you can look through other people's code for bugs and errors. If you spot them, you can avoid the module. If you're feeling generous, you can often submit patches because these packages are all open source.

If you've already installed the module, you can find its source code in your `node_modules` directory. You can almost always find modules on GitHub with a simple search, or from a link on the npm registry.

It's also worth checking a project's overall status. If a module is old but works reliably and has no open bugs, then it's probably safe. But if it has lots of bug reports and hasn't been updated in a long time, that's not a good sign!

10.4.2 Keeping your dependencies up to date

It's (almost) always a good idea to have the latest versions of things. People tune performance, fix bugs, and improve APIs. You *could* manually go through each of your dependencies to find out which versions were out of date, or you could use a tool built into npm: `npm outdated`.

Let's say that your project has Express 4.2.0 installed, but the latest version is 4.11.1 (which I'm sure will be out of date by the time you read this). In your project directory, run `npm outdated --depth 0` and you'll see an output something like this:

```
Package         Current  Wanted  Latest  Location
express           4.2.0   4.2.0  4.11.1  express
```

If you have other outdated packages, this command will report those too. Go into your `package.json`, update the versions, and run npm install to get the latest versions! It's a good idea to check for outdated packages frequently.

What's that depth thing?

`npm outdated --depth 0` will tell you all of the modules that are outdated that you installed. `npm outdated` *without* the depth flag tells you modules that are outdated, even ones you didn't directly install. For example, Express depends on a module called`cookie`. If `cookie` gets updated but Express doesn't update to the latest version of `cookie`, then you'll get a warning about `cookie`, even though it isn't really your "fault".

There's not much I can do if Express doesn't update to the latest version (that's largely out of my control), other than update to the latest version of Express (which *is* in my control). The `--depth` flag only shows actionable information, where leaving it out gives you a bunch of information you can't really use.

Another side note: you'll want to make sure that you're on the latest version of Node, too. Check nodejs.org and make sure you're on the latest version.

10.4.3 Check against the Node Security Project

Sometimes, modules have security issues. Some nice folks set up the Node Security Project, an ambitious undertaking to audit every module in the npm registry. If they find an insecure module, they post an advisory at http://nodesecurity.io/advisories.

The Node Security Project also comes with a command-line tool, called `nsp`. It's a simple but powerful tool that scans your `package.json` for insecure dependencies (by comparing them against their database).

To install it, run `npm install -g nsp` to install the module globally. Now, in the same directory as your `package.json`, type:

```
nsp audit-package
```

Most of the time, you'll get a nice message that tells you that your packages are known to be secure. But sometimes, one of your dependencies (or, more often, one of your dependencies' dependencies) has a security hole.

For example, Express depends on a module called `serve-static`; this is `express.static`, the static file middleware. In early 2015, a vulnerability was found in `serve-static`. If you're using a version of Express that depended on`serve-static`, run `nsp audit-package` and you'll see something like this:

```
Name            Installed  Patched  Vulnerable Dependency
serve-static        1.7.1  >=1.7.2  myproject > express
```

There are basically two important things here. The leftmost column tells you the name of the problematic dependency. The rightmost column shows you the chain of dependencies that lead to the problem. In this example, your project (called "myproject") is the first issue, which depends on Express, which then depends on `serve-static`. This means that Express needs to update in order to get the latest version of `serve-static`. If you depended on serve-static directly, you'd only see your project name in the list, like this:

```
Name            Installed  Patched   Vulnerable Dependency
serve-static        1.7.1  >=1.7.2   myproject
```

Note that modules can still be insecure; there are *so many* modules on npm that the Node Security Project can't possibly audit all of them. But it's another helpful tool to help keep your apps secure.

10.5 Handling server crashes

I've got some bad news: your server is might crash at some point.

There are loads of things that can crash your servers. Perhaps there's a bug in your code and you're referencing an undefined variable; perhaps a hacker has found a way to crash your server with malicious input; perhaps your servers have reached their capacities. Unfortunately, these apps can get wildly complicated, and at some point, they might crash.

And while this chapter has tips to help keep your apps running smoothly, you don't want a crash to completely ruin your day. While they're not ideal, you can recover from them.

The folks at Nodejitsu developed a simple tool called Forever. Its name might be a hint: it keeps your apps running forever. The important part: if your app crashes, Forever will try to restart it.

To install Forever, just run `npm install forever --save`. You've probably had an `npm start` script in your `package.json` for awhile, and we're going to change it from this:

Listing 10.16 A classic npm start script

```
…
"scripts": {
  "start": "node app.js"
}
…
```

…to this:

Listing 10.17 npm start with Forever

```
…
"scripts": {
  "start": "forever app.js"
}
…
```

And now your server will restart if it crashes!

> **NOTE** You can see a simple code example of this in action at the book's source code repository at https://github.com/EvanHahn/Express.js-in-Action-code/tree/master /Chapter_09/forever-example.

10.6 Various little tricks

We've already covered most of the big topics like cross-site scripting and HTTPS. There are a few other tricks that you can employ to make your Express applications even more secure. The topics in this section are hardly as essential in the ones above, but they're quick and easy and can lower the amount of places that you can be attacked.

10.6.1 No Express here!

If a hacker wants to break into your site, they've got a lot of things to try. If they know that your site is powered by Express and they know that Express or Node have some kind of security flaw, they can try to exploit it. It'd be better to leave hackers in the dark about this!

By default, however, Express publicizes itself. In every request, there's an HTTP header that identifies your site as powered by Express. `X-Powered-By: Express` is sent with every request, by default. This can easily be disabled with a setting:

Listing 10.18 Disabling X-Powered-By: Express

```
app.disable("x-powered-by");   #A
```

#A Disabling the x-powered-by option disables the setting of the header.

Disabling this will make it just a little harder for hackers. It'll hardly make you invincible—there are plenty of other avenues for attack—but it can help a little!

10.6.2 Preventing clickjacking

I think clickjacking is quite clever. It's relatively easy to prevent, but I almost feel guilty for doing so—it's such a clever trick.

Imagine I'm a hacker, and I want to find out information from your private social networking profile. I'd love it if you would just make your profile public. It'd be so easy, if I could just get you to click the big button:

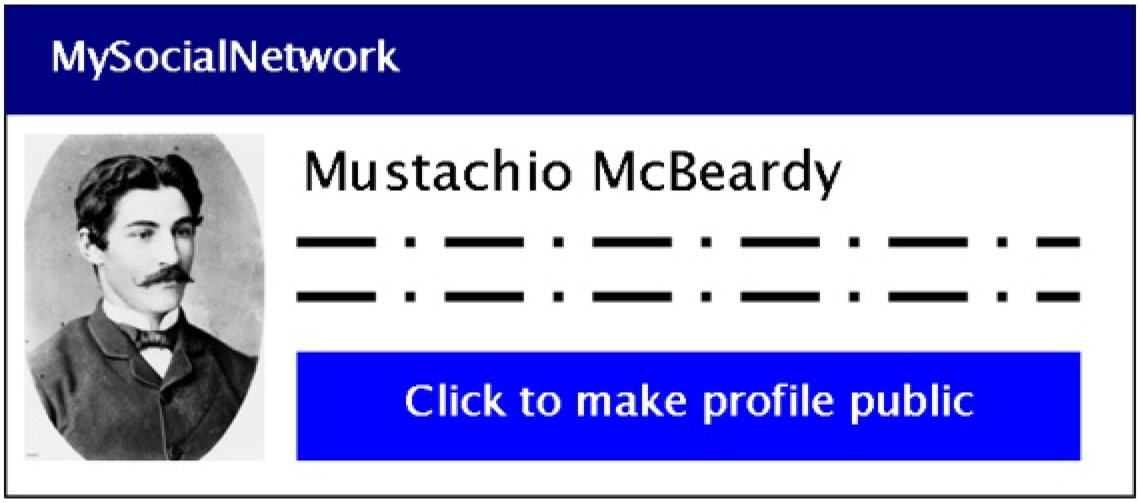


Figure 10.3 An example page for a social network.

Clickjacking takes advantage of browser frames—the ability to embed one page in another—to make this happen. I could send you a link to an innocent-looking page, which might look something like this:

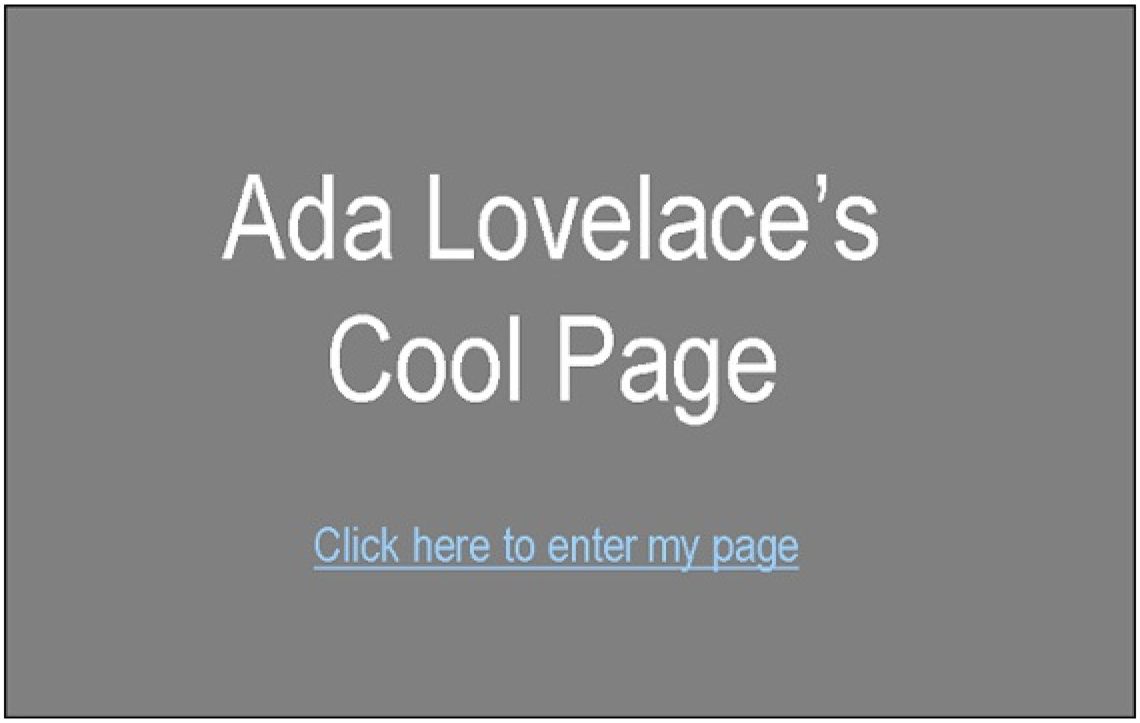


Figure 10.4 An innocent-looking page that's concealing a clickjacking attack.

But in reality, this innocent-looking page is concealing the social network page! There's an `<iframe>` of the social network site, and it's invisible. It's positioned *just right*, so that when you click "Click here to enter my page", you're actually clicking "Click to make profile public".

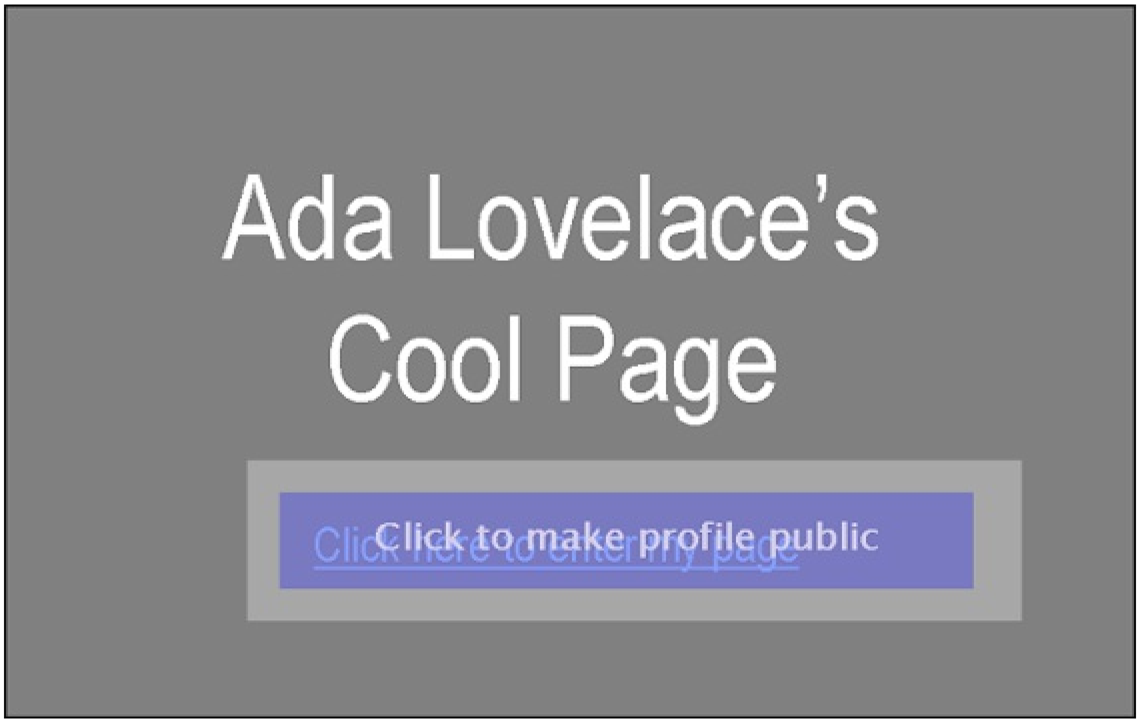


Figure 10.5 Not so innocent now, is it!?

I don't know about you, but I think that's quite clever. Unfortunately for hackers, it's quite easily prevented.

Most browsers (and *all* modern ones) listen for a header called `X-Frame-Options`. If it's loading a frame or iframe and that page sends a restrictive `X-Frame-Options`, the browser won't load the frame any longer.

`X-Frame-Options` has three options. `DENY` keeps *anyone* from putting your site in a frame, period. `SAMEORIGIN` keeps anyone *else* from putting your site in a frame, but your own site is allowed. You can also let *one* other site through with the `ALLOW-FROM` option. I'd recommend the `SAMEORIGIN` or `DENY` options.

As before, if you're using Helmet, you can set them quite easily:

Listing 10.19 Keeping your app out of frames

```
app.use(helmet.frameguard("sameorigin"));
// or…
app.use(helmet.frameguard("deny"));
```

This Helmet middleware will set the `X-Frame-Options` so you don't have to worry about your pages being susceptible to clickjacking attacks.

10.6.3 Keeping Adobe products out of your site

Adobe products like Flash Player and Reader can make cross-origin web requests. As a result, a Flash file could make requests to your server. If another website serves a malicious Flash file, users of that site could make arbitrary requests to your Express application (likely unknowingly). This could cause them to hammer your server with requests or to load resources you don't intend them to.

This is easily prevented by adding a file at the root of your site called `crossdomain.xml`. When an Adobe product is going to load a file off of your domain, it will first check the `crossdomain.xml` file to make sure your domain allows it. As the administrator, you can define this XML file to keep certain Flash users in or out of your site. It's likely, however, that you don't want *any* Flash users on your page. In that case, make sure you're serving this

XML content at the root of your site (at `/crossdomain.xml`):

Listing 10.20 The most restrictive crossdomain.xml

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM [CA]
"http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="none">
</cross-domain-policy>
```

This prevents any Flash users from loading content off of your site, unless they come from your domain. If you're interested in changing this policy, take a look at the spec athttps://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html.

This file can be served up in a few ways. If you're using Helmet from before, you can simply add a middleware and be done:

Listing 10.22 Serving crossdomain.xml with Helmet

```
app.use(helmet.crossdomain());
```

Alternatively, if you're serving static files (which you likely are), you can place that restrictive `crossdomain.xml` file into your static file directory.

10.6.4 Don't let browsers infer the filetype

Imagine a user has uploaded a plain text file to my server called `file.txt`. My server serves this with a `text/plain` content-type, because it's plain text. So far, this is simple. But what if `file.txt` contains something like this:

Listing 10.23 A malicious script that could be stored as plain text

```
function stealUserData() {
  // something evil in here…
}
stealUserData();
```

Even though we're serving this file as plain text, this looks like JavaScript, and some browsers will try to "sniff" the file type. That means that you can still run that file with `<script src="file.txt"></script>`. Many browsers will allow`file.txt` to be run even if the content-type isn't for JavaScript!

This example extends further if `file.txt` looks like HTML and the browser interprets it as HTML. That HTML page can contain malicious JavaScript, which could do lots of bad things!

Luckily, we can fix this with a single HTTP header. You can set the `X-Content-Type-Options` header to its only option, `nosniff`. Helmet comes with `noSniff` middleware, and you can use it like this:

Listing 10.24 Preventing browsers from sniffing MIME types

```
app.use(helmet.noSniff());
```

Nice that one HTTP header can fix this!

10.7 Summary

In this chapter, we've learned how:

- The "security mindset" can keep you more secure
- To keep your Express code bug-free, using tools like JSHint, testing, and awareness of common of bugs
- Various attacks; how they work, and how to prevent them. These included cross-site scripting, cross-site request forgery, man-in-the-middle attacks, clickjacking, and more.
- To handle the inevitable server crash
- Audit your third-party code

11 Deployment: Assets & Heroku

It's time to put our applications into the real world.

The first part of this chapter will discuss assets. If you're building any sort of website, it's very likely that you'll be serving some CSS and some JavaScript. It's common to concatenate and minify these assets for performance. It's also common to code in languages that compile to CSS (like SASS and LESS), just as it's common to code in laguages that transpile to JavaScript (like CoffeeScript or TypeScript) or to concatenate and minify JavaScript. Debates quickly turn into flame wars when talking about things like this; should you use LESS or SASS? Is CoffeeScript a good thing? Whichever you choose, I'll show you how to use a few of these tools to package up your assets for the web.

The rest of this chapter will show you how to build your Express applications and then put them online. There are *lots* of deployment options, but we'll choose one that's easy and free to try: Heroku. We'll add a few small things to our app and deploy an Express app into the wild!

After this chapter, you'll:

- Develop CSS with more ease using the LESS preprocessor
- Use Browserify to use `require` in the browser, just like in Node
- Minify your assets to make the smallest files possible
- Use Grunt to run this compilation and much more
- Use some Express middleware (connect-assets) as an alternative to this Grunt workflow
- Know how to deploy Express applications to the web with Heroku

11.1 LESS, a more pleasant way to write CSS

Harken back to Chapter 1, where we talked about the motivations for Express.

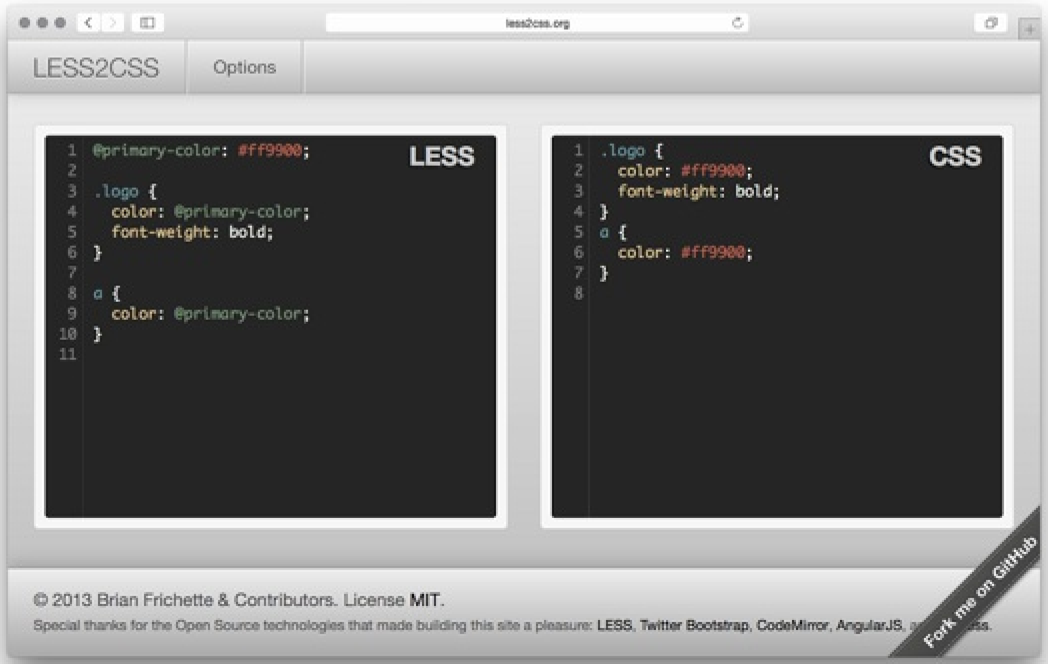
In short, we said that Node.js is powerful but its syntax can be a little cumbersome and it can be a little limited. That's why Express was made—it doesn't fundamentally change Node; it just smooths it out a bit.

In that way, LESS and CSS are a lot like Express and Node. In short, CSS is a powerful layout tool but its syntax can be cumbersome and limited. That's why LESS was made—it doesn't fundamentally change CSS; it just smooths it out a bit.

CSS is a powerful tool for laying out webpages, but it's missing a number of features that people wanted. For example, developers want to reduce repetition in their code by with constant variables instead of hard-coded values; variables are present in LESS but not CSS. LESS extends CSS and adds a number of powerful features.

Unlike Express, LESS is actually its own language. That means that it has to be compiled down into CSS in order to be used by web browsers—browsers don't “speak” LESS, they speak CSS.

We'll see two different ways to compile LESS to CSS in Express applications. For now, while you're trying LESS, visit http://less2css.org/. On the left of the page, you'll be able to type LESS code, and compiled CSS will appear on the right.



```
@primary-color: #ff9900;

.logo {
  color: @primary-color;
  font-weight: bold;
}

a {
  color: @primary-color;
}
```

```
.logo {
  color: #ff9900;
  font-weight: bold;
}
a {
  color: #ff9900;
}
```

© 2013 Brian Frichette & Contributors. License MIT.
Special thanks for the Open Source technologies that made building this site a pleasure: LESS, Twitter Bootstrap, CodeMirror, AngularJS, a

Figure 11.1 less2css.org in action.

We'll go through a few examples in the following sections and you can try them out on that website. When it's time to integrate LESS into our Express apps, we'll move to a better, automated method.

LESS is feature-filled, but it really has five major points:

1. Variables, allowing you to define things like colors once and use them everywhere
2. Functions, allowing you to manipulate variables (like darkening a color by 10%, for example)
3. Nesting selectors, allowing you to structure your stylesheet more like your HTML and reduce repetition
4. Mixins, allowing you to define reusable components and use them in various selectors
5. Includes, allowing you to split your stylesheets into multiple files (much like `require` in Node)

We'll do a *very* quick run-through of these major features. LESS is pretty complicated and we won't talk about every detail. If you're interested in the nitty-gritty features of LESS, take a look at its documentation at http://lesscss.org/.

11.1.1 Variables

CSS doesn't have variables. If your website's link color is `#29A1A4`, for example, and you decide you want to change it to `#454545`, you'd have to search for it everywhere in your CSS file and change it. If you want to experiment with a color that's used in many different places, you'll be doing find-replace, which can lead to various reliability issues. It's also unclear to other developers which color is which; where is that color used in various places?

LESS added variables to CSS, allowing you to solve this kind of problem.

For example, let's say you want to define your site's primary color as `#FF9900`.

In LESS, you might do something like this:

Listing 11.1 Variables in LESS

```
@primary-color: #ff9900;  #A

.logo {
  color: @primary-color;  #B
  font-weight: bold;
}

a {
  color: @primary-color;  #B
}
```

#A Define the variable primary-color.
#B Use that variable in several places.

If you run the LESS code in Listing 11.1 through a LESS compiler (like the one at http://less2css.org/), the following CSS will be produced:

Listing 11.2 The compiled CSS from Listing 11.1

```
.logo {
  color: #ff9900;  #A
  font-weight: bold;
}
a {
  color: #ff9900; #A
}
```

#A Notice that the variable is being inserted here.

As you can see, the variable is being inserted into the resulting CSS. Now, if we want to change the primary color of our site, we only have to do it in one place: the variable at the top.

You might also notice that LESS looks an awful lot like CSS, and that's intentional—it's a strict superset of the language. That means that any valid CSS is valid LESS (but not the other way around). That means that you can easily import your existing CSS stylesheets into LESS and everything will work.

11.1.2 Functions

LESS also has functions, which allow you to manipulate variables and values just like you could in a programming language like JavaScript. Like a typical programming language, there are a number of built-in functions that can help you out. Unlike a typical programming language, however, these functions are *all* built into the language; you *can't* define your own; you'll have to use another feature called "mixins", which we'll talk about in the next section.

LESS has a number of functions that you can use to manipulate colors. For example, imagine your links (your `<a>` tags) have a base color. When you hover over them, they should get lighter. When you click on them, they should get darker. In LESS, functions and variables make this easy.

Listing 11.3 Using functions to lighten and darken colors

```
@link-color: #0000ff;

a {
  color: @link-color;  #A
}
a:hover {
  color: lighten(@link-color, 25%);  #B
}
a:active {
  color: darken(@link-color, 20%);  #C
}
```

#A Use the link-color variable that we've defined before; nothing new here.
#B Lighten the link color by 25%.
#C Darken the link color by 20%.

After we compile this LESS into CSS, we'll get something like the following:

Listing 11.4 The compiled CSS from Listing 11.3

```
a {
  color: #0000ff;
}
a:hover {
  color: #8080ff; #A
}
```

```
a:active {
  color: #000099; #A
}
```

#A Notice that the colors are being manipulated to be lighter and darker.

As you can see, LESS makes it easier to lighten and darken colors. Sure, you could have written that CSS yourself, but choosing finding the lightened and darkened colors would have been a bit of a hassle.

There are a huge slew of other functions built into LESS. http://lesscss.org/functions/ lists them all.

11.1.3 Mixins

Perhaps you're at this section wishing you could define your *own* functions; why does LESS get all of the power? Enter *mixins*, a way of defining reusable CSS declarations that you can use throughout your stylesheets.

Perhaps the most common example is with vendor prefixing. If you want to use the CSS `border-radius` property, you have to prefix it to make sure it works in Chrome, Firefox, Internet Explorer, Safari, and the like. You've probably seen something like this:

```
.my-element {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  -ms-border-radius: 5px;
  border-radius: 5px;
}
```

In CSS, if you want to use border-radius and have it work on all browsers, you'll need the vendor prefixes. And if you want to put those prefixes, you'll have to write all of those *every time* you use border-radius. This can get tedious and is error-prone.

In LESS, rather than define the border-radius and then make several vendor prefixed copies, you can define a mixin, or a reusable component that you can use in multiple declarations. They look an awful lot like functions in other programming languages.

Listing 11.5 Mixins in LESS

```
.border-radius(@radius) {           #A
  -webkit-border-radius: @radius;   #A
     -moz-border-radius: @radius;   #A
      -ms-border-radius: @radius;   #A
          border-radius: @radius;   #A
}                                   #A

.my-element {
  .border-radius(5px);   #B
}
.my-other-element {
  .border-radius(10px); #B
}
```

#A Define the border-radius mixin.
#B Use our border-radius mixin in a couple of elements.

Now, if you run that LESS through a compiler, it produces the following CSS:

Listing 11.6 The compiled CSS from Listing 11.5

```
.my-element {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  -ms-border-radius: 5px;
  border-radius: 5px;
}
.my-other-element {
  -webkit-border-radius: 10px;
  -moz-border-radius: 10px;
  -ms-border-radius: 10px;
  border-radius: 10px;
}
```

As you can see, the mixin is expanded into the tedious vendor-prefixed declarations so that you don't have to write them every time.

11.1.4 Nesting

In HTML, your elements are nested. Everything goes inside the `<html>` tag, and then your content will go into the `<body>` tag. Inside the body, you might have a `<header>` with a `<nav>` for navigation. Your CSS doesn't exactly mirror this; if you wanted to style your header and the navigation inside of your header, you might write some CSS like this:

Listing 11.7 CSS example with no nesting

```
header {
  background-color: blue;
}
header nav {
  color: yellow;
}
```

In LESS, Listing 11.7 would be improved to this:

Listing 11.8 A simple LESS nesting example

```
header {
  background-color: blue;
  nav {                #A
    color: yellow;     #A
  }                    #A
}
```

#A Notice how the styling for the nav is inside of another selector.

LESS improves CSS to allow for nested rulesets. This means that your code will be shorter, more readable, and a better mirror of your HTML.

NESTING THE PARENT SELECTORS

Nested rulesets can refer to their parent element. This is useful in lots of places, and a good example is links and their hover states. You might have a selector for `a`, `a:visited`, `a:hover`, and `a:active`. In CSS, you might do this with four separate selectors. In LESS, you'll define an outer selector and then three inner selectors, one for each link state. It might look something like this:

Listing 11.9 Referring to parent selectors in LESS

```
a {
  color: #000099;
  &:visited {   #A
    color: #330099;
  }
  &:hover {   #A
    color: #0000ff;
  }
  &:active {   #A
    color: #ff0099;
  }
}
```

#A In LESS, you use the & sign to refer to the parent selector.

LESS nesting can do simple things like nesting your selectors to match your HTML, but it can also nest selectors in relation to the parent selectors.

11.1.5 Includes

As your site gets bigger and bigger, you'll start to have more and more styles. In CSS, you can break your code up into multiple files, but this incurs the performance penalty of multiple HTTP requests.

LESS allows you to split up your styles into multiple files, which are all concatenated into one CSS file at compilation time, saving performance. This means that developers can split their variables and mixins into separate files as needed, making for more modular code. You could also make one LESS file for the homepage, one for the user profiles page, and so on.

The syntax is quite simple:

Listing 11.10 Including another LESS file

```
@import "other-less-file";  #A
```

#A Imports "other-less-file.less" in the same folder.

11.1.6 Alternatives to LESS

At this point in the book, it should come as no surprise: there's more than one way to do CSS preprocessing. The elephant in the room is LESS's biggest “rival”, Sass. Sass is very similar to LESS; both have variables, mixins, nested selectors, includes, and integration with Express. As far as the languages go, they're pretty similar. Sass isn't originally a Node project, but it is very popular and has done a solid job integrating itself into the Node world. You can check it out athttp://sass-lang.com/.

Most people reading this book will either want to use LESS or Sass. While we'll use LESS in this book, you can usually substitute the word “LESS” for the word “Sass” and it will be the same. LESS and Sass vary slightly in syntax, but they're largely the same conceptually and in how you integrate them with Express.

There are smaller-time CSS preprocessors that aim to fundamentally change CSS in one way or another. Stylus makes CSS's syntax a lot nicer and Roole adds a number of powerful features, and while they are both great, they aren't as popular as LESS or Sass.

Other CSS preprocessors like Myth and cssnext take a different angle. Rather than try to make a new language that compiles to CSS, they compile upcoming versions of CSS to current-day CSS. For example, the next version of CSS has variables, so these preprocessors compile this new syntax into current-day CSS.

11.2 Using Browserify to require modules in the browser, just like in Node

In short, Browserify is a tool for packaging JavaScript that allows you to use the `require` function just like you do in Node. And I love Browserify. I just want to get that out of the way. Freakin' love this thing.

I once heard someone describe browser-based programming as "hostile." I love making client-side projects, but I must admit that there are a lot of potholes in the road: browser inconsistencies, no reliable module system, an overwhelming number of varying-quality packages, no real choice of programming language...the list goes on. Sometimes it's great, but sometimes it sucks! Browserify solves the module problem in a clever way: it lets you require modules exactly like you would in Node (in contrast to things like RequireJS, which are asynchronous and require an ugly callback). This is powerful for a few reasons.

First, this lets you easily define modules. If Browserify sees that `evan.js` requires `cake.js` and `burrito.js`, it'll package up `cake.js` and `burrito.js` and concatenate them into the compiled output file.

Second, it's almost completely consistent with Node modules. This is huge—both Node-based and browser-based JavaScript can require Node modules, letting you share code between server and client with no extra work. You can even require most native Node modules in the browser, and many Node-isms like `__dirname` are resolved.

I could write sonnets about Browserify. This thing is truly great. Let me show it to you.

11.2.1 A simple Browserify example

Let's say you want to write a webpage that generates a random color and sets the background to that color. Maybe you want to be inspired for the next great color scheme.

We're going to use an npm module called `random-color` (at https://www.npmjs.com/package/random-color), which just generates a random RGB color string. If you check out the source code for this module, you'll see that it knows nothing about the browser—it's only designed to work with Node's module system.

Make a new folder to build this. We'll make a `package.json` that looks something like this (your package versions may vary):

Listing 11.11 package.json for our simple Browserify example

```
{
  "private": true,
  "scripts": {
    "build-my-js": "browserify main.js -o compiled.js"
  },
  "dependencies": {
    "browserify": "^7.0.0",
    "random-color": "^0.0.1"
  }
}
```

Run `npm install` and then create a file called `main.js`. Put this inside:

Listing 11.12 main.js for our simple Browserify example

```
var randomColor = require("random-color");
document.body.style.background = randomColor();
```

Note that this file uses the `require` statement, but it's made for the browser, which doesn't have that natively. Get ready for your little mind to be blown!!

Finally, define a simple HTML file in the same directory with the following contents (it doesn't matter what you call it, so long as it ends in `.html`):

Listing 11.13 HTML file for our simple Browserify example

```
<!DOCTYPE html>
<html>
<body>
  <script src="compiled.js"></script>
</body>
</html>
```

Now, if you save all that and run `npm run build-my-js`, Browserify will compile `main.js` into a new file, `compiled.js`. Open the HTML file you saved to see a webpage that generates random colors every time you refresh!

You can open `compiled.js` to see that your code is there, as is the `random-color` module. The code will be ugly, but here's what it looks like:

```
(function e(t,n,r){function s(o,u){if(!n[o]){if(!t[o]){var a=typeof
require=="function"&&require;if(!u&&a)return a(o,!0);if(i) return i(o,!0);var f=new
Error("Cannot find module '"+o+
"'");throw f.code="MODULE_NOT_FOUND",f}var l=n[o]={ exports:{}};t[o]
[0].call(l.exports,function(e){var n=t[o] [1][e];return s(n?
n:e)},l,l.exports,e,t,n,r)}return n[o].exports}var i=typeof
require=="function"&&require;for(var o=0;o<r.length;o++)s(r[o]); return s})({1:
[function(require,module,exports){ var randomColor = require("random-color");
document.body.style.backgroundColor = randomColor();
},{"random-color":2}],2:[function(require,module,exports){ var random = require("rnd");
module.exports = color;
function color (max, min) {   max || (max = 255);   return 'rgb(' + random(max, min) +
', ' + random(max, min) + ', ' +   random(max, min) + ')'; }
},{"rnd":3}],3:[function(require,module,exports){ module.exports = random;
function random (max, min) {   max || (max = 999999999999);   min || (min = 0);
  return min + Math.floor(Math.random() * (max - min)); }
},{}]},{},[1]);
```

They're both wrapped in a bit of Browserify stuff to fake Node's module system, but they're there...and most importantly, they work! You can now require Node modules in the browser.

Browserify is so great. Love it.

> **NOTE** While you can require a number of utility libraries (even the built-in ones), there are some things you can't fake in the browser and therefore can't use in Browserify. For example, you can't run a web server in the browser, so some of the httpmodule is off-limits. But many things like `util` or modules you write are totally fair game!

As you write your code with Browserify, you'll want a nicer way to build this than having to run the build command every single time. Let's check out a tool that helps us use Browserify, LESS, and much much more.

11.3 Using Grunt to compile, minify, and more

We've taken a look at LESS and Browserify, but we haven't found an elegant way to wire them into our Express apps yet.

We'll look at two ways to handle this, the first of which is with the use of a tool called Grunt. Grunt (at http://gruntjs.com/) calls itself "The JavaScript Task Runner", which is exactly what it sounds like: it runs tasks. If you've ever used Make or Rake, Grunt will seem familiar.

Grunt defines a framework onto which you define tasks. Like Express, Grunt is a minimal framework. It can't do much alone; you'll need to install and configure other tasks for Grunt to run. These tasks include compiling CoffeeScript or LESS or SASS, concatenating JavaScript and CSS, running tests, and plenty more. You can find a full list of tasks at http://gruntjs.com/plugins, but we'll be using four today: compiling and concatenating JavaScript with Browserify, compiling LESS into CSS, minifying JavaScript and CSS, and using "watch" to keep us from typing the same commands over and over again.

Let's start by installing Grunt.

11.3.1 Installing Grunt

These instructions will deviate from the official Grunt instructions a bit. The documentation will tell you to install Grunt globally, but I believe that you should install everything locally if you can. This allows you to install multiple versions of Grunt on your system and doesn't pollute your globally installed packages. We'll talk more about these best practices in Chapter 12.

Every project has a `package.json`. If you want to add Grunt to a project, you'll want to define a new script so that you can run the local Grunt:

Listing 11.14 A script for running the local Grunt

```
...
"scripts": {
  "grunt": "grunt"
}, ...
```

If you'd like to follow along with these examples, you can make a new project with a barebones package.json like this one:

Listing 11.15 A barebones package.json for these examples

```
{
  "private": true,
  "scripts": {
    "grunt": "grunt"
  }
}
```

Grunt isn't set up yet, but when it is, this allows us to say npm run grunt to run the local Grunt.

Next, you'll want to npm install grunt --save-dev and npm install grunt-cli --save-dev (or just npm install grunt grunt-cli --save-dev) to save Grunt and its command-line tool as local dependencies.

Next, you'll want to create a something called a "Gruntfile", which Grunt examines to figure out what it should do. The Gruntfile lives at the root of your project (in the same folder as your package.json) and is called Gruntfile.js.

Here's a "hello world" Gruntfile. When you run Grunt, it will look at this Gruntfile, find the appropriate task, and run the code inside.

Listing 11.16 A skeleton Gruntfile

```
module.exports = function(grunt) {

  grunt.registerTask("default", "Say hello world.", function() {
    grunt.log.write("Hello world!");
  });

};
```

To try this out, type `npm run grunt` into your terminal. You should see the following output:

```
Running "default" task Hello world!
Done, without errors.
Grunt is now running the "hello world" task!
```

Unfortunately, "hello world" isn't of much use to us. Let's look at some more useful tasks we can define. If you'd like to follow along, you can take a look at this book's code samples at https://github.com/EvanHahn/Express.js-in-Action-code/tree/master/Chapter_11/grunt-examples.

11.3.2 Compiling LESS with Grunt

When we learned about LESS above, I recommended a website that compiled your code live, in front of you. That's great for learning and it's useful to make sure your code is being compiled correctly, but it's hardly an automated solution. You don't want to have to put all of your code into a website, copy-paste the resulting CSS, and copy it into a CSS file! Let's make Grunt do it. (If you're not using LESS, there are other Grunt tasks for your favorite preprocessor. Just search the Grunt plugins page at http://gruntjs.com/plugins.)

Let's start by writing a very simple LESS file, which we'll compile to CSS with Grunt.

Listing 11.17 A simple LESS file (in my_css/main.less)

```
article {
  display: block;
  h1 {
    font-size: 16pt;
    color: #900;
  }
  p {
    line-height: 1.5em;
  }
}
```

That should translate to the following CSS:

Listing 11.18 Listing 11.17 compiled to CSS

```
article {
  display: block;
}
article h1 {
  font-size: 16pt;
  color: #900;
}
article p {
  line-height: 1.5em;
}
```

And if we minify that CSS, it should look like this:

Listing 11.19 Listing 11.18, minified

```
article{display: block}article h1{font-size:16pt; color:#900}article p{line-
height:1.5em}
```

We can use a third-party LESS task for Grunt to get us there! Start by installing this Grunt LESS task with `npm install grunt-contrib-less --save-dev`. Next, add the following to your Gruntfile:

Listing 11.20 A Gruntfile with LESS

```
module.exports = function(grunt) {

  grunt.initConfig({  #Z
    less: {  #A
      main: {                              #B
        options: {                             #B
          paths: ["my_css"]                    #B
        },                                     #B
        files: {                               #B
          "tmp/build/main.css": "my_css/main.less"  #B
        }                                      #B
      }
    }
  });

  grunt.loadNpmTasks("grunt-contrib-less");  #D

  grunt.registerTask("default", ["less"]);  #E

};
```

#Z grunt.initConfig configures settings for each of your Grunt tasks. In this case, we're only configuring LESS right now.

#A We define the configuration for our LESS tasks. This is what the Grunt LESS task will look at.

#B Define the compilation configuration. This configuration tells the Grunt LESS plugin to compile my_css/main.less into tmp/build/main.css.

#D This loads the Grunt LESS plugin. Without this, we won't be able to compile anything!

#E This tells Grunt to run the LESS compilation task when we run “grunt” at the command line.

Now, when you run Grunt `npm run grunt`, your LESS will be compiled into `tmp/build/main.css`. After doing that, you'll need to make sure to serve that file.

SERVING THESE COMPILED ASSETS

Now that we've compiled something, we actually need to serve it to our visitors! We'll use Express's static middleware to do that. We'll just add `tmp/build` as part of our middleware stack. For example:

Listing 11.21 Static middleware with compiled files

```
var express = require("express");
var path = require("path");

var app = express();

app.use(express.static(path.resolve(__dirname, "public")));
app.use(express.static(path.resolve(__dirname, "tmp/build")));

app.listen(3000, function() {
  console.log("App started on port 3000.");
});
```

Now, you can serve files from public and compiled files from `tmp/build`!

> **NOTE** You likely don't want to commit compiled files into your repository, so you have to store them into a directory that you'll later ignore with version control. If you're using Git, add `tmp` to your `.gitignore` to make sure that your compiled assets aren't put into version control. Some people *do* like to commit these, so do what's right for you.

11.3.3 Using Browserify with Grunt

Browserify, in its wisdom, has Grunt integration, so we can automate the process of compiling our client-side JavaScript. Browserify...what an amazing piece of technology.

Start by installing `grunt-browserify`, a Grunt task for Browserify. Install it by running `npm install grunt-browserify --save-dev`, and then fill in Gruntfile.js with this:

Listing 11.22 A Gruntfile with Browserify

```
module.exports = function(grunt) {

  grunt.initConfig({
    less: { /* ... */ }, #A
    browserify: {        #B
      client: {                          #C
        src: ["my_javascripts/main.js"],   #C
        dest: "tmp/build/main.js",          #C
      }
    }
  });

  grunt.loadNpmTasks("grunt-contrib-less");
  grunt.loadNpmTasks("grunt-browserify");    #D

  grunt.registerTask("default", ["browserify", "less"]);  #E
};
```

#A Note that we can keep our LESS configuration in here; typical Gruntfiles often have many bits of configuration.
#B Start configuring Browserify...
#C Compile main.js file from my_javascripts into tmp/build/main.js.
#D Load the grunt-browserify task.
#E When we run "grunt" at the command line, run both Browserify and LESS.

Now, when you run Grunt with `npm run grunt`, this will compile `main.js` in a folder called `my_javascripts` into `tmp/build/main.js`. If you've followed the steps from the LESS guide above, this should already be served!

11.3.4 Minifying the JavaScript with Grunt

Unfortunately, Browserify doesn't minify your JavaScript for you; its only blemish. We'd like to do that to reduce file sizes and load times as best we can.

UglifyJS is a popular JavaScript minifier that crushes your code down to tiny sizes. We'll be a Grunt task that takes advantage of UglifyJS to minify your already-Browserified code, called `grunt-contrib-uglify`. You can read more about it at https://www.npmjs.com/package/grunt-contrib-uglify.

First, install the Grunt task as usual with `npm install grunt-contrib-uglify --save-dev`. Next, let's add this to our Gruntfile:

Listing 11.23 A Gruntfile with Browserify, LESS, and Uglify

```
module.exports = function(grunt) {

  grunt.initConfig({
    less: { /* ... */ },         #A
    browserify: { /* ... */ },   #A
    uglify: {                    #B
      myApp: {                   #B
        files: {                 #B
          "tmp/build/main.min.js": ["tmp/build/main.js"]  #B
        }                        #B
      }                          #B
    }                            #B
  });

  grunt.loadNpmTasks("grunt-browserify");
  grunt.loadNpmTasks("grunt-contrib-less");
  grunt.loadNpmTasks("grunt-contrib-uglify");

  grunt.registerTask("default", ["browserify", "less"]);  #C
  grunt.registerTask("build", ["browserify", "less", "uglify"]);  #C

};
```

#A As before, we've left the existing LESS and Browserify tasks.
#B This compiles your compiled JavaScript into a minified version. You can also overwrite the full JavaScript if you'd like: just set them both to "tmp/build/main.js".
#C We've defined a new task called "build" in addition to our existing task. This will run when we type "npm run grunt build".

`npm run grunt` won't do anything different than it did before—it'll run the default task, which in turns runs the Browserify and LESS tasks. But when you run `npm run grunt` **`build`**, you'll run both the Browserify task and the Uglify task. Now your JavaScript will be minified!

11.3.5 "grunt watch"

While you're developing, you don't want to have to run `npm run grunt` every time you edit a file. There's a Grunt task that watches your files and re-runs any Grunt tasks when a change occurs. Enter `grunt-contrib-watch`. Let's use it to auto-compile any CSS and JavaScript whenever they change.

Start by installing the task with `npm install grunt-contrib-watch --save-dev`, then add some stuff to your Gruntfile like so:

Listing 11.24 A Gruntfile with watching added

```
module.exports = function(grunt) {

  grunt.initConfig({
    less: { /* ... */ },
    browserify: { /* ... */ },
    uglify: { /* ... */ },
    watch: {
      scripts: {                #A
        files: ["**/*.js"],     #A
        tasks: ["browserify"]   #A
      },                        #A
      styles: {                 #B
        files: ["**/*.less"],   #B
        tasks: ["less"]         #B
      }                         #B
    }
  });

  grunt.loadNpmTasks("grunt-browserify");
  grunt.loadNpmTasks("grunt-contrib-less");
  grunt.loadNpmTasks("grunt-contrib-uglify");
  grunt.loadNpmTasks("grunt-contrib-watch");

  grunt.registerTask("default", ["browserify", "less"]);
  grunt.registerTask("build", ["browserify", "less", "uglify"]);
};
```

#A Tell the Grunt watch task to run the Browserify task any time a .js file changes.
#B Tell the Grunt watch task to run the LESS task any time a .less file changes.
#C Register the new watch task to execute when you run “grunt watch”.

In the above example, we specify all files to watch and tasks to run when they change—it's that simple. Now, when you run `npm run grunt watch`, Grunt will watch your files and compile your CSS/JavaScript as needed. For example, if you change a file with the `.less` file extension, the LESS task will run (but no other tasks will); this is because we've configured `.less` files to trigger that task.

I find this super useful for development and strongly recommend it.

11.3.6 Other helpful Grunt tasks

We've looked at a few Grunt tasks here, but there are loads more. You can find the full list on Grunt's website at http://gruntjs.com/plugins, but here are a few that might be helpful at some point:

- grunt-contrib-sass is the Sass version of the LESS plugin we used. If you'd rather use Sass or SCSS, give this a look.
- grunt-contrib-requirejs uses the Require.js module system instead of Browserify. If that sounds better to you, you can use it instead.
- grunt-contrib-concat simply concatenates files, which is a low-tech but popular solution for lots of problems.
- grunt-contrib-imagemin minifies images (like JPEGs and PNGs). If you want to save bandwidth, this is a good tool.
- grunt-contrib-coffee lets you write CoffeeScript instead of JavaScript for your client-side code.

11.4 Using connect-assets to compile LESS and CoffeeScript and more

I don't love Grunt, to be quite honest. I include it in the book because it's incredibly popular and powerful, but I find the code verbose and a little confusing. There's another solution for Express users: a piece of middleware called connect-assets (at https://github.com/adunkman/connect-assets).

connect-assets can concatenate, compile to, and minify JavaScript and CSS. It supports CoffeeScript, Stylus, LESS, SASS, and even some EJS. It doesn't support Browserify and isn't as configurable as build tools like Grunt or Gulp, but it's very easy to use.

connect-assets is heavily inspired by the Sprockets asset pipeline from the Ruby on Rails world. If you've used that, this will be quite familiar, but if you haven't, don't worry!

> **A REMINDER ABOUT CONNECT** Connect is another web framework for Node, and in short, Express middleware is compatible with Connect middleware. A lot of Express-compatible middleware has "connect" in the name like connect-assets.

11.4.1 Getting everything installed

You'll need to `npm install connect-assets --save` and any other compilers you'll need:

- coffee-script for CoffeeScript support
- stylus for Stylus support
- less for LESS support
- node-sass for SASS support
- ejs for some EJS support

The last two won't be used by default in development mode, but will be in production. If you don't change the default options and forget to install those, your app will fail in production. Make sure to get those installed! For example, to install LESS, run `npm install less --save`.

You'll also need to pick a directory for your assets to live. By default, connect-assets will look for your CSS-related assets in `assets/css` and your JavaScript-related assets in `assets/js`, but this is configurable. I recommend using the defaults while you're getting started, so make a directory called `assets` and put the `css` and `js` directories inside.

11.4.2 Setting up the middleware

The middleware has some quick-start options that make it easy to get started, but I strongly recommend configuring things. For example, one of the configuration options can keep connect-assets from muddying the global namespace, which it does by default. Here's what a simple application setup might look like:

Listing 11.25 Setting up the connect-assets middleware

```
var express = require("express");
var assets = require("connect-assets");

var app = express();
app.use(assets({
   helperContext: app.locals, #A
   paths: ["assets/css", "assets/js"] #B
 });

// ...
```

#A This attaches connect-assets's view helpers to app.locals, rather than making them global variables.
#B Specify any asset paths you're using. Order matters here—if main.js exists in multiple directories, for example, it'll only compile the one listed first.

This middleware has a number of sensible defaults. For example, it will enable minification and caching in production, but disable them in development. You can override this configuration if you truly want to; check the documentation for more detailed instructions.

Now that we've set up the middleware, we'll need to link to those assets from views.

11.4.3 Linking to assets from views

connect-assets provides two major helper functions to your views: `js` and `css`. `js("myfile")` will generate a `<script>` tag that corresponds to `myfile`. The `css` helper will do the same but for CSS, with a `<link>` tag. They return the HTML to include the most recent version of your assets, which means that they'll append a long hash to the name to make sure your browser doesn't use old cached assets.

If you're using Jade to render your views, you'll reference them from your views like this:

Listing 11.26 Linking to connect-assets assets from Jade

```
!= css("my-css-file")
!= js("my-javascript-file")
```

If you're using EJS instead, it's pretty similar. You reference connect-assets's helpers from your views like this:

Listing 11.27 Linking to connect-assets assets from EJS

```
<%- css("my-css-file") %>
<%- js("my-javascript-file") %>
```

If you're using another view engine, you'll need to make sure you aren't escaping HTML when you do this, because these helpers are spitting out raw HTML tags that shouldn't be escaped.

In any case, these will spit out something like this:

Listing 11.28 HTML generated by connect-assets

```
<link rel="stylesheet" href="/assets/my-css-file-{{SOME LONG HASH}}.css">
<script src="/assets/my-javascript-file-{{SOME LONG HASH}}.js>
```

And your assets will be loaded!

11.4.4 Concatenating scripts with directives

You can't concatenate CSS files this way. Instead, you should use the `@import` syntax in your CSS preprocessor (like LESS or Sass). But connect-assets lets you concatenate JavaScript files using specially-formatted comments.

Let's say that your JavaScript file requires jQuery. All you have to do is define a comment that starts with `//= require` and then connect-assets will concatenate those files for you magically.

Listing 11.29 main.js, which requires jQuery

```
//= require jquery
$(function() {
  // do what you need to do with jQuery
```

And that's concatenation! It's that easy.

Now that we've looked at two ways to compile our assets, let's look at how to deploy our applications to the real web with Heroku!

11.5 Deploying to Heroku

Heroku's website has buzzwords like “cloud platform” and “built by developers for developers”. To us, it's a way to deploy our Node.js applications onto the real internet for free. No more `localhost:3000`! You'll be able to have your apps on the real life internet.

Essentially, when you deploy your site, you're sending your code to be run somewhere. In this case, when we deploy to Heroku, we'll be sending out code to Heroku's servers and they'll run your Express applications.

Like everything, there are a lot of ways to deploy your site. Heroku may not be the best option for you, and you should explore all of your options. We choose it here because it's relatively simple and it costs nothing.

11.5.1 Getting Heroku set up

First, you'll need to get a Heroku account. Visit Heroku.com and sign up (if you don't already have an account). The signup process should be fairly straightforward if you've ever signed up for any account online.

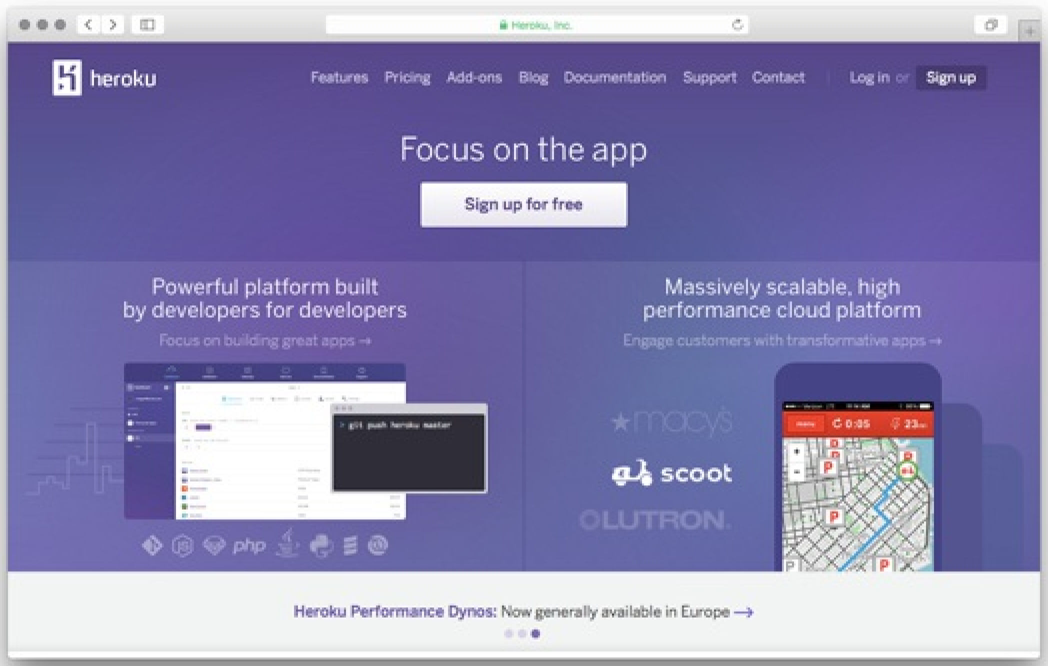

heroku
Features
Pricing
Add-ons
Blog
Documentation
Support
Contact
Log in or
Sign up
Focus on the app
Sign up for free
Powerful platform built by developers for developers
Focus on building great apps →
Massively scalable, high performance cloud platform
Engage customers with transformative apps →
Heroku Performance Dynos: Now generally available in Europe →

Figure 11.2 Heroku's homepage.

Next, you'll want to download and install the Heroku Toolbelt from https://toolbelt.heroku.com/. Follow the instructions for your specific operating system. Installing the Heroku Toolbelt on your computer will install three things:

1. The Heroku client, a command-line tool for managing Heroku apps. We'll use it to create and manage our Express apps.
2. Foreman, another command-line tool. We'll use it to define how we want our applications to run.
3. Git, the version control system that you may already have installed.

Once you've installed it, there's one last thing to do: authenticate your computer with Heroku. Open up a command line and type `heroku login`. This will ask you for your Heroku username and password.

Once you've done all that, Heroku should be set up!

11.5.2 Making a Heroku-ready app

Let's make a simple hello world application and deploy it to Heroku, shall we?

To set your app up for Heroku, you don't have to do too much different from what you normally would. While there are a few commands you'll need to run in order to deploy, the only changes you'll need to make are as follows:

1. Make sure to start the app on `process.env.PORT`.
2. Make sure your `package.json` lists a Node version.
3. Create a file that will be run when Heroku starts your app (called a Procfile). In our simple app, this file will only be one line.
4. Add a file called `.gitignore` to your project.

Let's make a simple app and make sure we cross off these things.

The Express part of this "hello world" application should be pretty easy for you at this point in the book, and there's not much special we have to do in order to make sure that it works for Heroku; it's only a line or two.

First, define your `package.json`:

Listing 11.30 package.json for our Heroku Express app

```
{
  "private": true,
  "scripts": {
    "start": "node app"
  },
  "dependencies": {
    "express": "^4.10.4"
  },
  "engines": {        #A
    "node": "0.10.x" #A
  }                   #A
}
```

#A This tells Heroku (and anyone running your app) that your app requires Node 0.10. This helps Heroku disambiguate.

Nothing too new there, but for the definition of which Node version to use. Next, define app.js, where our Hello World code resides:

Listing 11.31 A Hello World Express app (app.js)

```
var express = require("express");

var app = express();

app.set("port", process.env.PORT || 3000);

app.get("/", function(req, res) {
  res.send("Hello world!");
});

app.listen(app.get("port"), function() {
  console.log("App started on port " + app.get("port"));
});
```

Once again, not much new here. The only Heroku-specific thing here is how the port is set. Heroku will set an environment variable for the port which we'll access through `process.env.PORT`. If we never deal with that variable, we won't be able to start our app on Heroku on the proper port.

The next part is the most foreign thing we've seen so far: a `Procfile`. It might sound like a complicated new Heroku concept, but it's really simple. When you run your app, you type `npm start` into your command line. The Procfile codifies that, and tells Heroku to run npm start when your app begins. Create a file in the root of your directory and call it `Procfile` (capital P, no file extension):

Listing 11.32 Our application's Procfile

```
web: npm start
```

That's not too bad, right? Heroku is pretty nice.

Finally, we need to add a file that tells Git to ignore certain files. We don't need to push `node_modules` to our server, so let's make sure we ignore that file:

Listing 11.33 Our application's .gitignore file

```
node_modules
```

Now that we've got our application all ready to go, let's deploy it!

11.5.3 Deploying our first app

The first thing we'll need to do, if you haven't done it already, is put your app under version control with Git. I'm going to assume you at least know the basics of Git, but if you don't, check out Try Git at https://try.github.io.

To set up a Git project in this directory, type `git init`. Then use `git add .` to add all of your files and `git commit -m "Initial commit"` to commit those changes to your Git project. Once that's all ready to go, type the following command:

```
heroku create
```

This will set up a new URL for your Heroku app. The names it generates are always a bit wacky—I got *mighty-ravine-4205.herokuapp.com*—but that's the price you pay for free hosting! You can change the URL or associate a domain name you own with a Herkou address, but we won't go into that here.

Next, we'll want to tell our newly-created Heroku app that it's a production Node environment. We'll do this by setting the `NODE_ENV` environment variable on Heroku's servers. Set that variable by running this command:

```
heroku config:set NODE_ENV=production
```

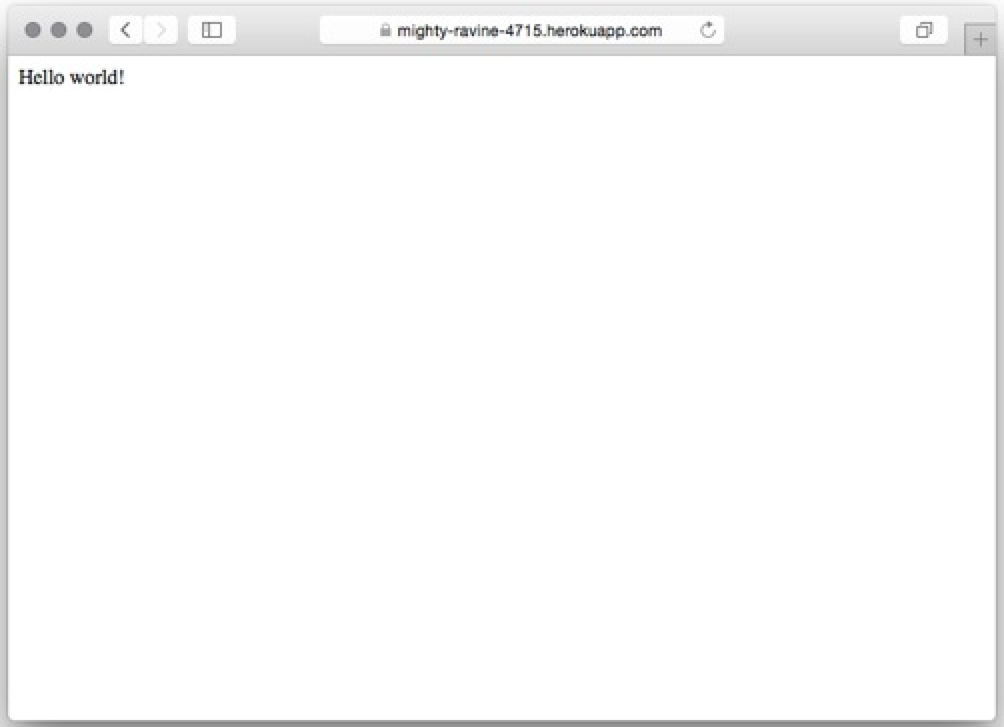
When you ran `heroku create`, Heroku added a remote Git server. When you push your code to Heroku, Heroku will deploy your app (or redeploy it if you've already deployed). This is just one Git command:

```
git push heroku master
```

This will first push your code to Heroku's servers, then set up their servers with all of your dependencies. You'll run `git push heroku master` every time you want to re-deploy; that's really the only command you'll run more than once. There's just one last thing to do: tell Heroku that it should run your app with one process so that it'll actually run on a real computer:

```
heroku ps:scale web=1
```

Suddenly, your app will be running on the real internet! You can type `heroku open` to open your app in your browser, and see it running! You can send this link to your friends. No more localhost, baby!



```
Hello world!
```

Figure 11.3 Your “hello world” app running on Heroku!

11.5.4 Running Grunt on Heroku

If you're using connect-assets to compile your assets, then Heroku will work just fine (assuming you've installed all of the dependencies properly). But if you want to use Grunt (or another task runner like Gulp), you'll need to run Grunt to build your assets when you deploy your site.

There's a little trick you can use to make this work, which leverages a nice little feature of npm: the post-install script. Heroku will run `npm install` when you deploy your app, and we can tell Heroku to run Grunt right after that in order to build all of our assets. This is a simple manner of adding another script to our `package.json`:

Listing 11.33 Running Grunt in a postinstall script

```
// …
"scripts": {
  // …
  "postinstall": "grunt build"   #A
},
// …
```

#A I use "grunt build" as an example—you could run whatever Grunt command you'd like.

Now, when anyone (including Heroku!) runs `npm install`, `grunt build` will run.

11.5.5 Making your server more crash-resistant

No offense, but your server might just crash.

It could be that you run out of memory, or that you have an uncaught exception, or that a user has found a way to break your server. If you've ever had this happen while you're developing, you've probably seen that an error sends your server process screeching to a halt. While you're developing, this is pretty helpful—you want to be aware that your app doesn't work! In production, however, it's much more likely that you want your app to work at all costs. If you have a crash, you'll want your app to be resilient and restart.

We've already seen Forever in our chapter about security, but a refresher: it is a tool to keep your server up and running, even in the face of crashes. Instead of typing `node app.js`, you'll just type `forever app.js`. Then, if your app crashes, Forever will restart it.

First, run `npm install forever --save` to install Forever as a dependency. Now, we need to run `forever app.js` to start our server. We could add this to the Procfile or change our `npm start` script, but I like to add a new script to`package.json`.

Open up your scripts in `package.json` and add the following:

Listing 11.34 Defining a script for running your server in production

```
// …
"scripts": {
  // …
  "production": "forever app.js"
},
// …
```

#A When you run “npm run production”, your app will start running forever.

Now, when you run `npm run production`, your app will start with Forever. The next step is to get Heroku to run this script, and that's just a simple matter of changing your Procfile:

Listing 11.35 Updating your Procfile to use Forever

```
web: npm run production
```

Now, when Heroku starts, it'll run your app with Forever and keep your app restarting after crashes!

As always with Heroku, commit these changes into Git. (You'll need to add your files with `git add .` and then commit them with `git commit -m "Your commit message here!"`.)Once that's done, you can deploy them to Heroku with`git push heroku master`.

You can use Forever in any kind of deployment, not just Heroku. While Heroku uses Procfiles and some of that will have to change depending on your server setup, you can use Forever wherever you choose to deploy.

11.6 Summary

In this chapter, we've learned a ton of stuff about compiled files, from views to concatenated JavaScript. We've seen the following:

- LESS for improving our CSS with a number of features.
- Browserify for packaging JavaScript, letting us share code between our client and our server.

- The flexible Grunt task runner, and a few of its many tasks.
- connect-assets as an alternative to Grunt for compiling and serving CSS and JavaScript.
- Deploying our applications to Heroku for the real internet!

12 Best practices

It's time to bring this book to a close.

If this book were a tragedy, we'd probably end with a dramatic death. If it were a comedy, we might have a romantic wedding. Unfortunately, this is a book about Express, a topic not known for its drama and romance. The best we'll get is this: a set of best practices for large Express applications. I'll do my best to make it romantic and dramatic.

With small applications, organization doesn't matter much. You can fit your app in a single file or a handful of small files. But as your apps become larger, these considerations become more important. How should you organize your files so that your codebase is easy to work with? What kind of conventions should you adhere to in order to best support a team of developers?

In this final chapter, I'll do my best to share my experience. Very little of this chapter will be strictly factual; I'll do my best to lend opinions to the unopinionated philosophy of Express with respect to what it takes to build a medium-to-large application with it. **We'll see**:

- High-level goals of simplicity
- Folder and file structures for your applications
- Locking down dependency versions for maximum reliability
- Installing dependencies locally and using npm scripts

I'll make sure to repeat this disclaimer, but remember: *this chapter is mostly opinions and conventions I've found. You may disagree or find that your application doesn't fit into these moulds.* That's the beauty of Express—you have a lot of flexibility.

This might not be as emotional as a comedy or a tragedy, but I'll do my best.

12.1 Simplicity

In this chapter of my opinions, let me offer an overarching one before we delve into specifics.

There are lots of best practices for maintaining large codebases, but I think they all boil down to one thing: *simplicity*. More explicitly, your code should be easy for other developers to follow and you should minimize how much context a person has to keep in their head.

In order to understand an Express application, you already have to know a lot. You have to be reasonably proficient in the JavaScript programming language in order to read the code; you have to understand how HTTP works in order to understand routing; you have to understand Node and its evented I/O; and then you have to understand all of Express's features like routing, middleware, views, and more. Each of these things took a long time to learn and likely builds on experience from earlier in your career. It's a huge pile of stuff to keep in your head!

Your applications should try to add to that massive pile of required knowledge as little as possible.

I think we've all written code (I certainly have) that's an intertwined mess that only we can hope to understand. I like to imagine one of those corkboards covered with pictures, all interconnected in a web of red string. Here are a couple of ways to see how deep the rabbit hole of your code goes.

- Look at a piece of your code—maybe it's a route handler or a middleware function—and ask yourself how many other things you'd need to know in order to understand it. Does it depend on a middleware earlier in the stack? How many different database models does it depend on? How many routers deep are you? How many files have you looked at to get to this point?
- How confused is your fellow developer? How quickly could they add a feature to your app? If they're confused and unable to work quickly, that might mean that your code is too intertwined.

You have to be pretty rigorous about simplicity, especially because Express is so flexible and unopinionated. We'll talk about some of these methods (and others) in this chapter, but a lot of it is more nebulous, so keep this in mind!

Alright, enough with this abstract stuff! Let's talk about specifics.

12.2 File structure pattern

Express applications can be organized however you please. You could put everything into one giant file if you wanted to. As you might imagine, this might not make for an easily-maintainable application.

Despite the fact that it's unopinionated, most Express applications I've worked with have a similar structure to the one in figure 12-1.

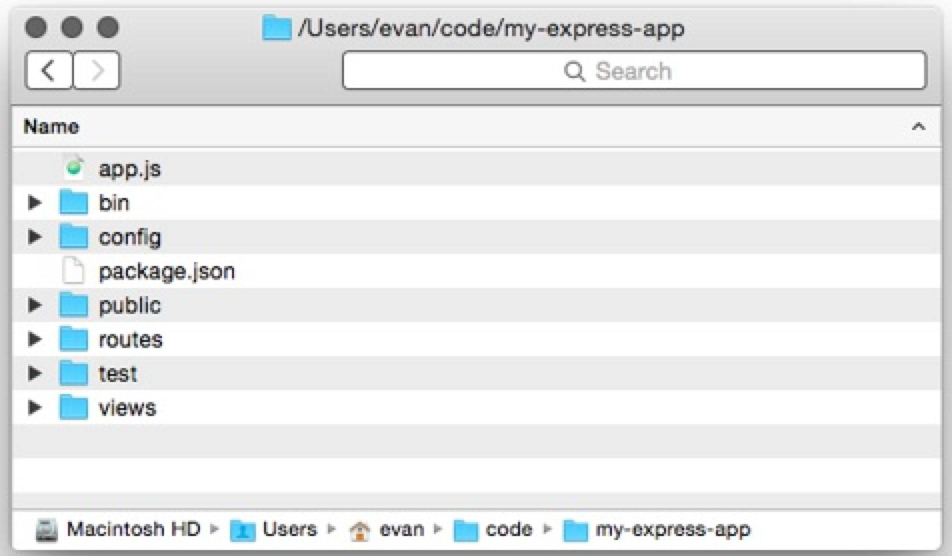


Figure 12-1 A common folder structure for Express applications.

Here are all of the common files in an Express application of this structure:

- `package.json` should come as no surprise—it's present in every Node project. This will have all of the app's dependencies as well as all of your npm scripts. We've seen different incarnations of this file all throughout the book and it's not different in a "big" app.
- `app.js` is the main application code—the entry point. This is where you actually call `express()` to instantiate a new Express application. This is where you put middleware that's common to all routes, like security or

static file middleware. This file doesn't start the app, as we'll see—it assigns the app to `module.exports`.

- `bin/` is a folder that holds executable scripts relevant to your application. There is often just one (listed below), but sometimes there are more that are required.
- `bin/www` is an executable Node script that `require`s your app (from `app.js` above) and starts it. Calling `npm start` runs this script.
- `config/` is a folder that'll hold any configuration for your app. It's often full of JSON files that specify things like default port numbers or localization strings.
- `public/` is a folder that's served by static file middleware. It'll have any static files inside – HTML pages, text files, images, videos etc. The HTML5 Boilerplate at https://html5boilerplate.com/ for example presents a good selection of common static files you might add here.
- `routes/` is a folder that holds a bunch of JavaScript files, each one exporting an Express Router. You might have a router for all URLs that start with `/users` and another for all that start with `/photos`. Chapter 5 has all the details about routers and routing—check out section 5.3 for examples of how this works.
- `test/` is a folder that holds all of your test code. Chapter 9 has all the juicy details about this.
- `views/` is a folder that holds all of your views. Typically they are written in EJS or Jade as shown in Chapter 7 but there are many other templating languages you can use.

The best way to see an app that has most of these conventions is by using the official Express application generator. You can install this with `npm install -g express-generator`. Once it's installed, you can run `express my-new-app`and it'll create a folder called `my-new-app` with a skeleton app set up, as shown in figure 12-1.

While these are just patterns and conventions, patterns like this tend to emerge in Express applications I have seen.

12.3 Locking down dependency versions

Node has far and away the best dependency system I've used. A coworker of mine said the following to me about Node and npm: "They *nailed* it."

npm uses something called *Semantic Versioning* (sometimes shortened to Semver) for all of its packages. Versions are broken up into three numbers: major, minor, and patch. For example, version 1.2.3 is major version 1, minor version 2, and patch version 3.

In the rules of Semantic Versioning, a major version upgrade can have a change that is considered "breaking". A *breaking change* is one where old code wouldn't be compatible with new code. For example, code that worked in Express major version 3 doesn't necessarily work with major version 4. Minor version changes are, by contrast, *not* breaking. They generally mean a new feature that doesn't break existing code. Patch versions are for, well, patches--they're reserved for bug fixes and performance enhancements. Patches shouldn't break your code; they should generally make things better.

> **MAJOR VERSION ZERO** There's one asterisk to this: basically anything goes if the major version is 0. The whole package is considered to be unstable at that point.

By default, when you `npm install --save` a package, it downloads the latest version from the npm registry and then puts an "optimistic" version number in your `package.json` file. That means that if someone else on your team runs `npm install` in the project (or if you are reinstalling), they might get a newer version that the one you originally downloaded. That new version can have a higher minor version or higher patch version, but it can't have a higher major version. That means that it doesn't download the absolute latest version of a package; it downloads the latest version that should still be compatible. Figure 12-2 expands on this.

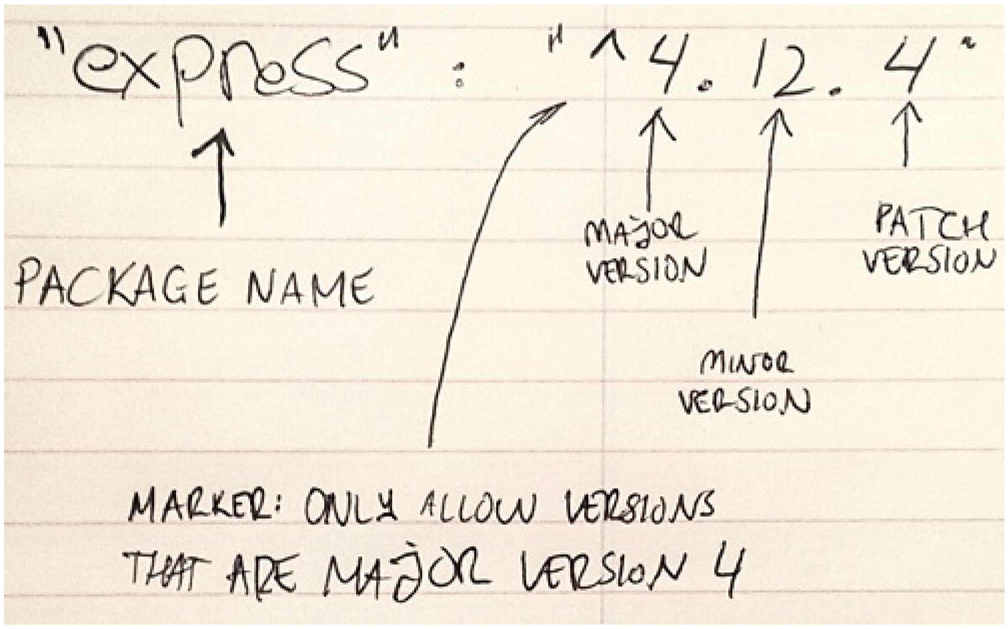


Figure 12-2 How optimistic versioning looks in package.json.

All good, right? If all packages adhere to Semantic Versioning, you should always want to get the latest compatible version so that you have all the latest features and have all the newest bug fixes.

But here's the rub: not all packages adhere perfectly to Semantic Versioning. Usually, it's because people use packages in ways the original developers don't intend. Perhaps you're relying on an untested feature or weird quirk in the library that is overlooked by the developers. You can't really blame these people--no programmer has a clean, bug-free track record, especially when other developers are using their code in unexpected ways.

I find that 99% of the time, this isn't an issue. The modules I use tend to be good about Semantic Versioning and npm's optimistic versioning works well. But when I'm deploying a business-critical application into production (also known as "the real world"), I like to lock down my dependency versions to minimize any potential hiccups. I don't want things to break with a new version of a package!

There are two ways to lock versions down: one is simple but less thorough and the other is very thorough.

12.3.1 The simple way: eschewing optimistic versioning

A quick way to solve this problem is by obliterating optimistic versioning in your `package.json`.

Optimistic versioning in your `package.json` file might look something like this:

Listing 12.1 Example of optimistic versioning in your package.json

```
// ...
"dependencies": {
  "express": "^4.12.4" #A
}
// ...
```

#A The ^ character indicates optimistic versioning is allowed.

If you are going back and editing your `package.json`, you can simply specify the dependency to an exact version. The example above would look like this:

Listing 12.2 Example of omitting optimistic versioning in a package.json

```
// ...
"dependencies": {
  "express": "4.12.4"  #A
}
// ...
```

#A Removing the ^ character from the version number indicates only that specific version of the package should be downloaded and used.

These edits are relatively easy to do and can lock a package down to a specific version.

If you're installing *new* packages, you can turn off npm's optimistic versioning by changing the `--save` flag to `-save-exact`. For example, `npm install --save express` becomes `npm install --save-exact express`. This will install the latest version of Express, just like always, but it won't mark it optimistically in your `package.json`—it'll specify an exact version.

This simple solution has a drawback: it doesn't pin down the version of sub-dependencies (the dependencies of your dependencies). Listing 12.3 shows the dependency tree of Express:

Listing 12.3 Express's dependency tree

```
your-express-app@0.0.0
└─┬ express@4.12.4
  ├─┬ accepts@1.2.9
  │ ├─┬ mime-types@2.1.1
  │ │ └── mime-db@1.13.0
  │ └── negotiator@0.5.3
  ├── content-disposition@0.5.0
  ├── content-type@1.0.1
  ├── cookie@0.1.2
  ├── cookie-signature@1.0.6
  ├─┬ debug@2.2.0
  │ └── ms@0.7.1
  ├── depd@1.0.1
  ├── escape-html@1.0.1
  ├─┬ etag@1.6.0
  │ └── crc@3.2.1
  ├── finalhandler@0.3.6
  ├── fresh@0.2.4
  ├── merge-descriptors@1.0.0
  ├── methods@1.1.1
  ├─┬ on-finished@2.2.1
  │ └── ee-first@1.1.0
  ├── parseurl@1.3.0
  ├── path-to-regexp@0.1.3
  ├─┬ proxy-addr@1.0.8
  │ ├── forwarded@0.1.0
  │ └── ipaddr.js@1.0.1
  ├── qs@2.4.2
  ├── range-parser@1.0.2
  ├─┬ send@0.12.3
  │ ├── destroy@1.0.3
  │ ├── mime@1.3.4
  │ └── ms@0.7.1
  ├── serve-static@1.9.3
  ├─┬ type-is@1.6.3
  │ ├── media-typer@0.3.0
  │ └─┬ mime-types@2.1.1
  │   └── mime-db@1.13.0
  ├── utils-merge@1.0.0
  └── vary@1.0.0
```

For example, I ran into a problem when trying to use the Backbone.js library. I wanted to pin to an exact version of Backbone, which was easy: I just specified the version. But in Backbone's `package.json`—which is out of my control!—it specified a version of Underscore.js that was optimistically versioned. That means that I could get a new version of Underscore if I reinstalled my packages, and more dangerously, I could get a new version of Underscore when deploying my code to the real world. For example, your dependency tree could look like this one day:

```
your-express-app@0.0.0
└─┬ backbone@1.2.3
  └── underscore@1.0.0
```

But if Underscore updated, it could look like this on another day:

```
your-express-app@0.0.0
└─┬ backbone@1.2.3
  └── underscore@1.1.0
```

Note the difference in Underscore's version here.

With this method, there is no way to ensure that the versions of your sub-dependencies (or sub-sub-dependencies, and so on) are pinned down. This might be okay, or it might not be. If it's not, you can use a nice feature of npm called "shrinkwrap".

12.3.2 The thorough way: npm's "shrinkwrap" command

The problem with the previous solution is that it doesn't lock down sub-dependency versions. npm has a subcommand called `shrinkwrap` that solves this problem.

Let's say you've run `npm install` and everything works just fine. You're at a state where you want to lock down your dependencies. At this point, run a single command from somewhere in your project:

```
npm shrinkwrap
```

You can run this in any Node project that has a `package.json` file and dependencies.

If all goes well, there will be a single line of output: "wrote npm-shrinkwrap.json". (If it failed, it's likely because you're executing this from a non-project directory or are missing a `package.json` file).

Take a look at this file. You'll see that it's got a list of dependencies, their versions, and then those dependencies' dependencies, and so on. Here's a snippet of a project that just has Express installed:

Listing 12.4 Snippet of an example npm-shrinkwrap.json file

```
{
  "dependencies": {
    "express": {
      "version": "4.12.4",
      // ...
      "dependencies": {
        "accepts": {
          "version": "1.2.2",
          // ...
          "dependencies": {
            "mime-types": {
              "version": "2.0.7",
              // ...
              "dependencies": {
                "mime-db": {
                  "version": "1.5.0",
                  // ...
                }
              }
            },
            "negotiator": {
              "version": "0.5.0",
              // ...
            }
          }
        },
        // ...
```

The main thing to notice is that the whole dependency tree is specified, not just the top layer like in `package.json`.

The next time you `npm install`, it won't look at the packages in `package.json`—it'll look at the files in `npm-shrinkwrap.json` and install from there. Every time `npm install` runs, it looks for the shrinkwrap file and tries to install from there. If you don't have one (as we haven't for the rest of this book), it'll look at `package.json`.

Like `package.json`, you typically check `npm-shrinkwrap.json` into version control. This allows all developers on the project to keep the same package versions, which is the whole point of shrinkwrapping!

UPGRADING DEPENDENCIES

This is all good once you've locked your dependencies in, but you probably don't want to freeze all of your dependencies forever! You might want to get bugfixes or patches or new features—you just want it to happen on your terms.

To update or add a dependency, you'll need to run `npm install` with a package name *and* a package version. For example, if you're updating Express from 4.12.0 to 4.12.1, you'll run `npm install express@4.12.1`. This will update the version in your `node_modules` folder, and you can start testing. Once it all looks good to you, you can run `npm shrinkwrap` again to lock in that dependency version.

Sometimes, shrinkwrapping isn't for you. You might want to get all of the latest and greatest features and patches without having to update manually. Sometimes, though, you want the security of having the same dependencies across all installations of your project.

12.4 Localized dependencies

Let's keep talking about dependencies, but with a different angle.

npm allows you to install packages globally on your system that execute as commands. There are a few "famous" ones, like Bower, Grunt, Mocha, and more. There's nothing wrong with doing this; there are a lot of tools that you need to install globally on your system. This means that, to run the Grunt command, you can type `grunt` from anywhere on your computer.

However, you can encounter some drawbacks when someone new comes into your project. For example, take Grunt. Two problems can occur when installing

Grunt globally:

1. The new developer doesn't have Grunt installed on their system at all. This means that you'll have to tell them to install it in your Readme or in some other documentation.
2. The second is related to the conversation in the previous section. What if they have Grunt installed but it's the wrong version? You could imagine them having a version of Grunt that's either too old or too new, which could lead to weird errors that might be tough to track down.

There's a pretty easy solution to these two problems: install it as a dependency of your project, not globally.

In Chapter 9, we used Mocha to use as a test framework. We could've installed this globally, but we didn't—we installed it locally to our project.

When you install Mocha, it installs the `mocha` executable command into `node_modules/.bin/mocha`. You can get at it in one of two ways: by executing it directly or by putting it inside an npm script.

INVOKING COMMANDS DIRECTLY

The simplest way is just to invoke these commands directly.

This is pretty darn easy, although it's a bit of typing: type the path to the command. If you're trying to run Mocha, just run `node_modules/.bin/mocha`. If you're trying to run Bower, just run `node_modules/.bin/bower`. (On Windows, running Mocha would be `node_modules\.bin\mocha`.)

There's not much to this conceptually!

EXECUTING COMMANDS FROM NPM SCRIPTS

The other way to do this is by adding the command as an npm script.

Once again, let's say that you want to run Mocha. Here's how you'd specify that as an npm script:

Listing 12.5 Specifying Mocha as an npm script

```
// …
"scripts": {
  "test": "mocha"
},
// …
```

When you type `npm test`, the `mocha` command is magically run. Let's resurface a diagram from Chapter 9 that explains how this works:

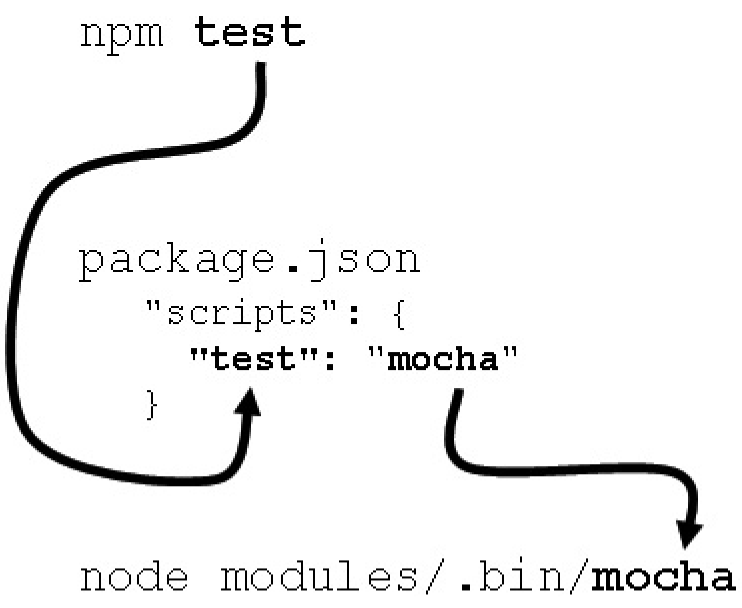


Figure 12-3 Typing npm test flows through a few steps before executing the command.

This is generally useful when you want to run the same kind of command over and over. It also keeps dependencies out of your global list!

12.5 Summary

In this final chapter, we:

- Learned about simplicity and the benefits of un-interleaving your code
- A convention to structure your app's files: a folder for routes, a folder for public files, a folder for views, a folder for common library-like functionality, and a folder for executables
- Using the `npm shrinkwrap` command to lock down dependency versions for reliability (and the benefits of doing so)
- How to avoid installing modules globally

That's the end of this chapter and this book! Go out and build cool things with Express.

A Other helpful modules

In this book, I covered a number of third-party Node modules, but there are loads of third-party modules that I couldn't get to. In this appendix, I'll give you a whirlwind tour of a bunch of modules I find useful in various situations. This list won't be thorough and is by no means exhaustive, but I hope it can help you find modules you'll like.

- Sequelize, an ORM for SQL. We discuss Mongoose in this book, whicl deals with the Mogo database; Sequelize is the Mongoose of SQL databases. It's an ORM, supports migrations, and interfaces with variou SQL databases. Check it out at http://sequelizejs.com/.
- Lodash is a utility library. You may have heard of the Underscore library, and Lodash is very similar. It boasts higher performance and a few extra features. You can read more at http://lodash.com/.
- Async is another utility library that makes it easier to handle various patterns in asynchronous programming. See more at https://github.com/caolan/async.
- Request is almost the opposite of Express. Where Express lets you accept incoming HTTP requests, Request lets you make outgoing HTTP requests It's got a simple API and you can find out more a https://www.npmjs.com/package/request.
- Gulp calls itself the "streaming build system". It's an alternative to tools like Grunt, and allows you to compile assets, minify code, run tests, and more. It uses Node's streams to increase performance. See http://gulpjs.com/ for more.
- node-canvas ports the HTML5 Canvas API to Node, allowing you to drav graphics on the server. You can see the documentation at https://github.com/Automattic/node-canvas.
- Sinon.js is useful in testing. Sometimes you want to test that a function is called, and a lot more. Sinon lets you make sure a function is called with specific arguments or a specific number of times. Check it out at http://sinonjs.org/.
- Zombie.js is a headless browser. There are other browser testing tools like Selenium and PhantomJS that spool up "real" browsers that you car

control. When you need 100% compatibility with browsers, they're a good call. But they can be kind of slow and unwieldy, which is where Zombie comes in. Zombie is a really quick headless browser that makes it easy to test your applications in a fake web browser. Its docs live at http://zombie.labnotes.org/.
- Supererror overrides console.error and makes it better, giving you line numbers, more info, and better formatting. Check it out at https://github.com/nebulade/supererror.

It's a short list, but I wanted to tell you that I love these modules! For more helpful Node resources and modules, you can check out:

- "Awesome Node.js" by Sindre Sorhus (at https://github.com/sindresorhus/awesome-nodejs)
- Eduardo Rolim's list of the same name (at https://github.com/vndmtrx/awesome-nodejs)
- Node Weekly (at http://nodeweekly.com/)
- Node Roundup section of DailyJS (at http://dailyjs.com/)