Functional

Design Patterns *for* Express.js

```
POST /books HTTP/1.1
Content-Type: application/json
Content-Length: 292

{
  "author": "Jonathan Lee Martin",
  "category": "learn-by-building",
  "language": "JavaScript"
}
```

A step-by-step guide to building elegant, *maintainable* node backends.

Functional Design Patterns for Express.js

A step-by-step guide to building elegant, maintainable Node.js backends.

By Jonathan Lee Martin

Jonathan Lee Martin
hello@jonathanleemartin.com
https://jonathanleemartin.com

Contents

Acknowledgments

> *Commit your works to the Lord*
> *And your plans will be established. (3)*
>
> *The mind of man plans his way,*
> *But the Lord directs his steps. (9)*
>
> — Proverbs 16:3, 9 (New American Standard Bible)

A few months ago, writing a book wasn't on the radar. Despite taking a travel sabbatical to invest in photography, I was mentally, physically and emotionally exhausted. I flew back to the States to figure out what was wrong.

I've always felt self sufficient, but for the first time in my adult life, I didn't have a plan for what was next. And that alone unnerved me — my goal-achieving nature *needed* to figure the next thing out. But after a season of being completely drained, there wasn't much to lose if I went back to basics and acknowledged Jesus in my planning. Instead of asking for His approval *after* I made my plans, I needed Him to *produce* the next step in me.

Dependence is frightening for a self-sufficient professional. I experienced what it's like to acknowledge that every breath and hint of energy, every ounce of motivation or creativity comes straight from Him. But through devotions and an outstanding local church, I learned to be wholly dependent on Him for the next *daily* step.

Jesus taught me what it means to *live* by faith: to so trust Him for tomorrow's step, that I stop making backup plans. To so rely on His promises, that I boldly ask Him to keep them.

These have been some of the best months of my life. Unplanned, miraculous in the day-to-day, outside my control and not on my bucket list. Everything I needed to get this book done fell right into place in spite of me. So it seems fitting to acknowledge Him, just as the composer Johann Sebastian Bach did:

Soli Deo gloria. To God alone be the glory!

Technical Reviewers

A technical book takes a special kind of reviewer, and I can't thank my technical reviewers enough — not just for their expertise, but for investing in me over the years. These

are the kindest and most brilliant developers I've had the privilege to call my friends and coworkers.

Jay Hayes and I worked at Big Nerd Ranch for five years as consultants and instructors. He has cheered me on and poured constant encouragement into me. His motivation, kindness and glowing regard have been the energy behind many of my projects, and I can safely attribute the reality of this book to his contagious curiosity.

Chris Aquino lined up my first opportunity to teach a web bootcamp six years ago. He has an inexplicable ability to believe in people, and whatever the hierarchical relationship — colleague, team manager, co-writer, co-instructor and friend — he has been believing in me, cheering me on and teaching me to be a better human. I wouldn't be doing what I love without his relentless positivity and kindness.

Joshua Martin has seen this book in many forms over the past year: bootcamps, video content, bullet points... it would be more accurate to call him my technical "pre"reviewer. He believes I can do anything, and in mentoring him as a developer, he has mentored me in being a better human being.

Josh Justice is arguably the kindest, punniest curmudgeon on Twitter. It's a rare polyglot who has an exceptional understanding of programming languages *and* the English language. When we worked together at Big Nerd Ranch, he was probably the only other developer with an opinion on serial commas and expertise in hyphenated words. Excellence, encouragement, razor-sharp feedback (look Josh, it's hyphenated!) — by the time Josh finished reviewing, I knew a stray "it's" would be more miraculous than getting `tar -czvf` right the first try.

Introduction

Learn the design patterns that transcend Express.js and recur throughout high-quality production codebases.

You've built backends in another language for a decade. You're a seasoned frontend JavaScript developer. You're a recent web bootcamp graduate. You're searching for an Express.js primer that isn't another screencast or exhaustive reference guide.

If any of those personas describe you, and you want to:

- **Learn the intuitions of developing elegant, maintainable backends.**
- **Learn without the distractions of every tangential tool in the ecosystem.**
- **Solidly grasp the motivation behind each concept as you build step-by-step.**
- **Expand your design palate with patterns that will transfer to other platforms.**

This book is for you. The pedagogical approach of this book is aimed at transferring design *intuitions* — motivated by real-world consulting experiences — in the fastest way possible. That translates to a razor-focused topic scope and no contrived examples to motivate tools you probably won't use, or shouldn't be using because they indicate deeper "code smells."

If you're looking for an exhaustive Express reference guide, prefer to read passively, or value books and video courses by their length, this book isn't for you — unless you're looking for a handsome adornment for your bookshelf!

Why Express?

Express is arguably the ubiquitous library for building Node backends. It is partly responsible for Node's surge in popularity, and many other Node frameworks build on top of Express. As of mid-2019, it is a dependency of 3.75 *million* codebases on Github alone. So if you hop into a Node codebase, chances are Express is part of it.

Express 5 is in development, but because a sizable group of tech giants depend on the API — directly or through a dependency — Express has essentially been on feature freeze for some time and is unlikely to see substantial overhauls.

This book steers away from version peculiarities and clever utility methods in favor of good design patterns. Thanks to these patterns, the backend we will build together has been rewritten in two other Node.js backend libraries with minimal changes.

Good design in an Express.js backend is good design anywhere. Some design patterns may be more idiomatic in one language than another, but the patterns you learn to develop Node backends will outlive Express and influence your design approaches in unrelated platforms.

Approach

There are countless books out there on backend design, so what makes this one different? In a word, the *approach*.

Many well-meaning books and courses are built on a more-is-better ethos: a single step-by-step course about Express is crammed with tangential topics like ES2015 JavaScript, databases and React. When the teaching approach and learning outcomes become secondary to the topic list, the result is a grab bag of goodies that *entertains* the developer rather than *educates*.

As a globetrotting educator, author and international speaker with a passion for craft, I've guided hundreds of developers — from career switchers to senior developers at Fortune 100 companies — through their journey into web development.

Both in the workplace and in the classroom, I've watched the entertainment model of learning cripple developers. So over the last six years of teaching one to sixteen week bootcamps, I've developed a pedagogical approach for developers at all skill levels.

Pedagogy — the method and practice of teaching — asks the essential question, *what does it mean to teach well*? My approach to vocational teaching is based on a few axioms:

- Teach and apply one concept at a time to minimize cognitive load.
- Focus on contextual learning.
- Leverage the ability to generalize concepts and apply in new contexts.
- Emphasize transmutable concepts.
- Dispel magic by building magic from scratch.
- Encourage fearless curiosity that dispels magic.
- Facilitate self-discovery, then follow with reinforcement.
- Engender love for the abstract from the concrete — not the reverse.
- Transfer intuition — not concepts — as quickly as possible.
- Quality is inversely proportional to length. Conciseness is kindness in practice.

Like a well-designed app, good pedagogy becomes a transparent part of the learning process by removing obstacles to learning — including itself!

Topics

This course focuses on best practice, conventional backend design for *pure* backend APIs. It is not exhaustive, comprehensive or targeted at advanced Express developers who are trying to scale huge legacy backends.

As we build a full-featured backend together, expect to work through:

- HTTP from scratch
- Request-response (life)cycle
- Express.js features that appear in high-quality codebases
- Testing backend routes with Insomnia
- Conventional headers for pure APIs
- Router design pattern
- Decoupling backend code
- Functional-style design patterns
- Currying and partially applied functions
- Dynamic segments
- Working with bodies
- Function objects
- Middleware
- Global vs. route middleware
- Middleware factories
- Common middleware libraries
- Authentication vs. authorization
- Password authentication
- Authentication with JSON Web Tokens
- Authorization design patterns

Because of this book's razor-focused approach, it intentionally omits:

- ES2015–ES2017 JavaScript
- RESTful conventions
- Databases
- Node essentials
- Frontend
- Cookies and sessions
- Passport.js
- Templating
- Niche Express methods, especially if they are symptomatic of design flaws.

Instead, it is this book's intention to equip developers — who already have a thorough applied knowledge of JavaScript, some light Node experience, and who have preferably built a backend before in any language or framework — with design insights.

Prerequisites

It is recommended that you have a strong foundation in JavaScript, preferably through hands-on product development. If your JavaScript experience is academic or limited to occasional hacking, the learning outcomes of this book may not be valuable.

Specifically, it is strongly recommended that:

- You have solid hands-on experience in JavaScript and Node.js.

- You are immensely comfortable with async programming in JavaScript with callbacks, async functions and Promises.
- You have ES2015 (previously called ES6) under your belt, especially destructuring syntax and arrow functions.
- You have an experiential understanding of HTTP, though a rigorous understanding is unnecessary.

Some things are *not* required to get the most out of this book! You don't need prior backend experience. If you understand how servers and clients interact, experience from either side of the equation is sufficient.

Let's Get Started

Throughout this book, we'll be building a full-featured Express backend together called **Pony Express**. Starting from an empty directory, we will intentionally bump into codebase growing pains to motivate functional design patterns and Express features.

But first, in the next chapter we'll detour from Node altogether and demystify the core abstraction of the web: **HTTP**.

Part I

Express Essentials

Chapter 1

How Servers Talk

You've probably built backends or frontends before, but what exactly happens when you load up www.google.com in the browser? When your React-powered frontend fires an AJAX request to the backend API, how does the backend handle those requests?

If you dialed up a server without a browser, would you know what to say?

Many seasoned web developers — whether backend or frontend — don't always have a concrete idea of what goes on between the frontend and backend, and that's okay! In fact, it's a testament to the web platform's exceptional choice of abstractions.

It also means any developer — newcomer or senior — can level up their web chops by dispelling the magic of **Hypertext Transfer Protocol**, better known as **HTTP**.

HTTP: The Core Abstraction of the Web

HTTP is the universal language of the web. That ubiquity means developers can produce groundbreaking web experiences without knowing the nuances of HTTP. So is it worth developing a fundamental understanding of HTTP?

HTTP is a great example of an **abstraction** for backend and frontend communication. Its flexibility embodies the decentralized, democratic spirit of the web and allows clients and backends with completely different capabilities to collaborate.

Good abstractions have a few characteristics:

- They expose a self-consistent API that culls infinite possibilities to a few domain-focused capabilities.
- The boundaries between API functions show **cohesion**, which means each function focuses on one responsibility.
- They cover up distracting differences in the underlying technology.
- They provide a common vocabulary, such as a **Domain Specific Language** (DSL).
- They are idiomatic to the intended programming language or platform.
- They tend to guide developers into good design patterns.
- They suggest idiomatic use cases and hint at underlying strengths and limitations.

The last two characteristics are some of the most compelling reasons to become intimately familiar with HTTP at a fundamental level: by stepping down to the level of an HTTP conversation, it's easy to determine which design approach will be idiomatic. And when things go wrong, you will know how to dig for the problem.

Most of this chapter won't be new to you, but just to be sure, let's pull back the covers on how frontends and backends communicate with HTTP. If you've worked on frontends and backends for a while, HTTP may actually be *simpler* than you thought.

Installing `telnet`

Instead of using a browser to view a website, let's drop down to the HTTP level and have our own text conversation with http://expressjs.com using the `telnet` TCP client.

The **`telnet`** client comes with many Linux-based operating systems, so check if you already have it with the `which` command:

```
$ which telnet
/usr/local/bin/telnet
```

The **`which`** command shows where a command is installed. It's okay if a different path is printed — such as `/usr/bin/telnet` — but if nothing is printed out, `telnet` is not installed.

On Linux

Most Linux-based operating systems already include the `telnet` client, but if `which telnet` doesn't print anything, you can install it with the OS's bundled package manager:

```
# On Ubuntu:
$ apt-get install telnet

# On Fedora:
$ yum install telnet
```

On macOS

`telnet` is a common omission on macOS, but easy to install through the Homebrew package manager. You've probably already installed Homebrew:

```
$ which brew
/usr/local/bin/brew
```

If nothing shows up, go to https://brew.sh and follow the famous one-line installation instructions. Installing `telnet` — and many other developer staples from the Linux world — is easy with Homebrew:

```
$ brew install telnet
```

Restart your terminal and run `which telnet` again. If a path prints out, you're all set!

An HTTP Conversation with `telnet`

Let's start our own HTTP conversation using `telnet`. Type the following in your terminal:

```
$ telnet expressjs.com 80
```

We just opened a raw TCP connection to http://expressjs.com. By default, web servers listen on port 80 for unencrypted HTTP connections. Once the connection is open, we can have a simple two-way text conversation with the server by following the HTTP protocol.

Like your browser, the `telnet` command is a **client** or **user agent**: it initiates the TCP connection and makes the requests.

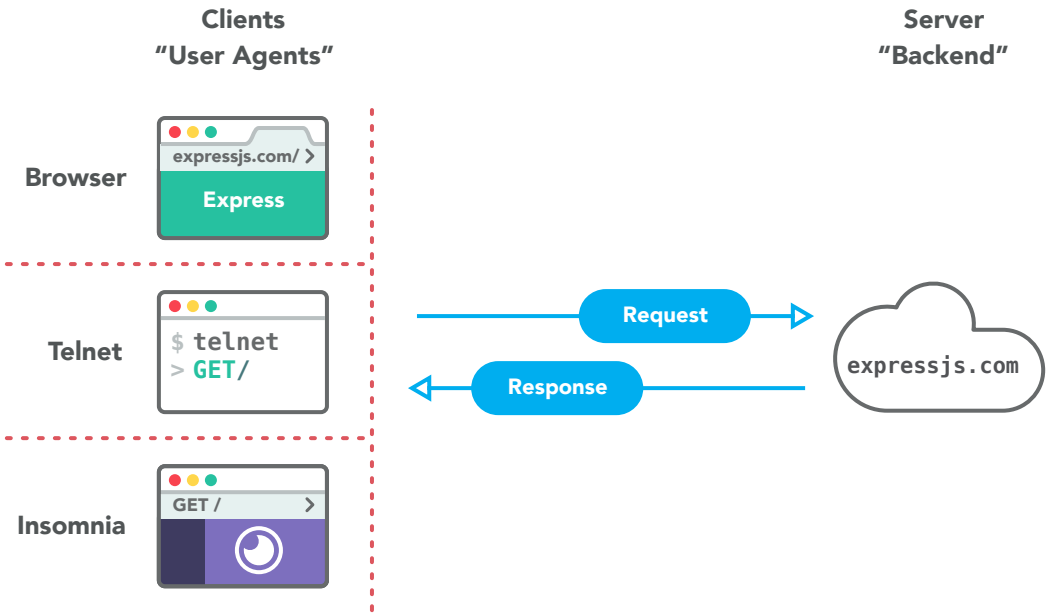


Figure 1.1: *Clients initiate the request, servers respond.*

To request the homepage like a browser would, we need to specify the **path**, which is everything that comes after the domain, `expressjs.com`. The path for the homepage is `/`.

The connection probably timed out while you were reading, so don't forget to rerun the `telnet` command. Next, type this request in your `telnet` session and hit the return key:

```
GET / HTTP/1.1
```

There are three parts to this request:

1. The **HTTP method** — also called the **HTTP verb** — such as `GET`, `POST` or `DELETE`.
2. The **resource path**, such as `/`, `/homepage.html` or `/posts/1.json`.
3. The **HTTP version**. If this is omitted, it is assumed to be the good ol' 1991 version of HTTP, `HTTP/0.9`. When we test our own backend in the next chapter, you can occasionally omit the HTTP version, but most servers only support `HTTP/1.1` and will return `400 Bad Request` if omitted.

That's enough for a valid request, but because the same IP address commonly hosts many domains, we need to include one **request header**, `Host`, so the server knows for certain which domain we are trying to connect to.

Assuming your `telnet` session hasn't timed out, type this header immediately after the request line you just typed in:

```
Host: expressjs.com
```

Your final request should look like this:

```
GET / HTTP/1.1
Host: expressjs.com
```

Hit the return key a couple times. The server waits for an empty line to indicate the request is finished, then it responds. Here's an abridged example of what the response might look like:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Connection: keep-alive
Vary: Accept-Encoding
Cache-Control: max-age=600

<!DOCTYPE html>
<html lang="en">
<head>
<title>Express - Node.js web application framework</title>
[···]
```

The server's response mirrors the structure of the request:

1. The first line details the HTTP version and **response status code**. `200 OK` means the `expressjs.com` server understood our request.
2. The **response headers** come between the first line and empty line. This is probably where you've spent a good deal of time debugging your own backends and frontends!
3. The **response body** is everything that comes after the blank line. Since we asked for the homepage, the body is an HTML document.

That's it! The complex behavior of visiting a website or making an AJAX request to a backend API is actually a human-readable conversation that starts with one request and ends with one response. This brief conversation is called the **request-response cycle**.

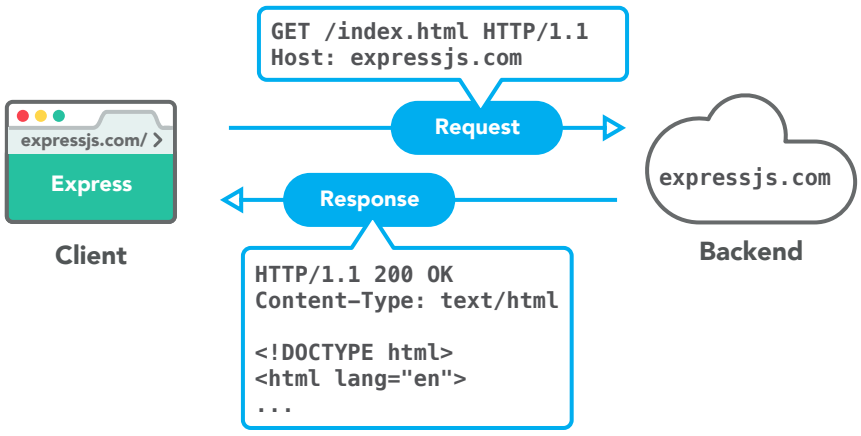


Figure 1.2: *One request-response cycle with `expressjs.com`.*

The first version of HTTP (0.9) immediately closes the connection after one request-response cycle. Later versions of HTTP keep the connection open by default for better performance, so it feels more like chatting with a chatbot.

When you're done talking to the server and reminiscing about the web's old-fashioned origins, tap `Ctrl-C` to close the connection.

Talking to a Backend API

Talking to a backend API is no different from talking to a webpage server: the main difference is the contents of the response body. Websites like http://expressjs.com are HTTP servers that respond with HTML documents as the body. Backend APIs are HTTP servers that typically respond with JSON-formatted strings.

Our server will be a *pure* backend API, so it will reply with JSON-formatted strings. Let's get a feel for what a backend API looks like from `telnet`:

```
$ telnet jsonplaceholder.typicode.com 80
```

JSONPlaceholder is a fake JSON backend API with endpoints and conventions similar to many backends. Let's look up a post:

```
GET /posts/1 HTTP/1.1
Host: jsonplaceholder.typicode.com
```

Don't forget to hit return twice. The response should look like this:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 292
Connection: keep-alive
Vary: Origin, Accept-Encoding
Cache-Control: public, max-age=14400

{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere...",
  "body": "quia et suscipit\nsuscipit..."
}
```

Making Requests with Insomnia

As HTTP clients go, `telnet` and the browser are at either extreme: `telnet` is a bit tedious for testing a backend, while the browser is intended for rich web experiences. As we build our backend, we will use the Insomnia HTTP client as a happy middle ground.

Go to https://insomnia.rest to download the Insomnia desktop app.

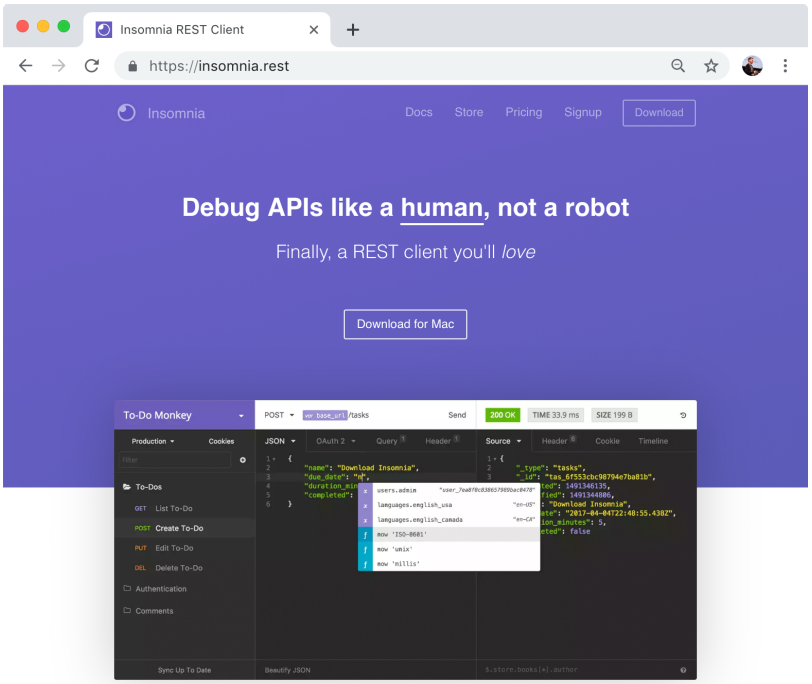


Figure 1.3: *Insomina, a slick HTTP client for testing backends.*

Insomnia is a purpose-built GUI for testing backend APIs. It saves requests as you create them, so it's dead simple to resend them as your backend evolves. Let's revisit our first HTTP request to the Express homepage. Click "New Request" and name it "Express Homepage", then change the method to "GET" and click "Create".

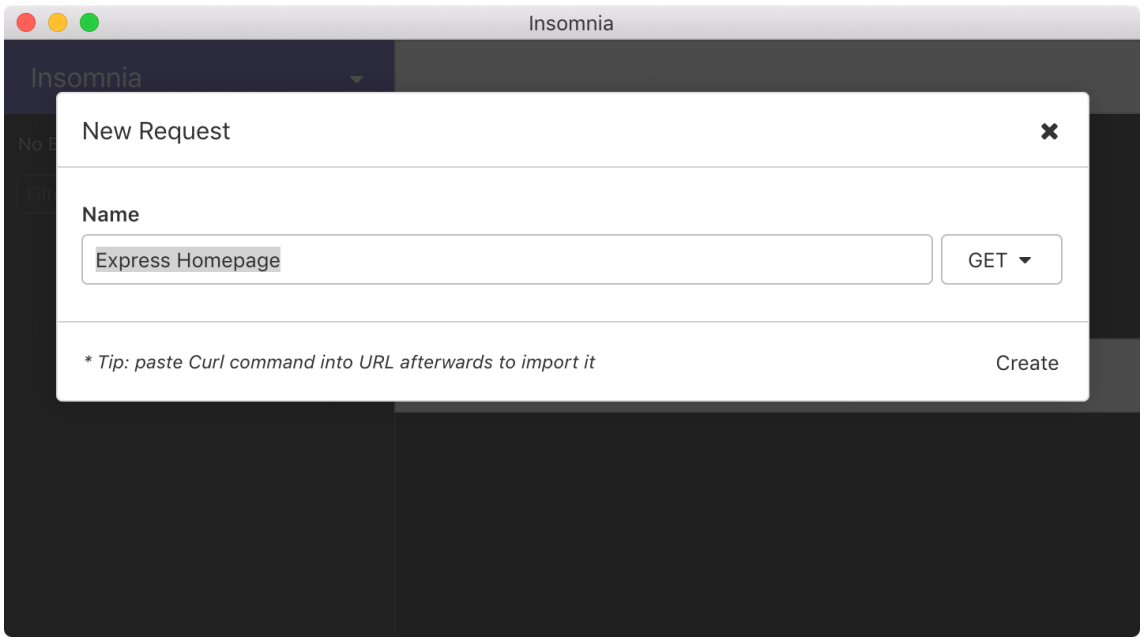


Figure 1.4: *Creating an Insomnia request.*

Insomnia is split into two panels. The left panel is where we can customize anything about the request: request headers, request body, HTTP method and the URL. Change the URL to http://expressjs.com/ and click "Send".

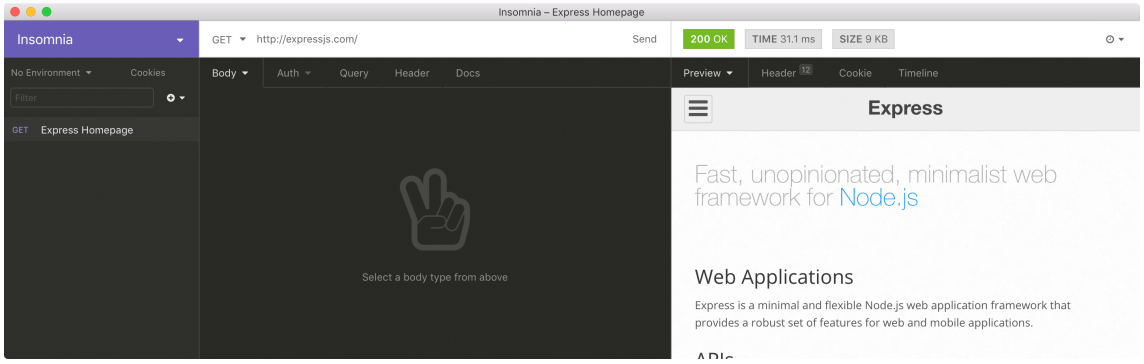


Figure 1.5: *An Insomnia request for the Express homepage.*

The right panel shows everything related to the server's response. By default, it shows a nice preview of the webpage. Select "Raw Data" from the "Preview" drop-down menu — that's exactly what we saw in our `telnet` session.

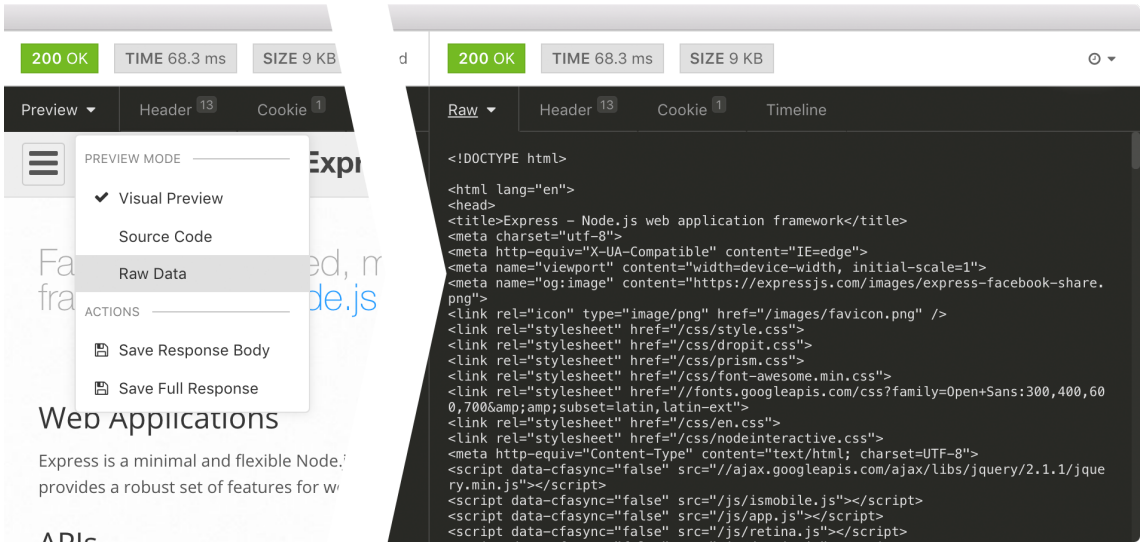


Figure 1.6: *After changing the Preview to Raw Data, the response looks like it did in* `telnet`.

The response headers under "Header" should also look the same as our `telnet` session. The response body was HTML code, so the response also includes a `Content-Type: text/html` header.

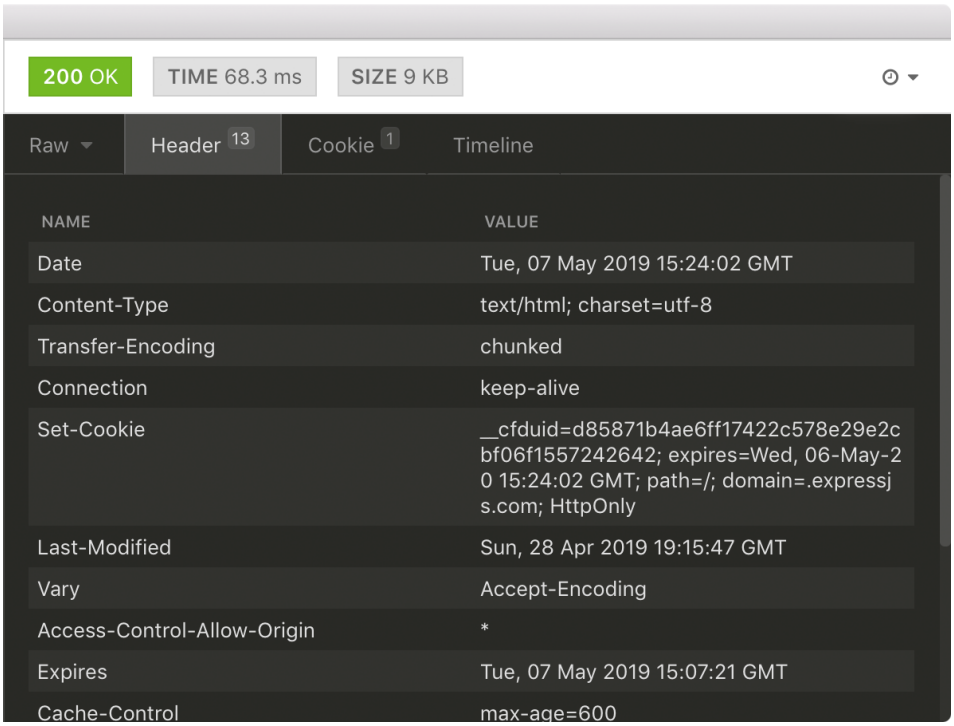


Figure 1.7: *Response headers for the Express homepage.*

You can see the entire text conversation between Insomnia and http://expressjs.com/ from the "Timeline" tab:

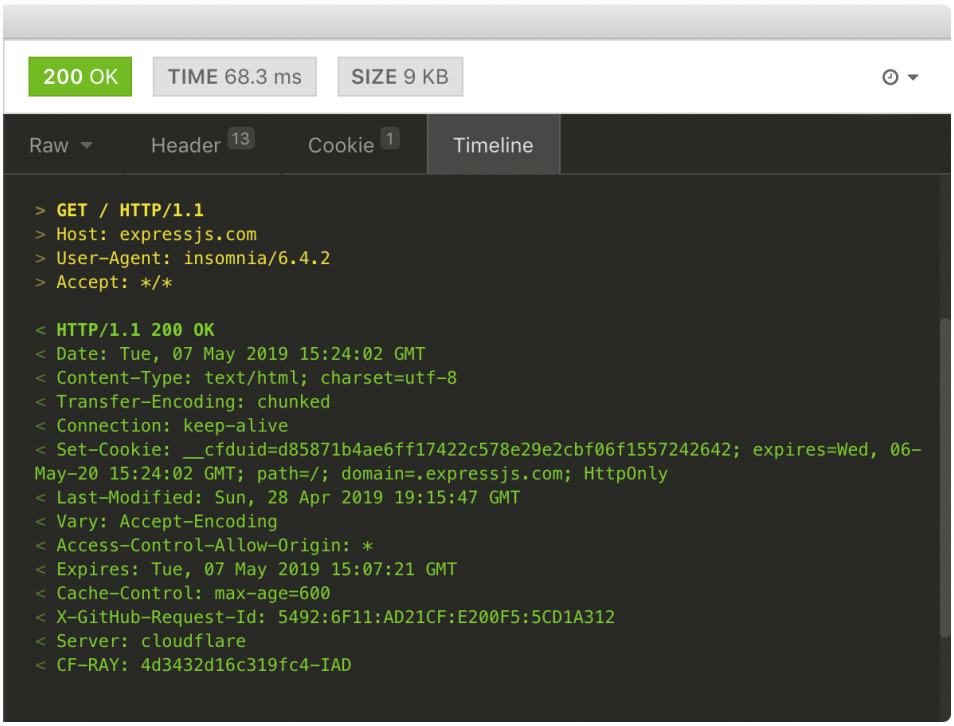


Figure 1.8: *No magic here, Insomnia is really just texting,* `telnet` *style.*

Let's repeat that for a JSON backend API. Create a new Insomnia request called "View Post", set the request URL to http://jsonplaceholder.typicode.com/posts/1 and click "Send":

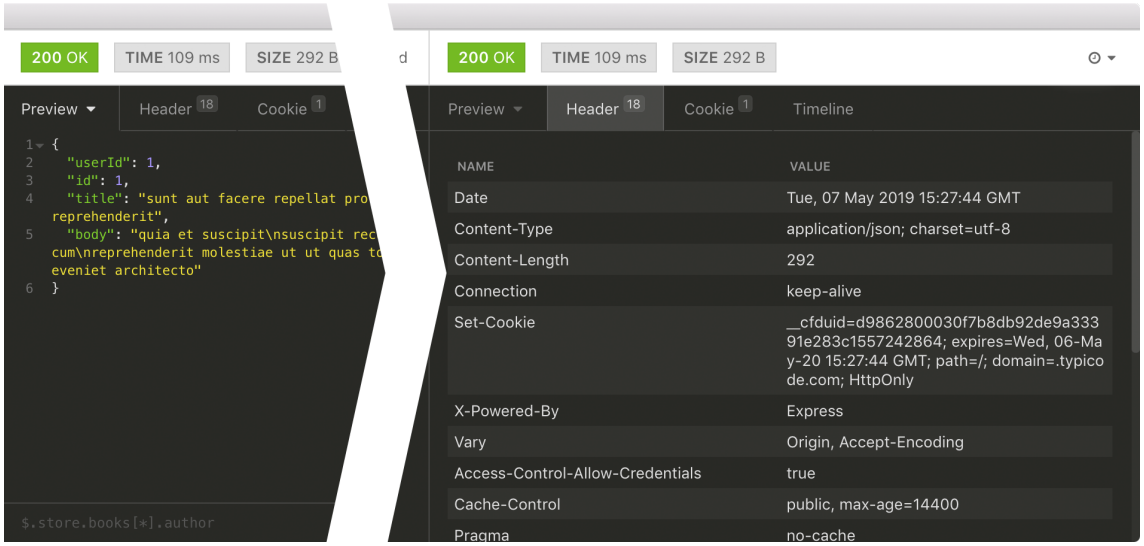


Figure 1.9: *Backends usually respond with JSON and include* `Content-Type: application/json`*.*

The response from JSONPlaceholder is still text, but it's formatted as JSON and includes a `Content-Type: application/json` header. That's pretty much the only difference between a webpage server and a backend API: the response body and content type.

Go Further

If you want to continue experimenting with `telnet` and Insomnia, there is an expansive compilation of public backend APIs at https://github.com/toddmotto/public-apis that includes everything from weather APIs to bacon ipsum APIs.

In the next chapter, we'll begin building our own backend API, **Pony Express**.

Chapter 2

Responding to Requests

The best way to motivate the design challenges of an Express backend is to build one! We'll be developing **Pony Express**, a simple mail server that stores emails and users. The Pony Express backend API won't speak SMTP or IMAP like a traditional mail server, but will behave more like a conventional JSON backend API.

Hop to a terminal and create a new directory for the project:

```
$ mkdir pony-express
$ cd pony-express
```

Make sure you are running Node 8 or greater:

```
$ node --version
v10.15.3
```

Our Node project needs a `package.json` file. Make sure you've switched into the project directory, then initialize a `package.json` file with default options:

```
$ npm init -y
```

Open your project folder in your preferred text editor. If you are using Visual Studio Code, you can open an editor from the terminal with the `code` command:

```
$ code ./
```

Simple Servers with the `http` Module

Let's start off Pony Express with the built-in `http` module and respond to any request by echoing the request method and URL. Create a new file, `index.js`:

index.js

```
+ const http = require('http');
+
+ let server = http.createServer((req, res) => {
+   let route = req.method + ' ' + req.url;
+   res.end('You asked for ' + route);
+ });
+
+ server.listen(3000);
```

Hop back to the terminal and boot the server up:

```
$ node index.js
```

Keep in mind that we don't have any `console.log()` statements, so you won't see anything printed in the terminal. So long as it hangs and doesn't print out a stack trace, it's running!

When you open http://localhost:3000/emails in your browser, you should see "You asked for GET /emails" printed. That's it! We've built a server with Node's `http` module.

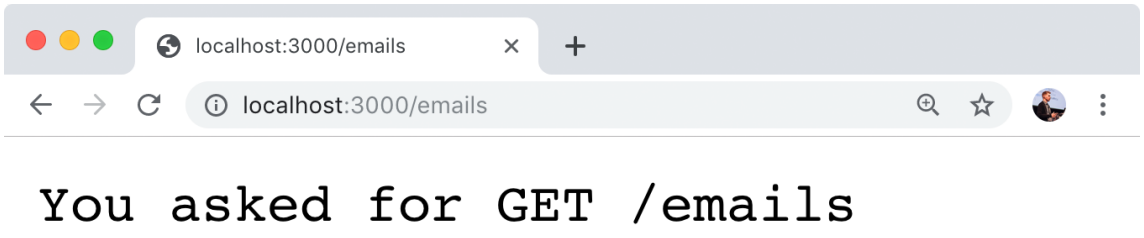


Figure 2.1: *A humble backend response, as seen by the browser.*

Together, the request method `GET` and request URL `/emails` are called a **route**. Although production backends may respond to hundreds or thousands of unique routes,

all that logic lives in one callback: the argument to `http.createServer()`. That callback — called the **request handler** or **request listener** — is where the entire brains of a backend server lives. When an HTTP request comes in, `http.createServer()` runs this callback. So whether fifty browsers connect to http://localhost:3000/emails or one browser opens fifty tabs to this URL, the request handler callback will execute fifty times.

Notice that a request handler receives two arguments, both objects: `req` and `res`, which are short for "request" and "response". The request object contains all the details about the request the server received, so most of the time you should treat `req` as read-only. Here's an abridged example of what a request object looks like:

http.IncomingMessage

```
{
  headers: { accept: "application/json" },
  httpVersion: "1.1",
  method: "GET",
  trailers: {},
  upgrade: false,
  url: "/emails",
}
```

The response object is a bit more confusing: even though it is an argument to our request handler, it doesn't really have any information in it. Instead, it's a blank box for the request handler to fill — through mutation or the attached methods — and `http.createServer()` will formulate the real HTTP response to the client that made the request. So while `req` is read-only, `res` is write-only.

Because the names are so similar, it's easy to mix them up and accidentally write to the request object or read headers from the response object. If you run into strange errors while following step-by-step, double check for `req` vs. `res` mixups!

Speaking HTTP over Telnet

Eventually Pony Express will receive requests from fancy clients like a frontend web app. But to start, let's talk to it with the `telnet` client so we can build a concrete mental model. You'll want to leave your server running, so open a new terminal window for your `telnet` session and make a `GET /emails` request:

```
$ telnet localhost 3000
> GET /emails HTTP/1.1
>
```

Don't forget to tap the return key twice. The server response should look like this:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 19

You asked for GET /emails
```

Except for the first line, the server response is structured exactly like the request. The `res.end()` method added a response body after the headers and automatically inserted a `Content-Length` header.

Responding to Different Routes

Our backend isn't very useful, how can we make it send different responses for different routes? We can start with the simplest solution: add an `if...else` statement.

Pony Express is a mail server, so there should be routes to list users and emails. Just so we don't get bogged down in database stuff, download some sample emails and users as JSON from learn.nybblr.com/express/fixtures.zip. Unzip the download and move the entire `fixtures/` directory into your project directory so it looks like this:

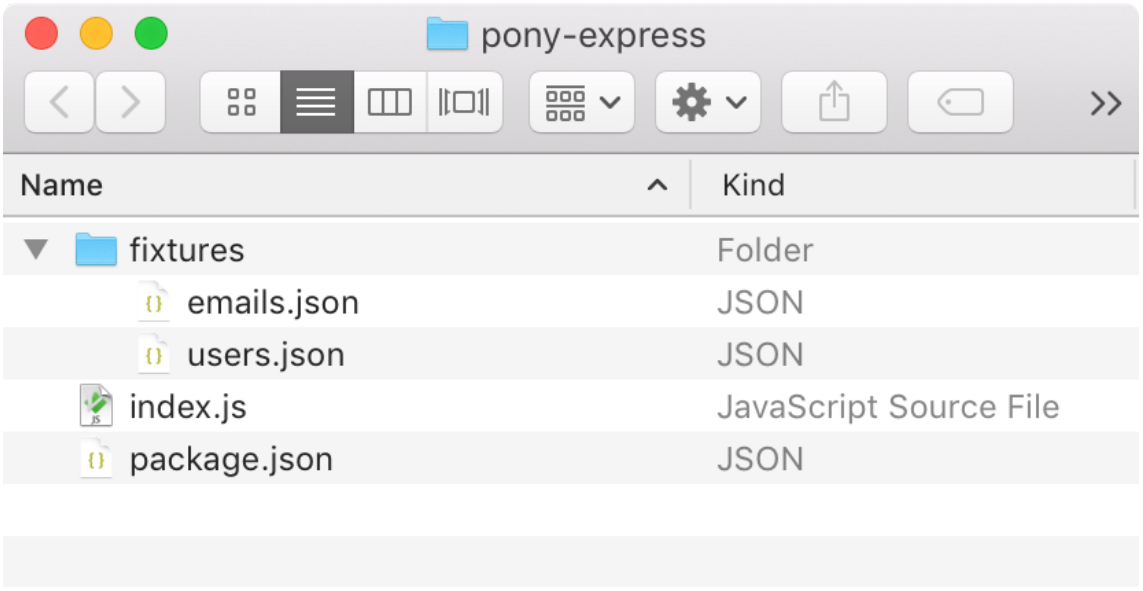


Figure 2.2: *Directory structure of Pony Express.*

When a client makes a request for users or emails, let's respond with the respective array as JSON:

index.js

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

+ const users = require('./fixtures/users');
+ const emails = require('./fixtures/emails');

  let server = http.createServer((req, res) => {
    let route = req.method + ' ' + req.url;

+   if (route === 'GET /users') {
+     res.end(JSON.stringify(users));
+   } else if (route === 'GET /emails') {
+     res.end(JSON.stringify(emails));
+   } else {
      res.end('You asked for ' + route);
+   }
  });

  server.listen(3000);
```

Don't forget to restart the server after you finish your changes.

From now on, we'll make requests through Insomnia and only whip out `telnet` occasionally to dispel magic. In Insomnia, create a new request for `GET /users` and `GET /emails`, then try sending both.

Insomnia shows all the response details in a nice UI on the right, but you can always see the raw text conversation by switching to the Timeline tab. Looks pretty similar to `telnet`!

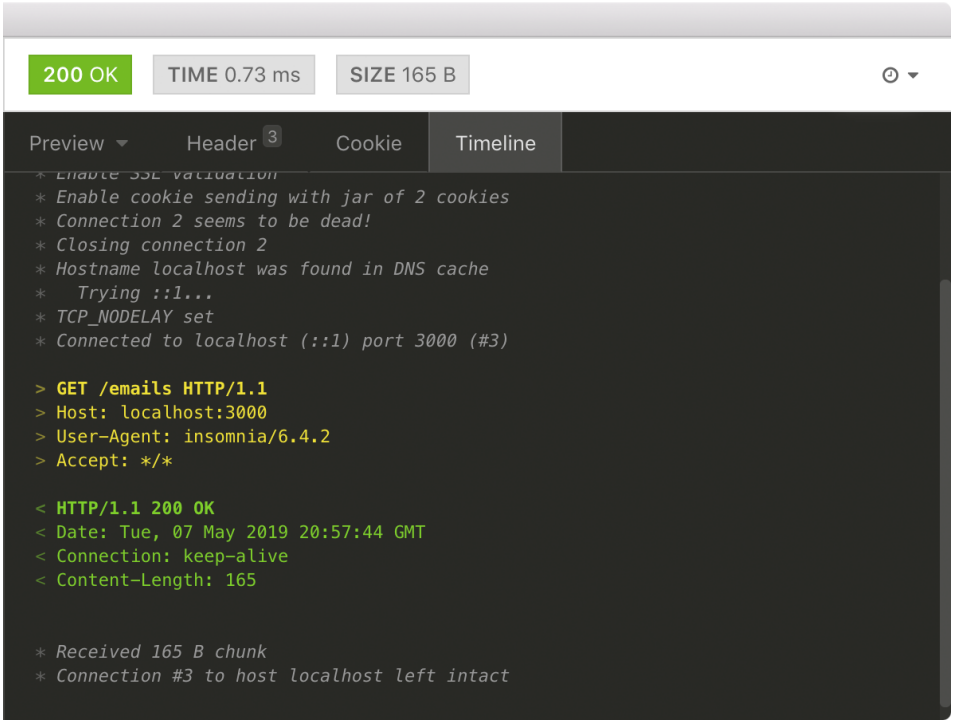


Figure 2.3: *Insomnia's timeline tab shows the raw HTTP conversation.*

Notice the response headers. It's pretty empty right now, but since our server responds with a JSON-formatted body, it's supposed to include a `Content-Type: application/json` header in the response. That doesn't come for free, but we'll fix it later.

Restarting our backend after every change is getting old, let's install `nodemon`, a drop-in replacement for the `node` command that automatically restarts the server when anything in the directory changes:

```
$ npm install --save-dev nodemon
$ npx nodemon index.js
```

What's with `npx`? The `npx` command allows us to run terminal commands that come bundled with a Node module, but aren't installed globally. If you happened to install `nodemon` with `npm install -g nodemon`, you could drop the `npx` part, but it's better practice to bundle developer tools with the project by listing them in the `devDependencies` section of `package.json`.

From now on, we'll boot up the server using `npx nodemon index.js`.

Hello, Express

The entire brains of your backend lives in that callback function to `http.createServer()`, and it will become massive if we don't break it up.

That's where Express comes in: **Express** is a lightweight library for architecting the request handler callback. Let's install Express:

```
$ npm install express
```

Express isn't designed to replace `http.createServer()`, but to build on top of it with a few powerful design patterns. That means almost nothing will change when we "switch" to Express! Let's wire it up in `index.js`:

index.js

```
  const http = require('http');
+ const express = require('express');

  const users = require('./fixtures/users');
  const emails = require('./fixtures/emails');

+ let app = express();

- let server = http.createServer((req, res) => {
+ app.use((req, res) => {
    [···]
  });

+ let server = http.createServer(app);

  server.listen(3000);
```

Save your changes, but don't restart the server — remember, `nodemon` will do that automatically. Test your Insomnia requests again to make sure everything works as before.

The `express()` function is a factory for building the request handler, but we still use Node's built-in `http` module to listen for incoming HTTP requests. Express is simply in the business of helping you build a massive callback. That's a significant contribution since that's where the entire brains of the backend lives.

Express Shorthands

That probably isn't the syntax you've seen for booting up an Express backend. Because Express apps are always used with Node's `http` module, a popular shorthand is provided to boot up an Express app:

index.js

```
- const http = require('http');
  const express = require('express');

  [···]

- let server = http.createServer(app);

- server.listen(3000);
+ app.listen(3000);
```

The `app.listen()` method does exactly what we had before. Most developers stick with this shorthand, but it helps to break it out into what's actually going on. It also demonstrates one reason Express is so popular: it doesn't replace or even augment the `http` module, it just helps you build the request handler.

However, Express does add some useful methods to the response object. Let's use `res.send()` to automatically convert the list of emails and users into JSON-formatted strings:

index.js

```
    [···]

    if (route === 'GET /users') {
-     res.end(JSON.stringify(users));
+     res.send(users);
    } else if (route === 'GET /emails') {
-     res.end(JSON.stringify(emails));
+     res.send(emails);
    } else {
      res.end('You asked for ' + route);
    }

    [···]
```

Check the route with Insomnia: the response headers have a few additions!

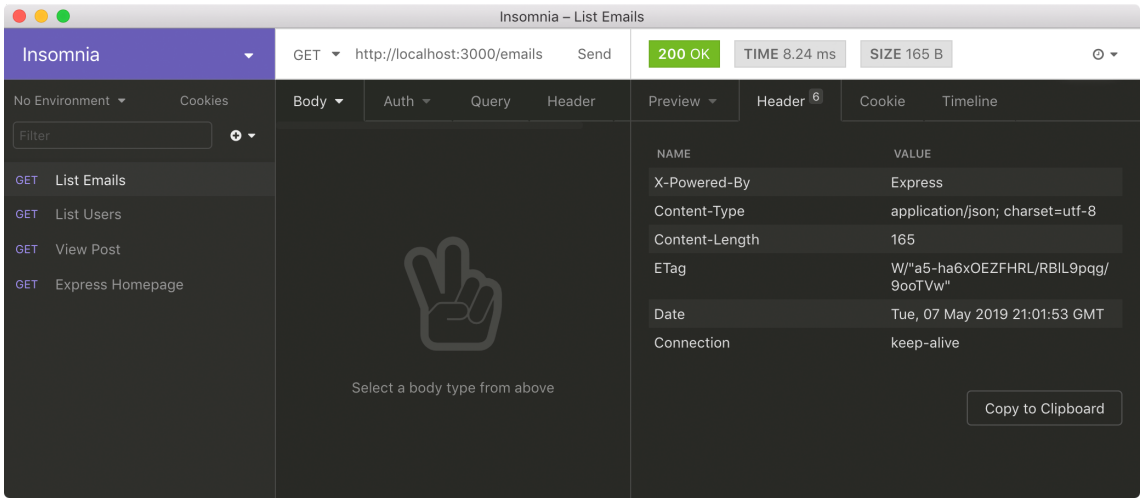


Figure 2.4: *Express adds a few headers to the response automatically.*

It added a `Content-Type` header! Express helper methods do a lot of nice things like that. While the vanilla `.end()` method is Node's generic stream method and just sends strings as is, the `.send()` method typically responds with JSON and adds response headers such as `application/json`. Since Pony Express will mostly respond with JSON-formatted bodies, we'll almost always use `.send()` in place of `.end()`.

The Pony Express is off to humble beginnings. We aren't really leveraging any of the features Express offers, but perhaps Express's best feature is that it doesn't replace the `http` module: in fact, it builds on it. That alone makes Express an exceptional foundation for building complex backends, because as the requirements expand we'll be limited by Node's fantastically robust `http` module, and *not* by Express!

Go Further

There are many nifty features and tips we don't have time to cover throughout the book — that's where you come in! Some chapters include challenges for you, the reader, to try on your own. It's easy to turn into a zombie typist in a step-by-step guide, so these exercises are an invitation to leave your undead form behind and level up!

Multiple Response Types

Right now, the `GET /users` and `GET /emails` routes always respond with JSON, but clients can specify a preferred response `Content-Type` by including an `Accept` header in the request:

```
GET /emails HTTP/1.1
Accept: text/csv
```

In addition to JSON, add support for a CSV or XML response based on the `Accept` request header. Insomnia sets the `Accept` header to `*/*` by default, but you can override that by adding an `Accept` header to the request:

- For JSON, use `Accept: application/json`
- For CSV, use `Accept: text/csv`
- For XML, use `Accept: application/xml`

If the `Accept` header is not specified, default to a JSON response. A few tips on this challenge:

- Don't manually parse the `Accept` header — it's fairly complex. Instead, check out Express's `req.accepts()` method. Also, don't manually set the `Content-Type` header on the response; instead, use Express's `res.type()` method.
- There are plenty of modules to generate CSV. Check out the `stringify()` method of the aptly named `csv` node module.
- To generate XML, check out the `xmlbuilder` module.

Chapter 3

Express Router

Backend APIs often respond to hundreds or thousands of unique method and path combinations. Each method and path combination — such as `GET /users` or `POST /emails` — is called a **route**. But no matter how many routes your backend API supports, every single request will need to be processed by a single request handler function. That means `index.js` will grow with every new route: even if each route took only one line of code, that's a large file and a nightmarish recipe for merge conflicts.

How can we architect the request handler callback such that, for every new route, the number of files grows while the average file length stays the same? Put another way, how do we design a backend so the codebase scales horizontally instead of vertically?

Refactoring with the Router Pattern

The easiest way to accomplish this is by applying the **Router design pattern**, not to be confused with Express's Router API. The Router design pattern is a common refactor to obliterate ballooning `switch` statements or `if...else` statements that share similar predicates.

There are a few steps to apply this design pattern:

1. Extract the body of each case into a function.
2. Replace the body of each case with an invocation of that function.
3. Create a map from each predicate condition to its corresponding function.
4. Replace the `switch` or `if...else` statement with one function lookup and invocation.

One of the strengths of this refactor is that, at each step in the refactor, the code should still run so you can catch bugs early on. Try not to skip ahead, but take the refactor one step at a time.

In the request handler of `index.js`, extract the body of each case into a function:

index.js

```
[···]

let app = express();

+ let getUsersRoute = (req, res) => {
+   res.send(users);
+ };
+
+ let getEmailsRoute = (req, res) => {
+   res.send(emails);
+ };

app.use((req, res) => {
  [···]
```

The second step is to replace the body of each case with its function. If your functions were invoked with slightly different arguments, you'd need to do a little extra refactoring. Since both routes have the same function signature, we can continue with the refactor:

index.js

```
[···]

app.use((req, res) => {
  let route = req.method + ' ' + req.url;

  if (route === 'GET /users') {
-   res.send(users);
+   getUsersRoute(req, res);
  } else if (route === 'GET /emails') {
-   res.send(emails);
+   getEmailsRoute(req, res);
  } else {
    res.end('You asked for ' + route);
  }
});

[···]
```

Our code should still work after each step in the refactor, so give your `GET /users` and `GET /emails` routes a quick test with Insomnia.

The third step is to create some sort of mapping from the predicate condition to a corresponding route. Since the `if...else` conditions are always a comparison with a string like `"GET /emails"`, we can use a plain ol' JavaScript object:

index.js

```
  [···]

  let getUsersRoute = (req, res) => {
    res.send(users);
  };

  let getEmailsRoute = (req, res) => {
    res.send(emails);
  };

+ let routes = {
+   'GET /users': getUsersRoute,
+   'GET /emails': getEmailsRoute,
+ };

  app.use((req, res) => {
    [···]
```

The fourth and final step is to replace the `if...else` cases with a single lookup in the list of routes:

index.js

```
  [···]

  app.use((req, res) => {
    let route = req.method + ' ' + req.url;
+   let handler = routes[route];

-   if (route === 'GET /users') {
-     getUsersRoute(req, res);
-   } else if (route === 'GET /emails') {
-     getEmailsRoute(req, res);
+   if (handler) {
+     handler(req, res);
    } else {
      res.end('You asked for ' + route);
    }
  });

  [···]
```

What about that last `else` statement? We still need a fallback to catch any unknown routes like `GET /spam`, but you could extract the logic into a separate function like `noRouteFound()` to remove the `if...else` statement altogether:

index.js

```
[···]

+ let noRouteFound = (req, res) => {
+   let route = req.method + ' ' + req.url;
+   res.end('You asked for ' + route);
+ };

  app.use((req, res) => {
    let route = req.method + ' ' + req.url;
-   let handler = routes[route];
+   let handler = routes[route] || noRouteFound;

-   if (handler) {
    handler(req, res);
-   } else {
-     res.end('You asked for ' + route);
-   }
  });

[···]
```

Send a few requests with Insomnia to make sure the routes still work. Huzzah! We eliminated a growing `if...else` statement, and in the process extracted individual routes outside the request handler.

Express Router

Now that we've applied the Router design pattern, which part is the "router"? In this context, a **Router** is a function whose only responsibility is to delegate logic to another function. So the entire callback to `app.use()` is a Router function!

Let's make this a bit more obvious by assigning the request handler callback to a variable before passing it to `app.use()`:

index.js

```
  [···]

- app.use((req, res) => {
+ let router = (req, res) => {
    let route = req.method + ' ' + req.url;
    let handler = routes[route] || noRouteFound;

    handler(req, res);
- });
+ };

+ app.use(router);

  app.listen(3000);
```

While route functions are unique to the particular backend you're building, the router function we extracted is common to all backends. Well, that just happens to be Express's flagship feature: `express.Router()` generates a Router function much like the one we just wrote. Let's swap it in!

index.js

```
  [···]

- let routes = {
-   'GET /users': getUsersRoute,
-   'GET /emails': getEmailsRoute,
- };

- let noRouteFound = (req, res) => { ... };

- let router = (req, res) => { ... };
+ let router = express.Router();

+ router.get('/users', getUsersRoute);
+ router.get('/emails', getEmailsRoute);

  app.use(router);

  [···]
```

Whoa, look at that! This behaves identically to what we had before, but it's much terser. Express's Router provides an *expressive* API for creating a route map with methods like `router.get()`.

Test out your routes with Insomnia once more — despite the mass deletions, the back-end should respond identically to before.

Functions with Methods

But wait, the `router()` generated by `express.Router()` is a function, just like the handwritten `router()` it replaces. If `router()` is a function, why does it have methods like `router.get()`?

This is a recurring API design style for JavaScript libraries, and especially Express. In fact, we already saw that `express()` returns a function we called `app()`, yet `app` has a `.use()` method. Here's a shortened example:

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

let app = express();

// `app` is definitely an object
// because it has methods like `.use()`:
app.use((req, res) => {
  res.send('Hello');
});

// `app()` is definitely a function too
// because it can be invoked. These are the same:
let server = http.createServer(app);
let server = http.createServer(
  (req, res) => app(req, res)
);

server.listen(3000);
```

In JavaScript, functions are also objects: that means they can be invoked, but also have methods. Unsurprisingly, JavaScript functions are called **function objects**. In Express, this duality makes it easy to seamlessly combine vanilla functions with libraries that include an elegant configuration API.

Routes with Dynamic Segments

Our server is made of many small functions, so it should be trivial to tease apart the codebase as it grows. But before we test that theory out, let's add a couple more routes.

If an API client needs to look up a particular user, the conventional route would be `GET /users/<id>`. For example, `GET /users/1` should respond with the JSON-formatted data for the user with ID #1. Express's Router makes it easy to match wildcards like this with **Dynamic Segments**:

index.js

```
  [...]

  let getUsersRoute = (req, res) => {
    res.send(users);
  };

+ let getUserRoute = (req, res) => {
+   console.log(req.params);
+ };

  [...]

  router.get('/users', getUsersRoute);
+ router.get('/users/:id', getUserRoute);

  [...]
```

Dynamic segments are denoted by a colon and short name, such as `:id`. When a request for `GET /users/1` comes in, it will match the route for `/users/:id`, and the dynamic segment `:id` will match `1`. The `getUserRoute()` function can retrieve that value in `req.params.id`.

Create a new request in Insomnia for `GET /users/1`, then fire it off. The request will hang, but in the terminal you should see an object with one key-value pair logged:

```
{ id: '1' }
```

A route can be defined with more than one dynamic segment. For example, to add a route that lists all emails from user #1 to user #2, you could add a route like this:

```
// Given a request like:
// GET /emails/from/1/to/2
// Then req.params will be { sender: '1', recipient: '2' }
router.get('/emails/from/:sender/to/:recipient',
  (req, res) => {
    console.log(req.params);
  };
);
```

Let's finish up the `getUserRoute()` in `index.js`:

index.js

```
[···]

let getUserRoute = (req, res) => {
-   console.log(req.params);

+   let user = users.find(user => user.id === req.params.id);
+   res.send(user);
};

[···]
```

Try out `GET /users/1` in Insomnia. This time, the server should respond with a JSON object for just user #1. Let's add a similar route for looking up an email by ID:

index.js

```
[···]

let getEmailsRoute = (req, res) => {
  res.send(emails);
};

+ let getEmailRoute = (req, res) => {
+   let email = emails.find(email => email.id === req.params.id);
+   res.send(email);
+ };

[···]

router.get('/emails', getEmailsRoute);
+ router.get('/emails/:id', getEmailRoute);

app.use(router);

[···]
```

Keep in mind that, since the entire path is just a string, dynamic segments in `req.params` will always be strings too, even if the value looks like a number. If your IDs are stored as integers in your database, the type mismatch can cause some irritating bugs.

Create a new Insomnia request for `GET /emails/1` and test it out. It should behave pretty much the same as the `getUserRoute()` and respond with just email #1.

Using Multiple Routers

You don't need to have just one router. In fact, it's a good idea to create a dedicated router for each type of resource. That means we should have a router for all `/users` routes and another router for all `/emails` routes:

index.js

```
[···]

- let router = express.Router();
+ let usersRouter = express.Router();

- router.get('/users', getUsersRoute);
+ usersRouter.get('/users', getUsersRoute);
- router.get('/users/:id', getUserRoute);
+ usersRouter.get('/users/:id', getUserRoute);

+ let emailsRouter = express.Router();

- router.get('/emails', getEmailsRoute);
+ emailsRouter.get('/emails', getEmailsRoute);
- router.get('/emails/:id', getEmailRoute);
+ emailsRouter.get('/emails/:id', getEmailRoute);

- app.use(router);
+ app.use(usersRouter);
+ app.use(emailsRouter);

[···]
```

Why not leave all the routes on one router? As the backend's functionality grows, the codebase will tend to scale vertically — longer files on average — instead of horizontally.

By splitting each resource into its own router, the average file size will stay the same as backend functionality grows; only the number of files will increase. Software that grows like this is generally easier to maintain. It means each file focuses on one responsibility, and it becomes easier for several developers to work in the same codebase without merge conflicts. It's like a router for your routers!

When adding a router, we can specify a path prefix to `app.use()` that will be shared by all its routes. Let's try it in `index.js`:

index.js

```
[···]

let usersRouter = express.Router();

- usersRouter.get('/users', getUsersRoute);
+ usersRouter.get('/', getUsersRoute);
- usersRouter.get('/users/:id', getUserRoute);
+ usersRouter.get('/:id', getUserRoute);

let emailsRouter = express.Router();

- emailsRouter.get('/emails', getEmailsRoute);
+ emailsRouter.get('/', getEmailsRoute);
- emailsRouter.get('/emails/:id', getEmailRoute);
+ emailsRouter.get('/:id', getEmailRoute);

- app.use(usersRouter);
+ app.use('/users', usersRouter);
- app.use(emailsRouter);
+ app.use('/emails', emailsRouter);

[···]
```

As always, make sure your requests still work in Insomnia.

Extracting Routers into Files

Now that we have decoupled the routes of `/users` and `/emails`, it's a cinch to extract the `usersRouter()` and `emailsRouter()` into separate files.

Create a new folder called `routes/` in your project directory, then create two files: `routes/users.js` and `routes/emails.js`. Move the `usersRouter()` and its corresponding route functions into `routes/users.js`:

routes/users.js

```
+ const express = require('express');
+ const users = require('../fixtures/users');
+
+ let getUsersRoute = (req, res) => {
+   res.send(users);
+ };
+
+ let getUserRoute = (req, res) => {
+   let user = users.find(user => user.id === req.params.id);
+   res.send(user);
+ };
+
+ let usersRouter = express.Router();
+
+ usersRouter.get('/', getUsersRoute);
+ usersRouter.get('/:id', getUserRoute);
+
+ module.exports = usersRouter;
```

Don't forget to `require()` Express and export the `usersRouter()` function. Let's do the same and move `emailsRouter()` with its route functions into `routes/emails.js`:

routes/emails.js

```
+ const express = require('express');
+ const emails = require('../fixtures/emails');
+
+ let getEmailsRoute = (req, res) => {
+   res.send(emails);
+ };
+
+ let getEmailRoute = (req, res) => {
+   let email = emails.find(email => email.id === req.params.id);
+   res.send(email);
+ };
+
+ let emailsRouter = express.Router();
+
+ emailsRouter.get('/', getEmailsRoute);
+ emailsRouter.get('/:id', getEmailRoute);
+
+ module.exports = emailsRouter;
```

Now we can delete the migrated code from `index.js` and `require()` the two router files:

index.js

```
  [···]

- const users = require('./fixtures/users');
- const emails = require('./fixtures/emails');

+ const usersRouter = require('./routes/users');
+ const emailsRouter = require('./routes/emails');

  let app = express();

- let getUsersRoute = (req, res) => { ... };
- let getUserRoute = (req, res) => { ... };
- let getEmailsRoute = (req, res) => { ... };
- let getEmailRoute = (req, res) => { ... };
-
- let usersRouter = express.Router();
-
- usersRouter.get('/', getUsersRoute);
- usersRouter.get('/:id', getUserRoute);
-
- let emailsRouter = express.Router();
-
- emailsRouter.get('/', getEmailsRoute);
- emailsRouter.get('/:id', getEmailRoute);

  [···]
```

That's a lot of deletions, so make sure to test all your routes in Insomnia. By now, `index.js` file should be nice and short:

index.js

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

const usersRouter = require('./routes/users');
const emailsRouter = require('./routes/emails');

let app = express();

app.use('/users', usersRouter);
app.use('/emails', emailsRouter);

app.listen(3000);
```

Our codebase went through a lot of changes — here's a quick recap:

1. We applied the Router design pattern to extract individual routes into functions and remove the growing `if...else` statement.
2. We replaced our handwritten `router()` with `express.Router()` without modifying any route functions.
3. We added routes to get a user or email by ID using dynamic segments.
4. We split routes for `/users` and `/emails` into two routers that live in separate files.

As our backend continues to grow, `index.js` won't grow that much. Instead, the number of files will grow as we add backend functionality. Moreover, very little of our existing code will be modified: we will mostly add new code.

That's a strong indicator of maintainable, robust code: well-designed code doesn't change, it is only added to. As little existing code should be changed when adding new functionality. That way, existing routes and their tests are unlikely to break!

Go Further

Routing on the `Accept` Header

Remember last chapter's challenge? Unless you discovered Express's `res.format()` method, your code for multiple response types is probably a bunch of `if...else` statements. Use the Router design pattern to refactor that code, then extract the entire router into a function that you can use for any route!

A few tips:

- Start by extracting the entire `if...else` statement into a function called `respond()` with a signature like `(req, res, data) =>`. This `respond()` function should follow the spirit of the `router()` function we extracted earlier.

- As you apply the Router design pattern, the main difference is how data gets converted to a string. Those differences can be extracted to an object called `formatters` that maps from content types — `json`, `csv`, `xml` — to a function that converts data to a string. The `formatters` object will closely resemble the `routes` object we created earlier.

Chapter 4

Working with Request Bodies

Pony Express can retrieve existing emails, but clients also need a backend route to send new emails. The conventional way to send details about a new resource — like an email — is to include a **request body**.

In the same way the server's response can include a body, a client's request can include a body. But what should the request body look like for creating a new email? Well, the request body will look identical to the response body from `getEmailRoute()`: a JSON-formatted string.

Request Body Lifecycle

The RESTful convention for creating a new email would be a request to `POST /emails` that includes a JSON-formatted request body. If we were to handwrite the HTTP request, it would look like this:

```
POST /emails HTTP/1.1
Content-Type: application/json

{
  "from": "1",
  "to": "2",
  "subject": "Hello",
  "body": "World"
}
```

Let's add a new route to respond to this request in `routes/emails.js`:

routes/emails.js

```
  [···]

  let getEmailRoute = (req, res) => { ... };

+ let createEmailRoute = (req, res) => {
+   console.log('Creating email...');
+ };

  let emailsRouter = express.Router();

  emailsRouter.get('/', getEmailsRoute);
  emailsRouter.get('/:id', getEmailRoute);
+ emailsRouter.post('/', createEmailRoute);

  [···]
```

How does `createEmailRoute()` access the request body? Well, since Express builds on Node's built-in `http` module, the `req` object is a stream object that represents the request body, so we can listen for the `data` and `end` events:

```
  [···]

  let createEmailRoute = (req, res) => {
    console.log('Creating email...');
+   req.on('data', () => console.log('Chunk...'));
+   req.on('end', () => console.log('Received body...'));
  };

  [···]
```

Now to test it out! To build a solid mental model of how request bodies work with Node's `http` module, open a `telnet` session in a new terminal window:

```
$ telnet localhost 3000
```

We'll start with a short JSON request body. Try entering this HTTP request one line at a time, and keep an eye on the backend's terminal as you do:

```
POST /emails HTTP/1.1
Content-Type: application/json

{
  "my": "body"
}
```

Your `telnet` session will probably blow up after you type the first line of the body:

```
$ telnet localhost 3000
POST /emails HTTP/1.1
Content-Type: application/json

{
HTTP/1.1 400 Bad Request

Connection closed by foreign host.
```

What just happened? Although it looks like you're sending a request to the server all at once, the HTTP protocol allows you to send a chunk at a time. Every time you hit the return key, `telnet` sends the entire line to the backend. If you were watching the backend's terminal, the logs indicate that `createEmailRoute()` started handling the request as soon as you hit return twice in `telnet`.

However, Express wasn't expecting to receive a body at all. Requests must include a `Content-Length` header to let the server know a body will follow and how long the body will be. If you try to include a body with a request but forget the `Content-Length` header, Express will automatically respond with `400 Bad Request`.

Let's retry the request, but this time include a `Content-Length` header to specify the body's length in bytes. It might help to compose your HTTP request in a text editor first for tweaking, then copy-paste the entire request into `telnet`. Depending on your operating system and text editor, line breaks count as 1 or 2 bytes. In `telnet`, line breaks count as 2 bytes for a total byte length of 20:

```
POST /emails HTTP/1.1
Content-Type: application/json
Content-Length: 20

{
  "my": "body"
}
```

When you finish typing the request body in `telnet` and hit return, nothing will happen. However, there should be a few log statements in the backend terminal:

```
Creating email...
Chunk...
Chunk...
Chunk...
Received body...
```

That's good! It means Express received our request body without complaints, but `createEmailRoute()` isn't doing anything with the body yet.

Reading Request Bodies

The request body is streamed over in chunks, but the `createEmailRoute()` function needs the entire JSON-formatted request body. Let's write a helper function to buffer up the request body into one string. Make a new directory called `lib/`, then create a file called `lib/read-body.js`:

lib/read-body.js

```
+ let readBody = (req) =>
+   new Promise(resolve => {
+     let chunks = [];
+     req.on('data', (chunk) => {
+       console.log('Chunk:', chunk.toString());
+       chunks.push(chunk);
+     });
+     req.on('end', () => {
+       resolve(Buffer.concat(chunks));
+     });
+   });
+
+ module.exports = readBody;
```

The `readBody()` function collects each "chunk" of the body it receives and returns a Promise that resolves once the entire request body has been received. When sending an HTTP request with `telnet`, each chunk will be one line of the body. In Node, chunks are `Buffer` objects — Node's binary datatype — but can easily be converted to a string for debugging.

In `routes/emails.js`, invoke `readBody()` from `createEmailRoute()`:

routes/emails.js

```
  const express = require('express');
+ const readBody = require('../lib/read-body');
  const emails = require('../fixtures/emails');

  [···]

- let createEmailRoute = (req, res) => {
+ let createEmailRoute = async (req, res) => {
    console.log('Creating email...');
+   let body = await readBody(req);
-   req.on('data', () => console.log('Chunk...'));
-   req.on('end', () => console.log('Received body...'));

+   console.log('Received body...');
+   console.log(body.toString());
  };

  [···]
```

Since `readBody()` returns a Promise, we changed `createEmailRoute()` to an async function so we can use the `await` operator.

Hop back into `telnet` to retry your `POST /emails` request and keep an eye on the server logs in the other terminal window. As you type each line of the body, `readBody()` should immediately print each line:

```
Creating email...
Chunk: {
Chunk:   "my": "body"
Chunk: }
Received body...
{
  "my": "body"
}
```

It works! Thanks to JavaScript's highly asynchronous nature, a function like `createEmailRoute()` can handle a slow request without complicating the code. Here's a visual recap of the timeline for an HTTP request:

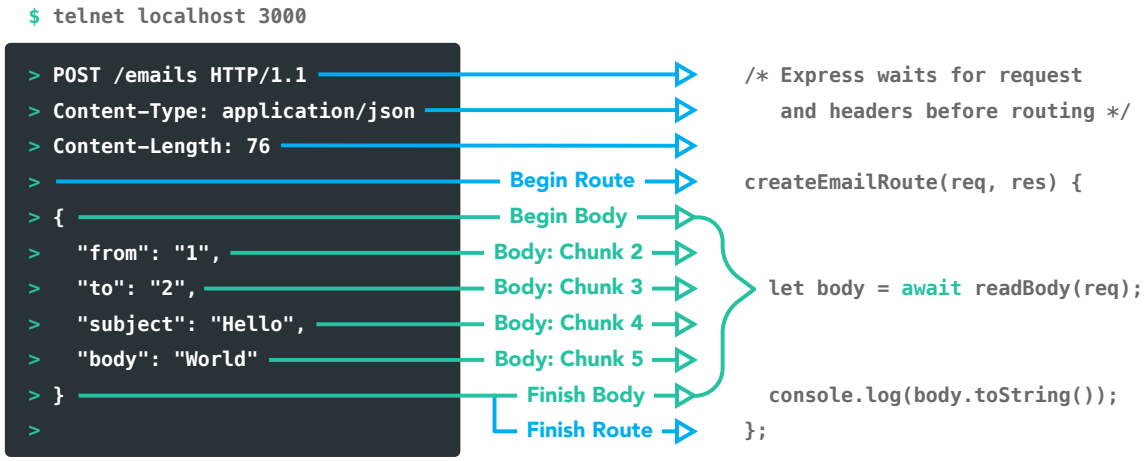


Figure 4.1: *Timeline of handling an HTTP request with a body.*

Finishing Up the Create Endpoint

Now let's use the request body to add a new email to the `emails` array:

routes/emails.js

```
  [···]

  let createEmailRoute = async (req, res) => {
-   console.log('Creating email...');
    let body = await readBody(req);
-   console.log('Received body...');
-   console.log(body.toString());
+   let newEmail = JSON.parse(body);
+   emails.push(newEmail);
  };

  [···]
```

Rerun your `POST /emails` request in `telnet`, but this time specify a full email in the request body. It may take a few tries to get the request just right, so you may want to compose the entire HTTP request in a text editor, then copy-paste it into `telnet`:

```
POST /emails HTTP/1.1
Content-Type: application/json
Content-Length: 76

{
  "from": "1",
  "to": "2",
  "subject": "Hello",
  "body": "World"
}
```

To make sure it worked, hop into Insomnia and check the response for `GET /emails`. The response should include a new email!

We have a couple loose ends to wrap up. First, the server doesn't respond to the request after it's finished creating the email. The conventional response after creating a new resource is a `201 Created` status code and a JSON-formatted body:

routes/emails.js

```
  [···]

  let createEmailRoute = async (req, res) => {
    let body = await readBody(req);
    let newEmail = JSON.parse(body);
    emails.push(newEmail);
+   res.status(201);
+   res.send(newEmail);
  };

  [···]
```

Rerun the HTTP request in `telnet`. Once you finish typing the body, the server should respond with exactly the same body:

```
HTTP/1.1 201 Created
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 54

{"from":"1","to":"2","subject":"Hello",
 "body":"World"}
```

One last thing: every email needs an `id` attribute so we can look it up later. We aren't using a database which would generate the ID automatically, but we can write a quick helper function to generate a random ID. Let's put it in a new file, `lib/generate-id.js`:

lib/generate-id.js

```
+ const crypto = require('crypto');
+
+ let generateId = () =>
+   crypto.randomBytes(8).toString('hex');
+
+ module.exports = generateId;
```

In `createEmailRoute()`, we'll generate an ID and add it to the `newEmail` object before pushing it into the `emails` array:

routes/emails.js

```
  const express = require('express');
  const readBody = require('../lib/read-body');
+ const generateId = require('../lib/generate-id');
  const emails = require('../fixtures/emails');

  [···]

  let createEmailRoute = async (req, res) => {
    let body = await readBody(req);
-   let newEmail = JSON.parse(body);
+   let newEmail = { ...JSON.parse(body), id: generateId() };
    [···]
  };

  [···]
```

Try the request once more in `telnet`. The response should include an `"id"` attribute.

```
HTTP/1.1 201 Created
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 78

{"from":"1","to":"2","subject":"Hello",
 "body":"World","id":"df830371a64cab4e"}
```

We made these requests with `telnet` to help dispel the magic around request bodies, but from now on we'll stick to Insomnia.

Let's mirror what we've done in `telnet` with a new Insomnia request. First, create a new Insomnia request for `POST /emails` — don't forget to change the method from `GET` to `POST`. Then, to include a JSON-formatted body, select the "Body" tab and change the type from "No Body" to "JSON".

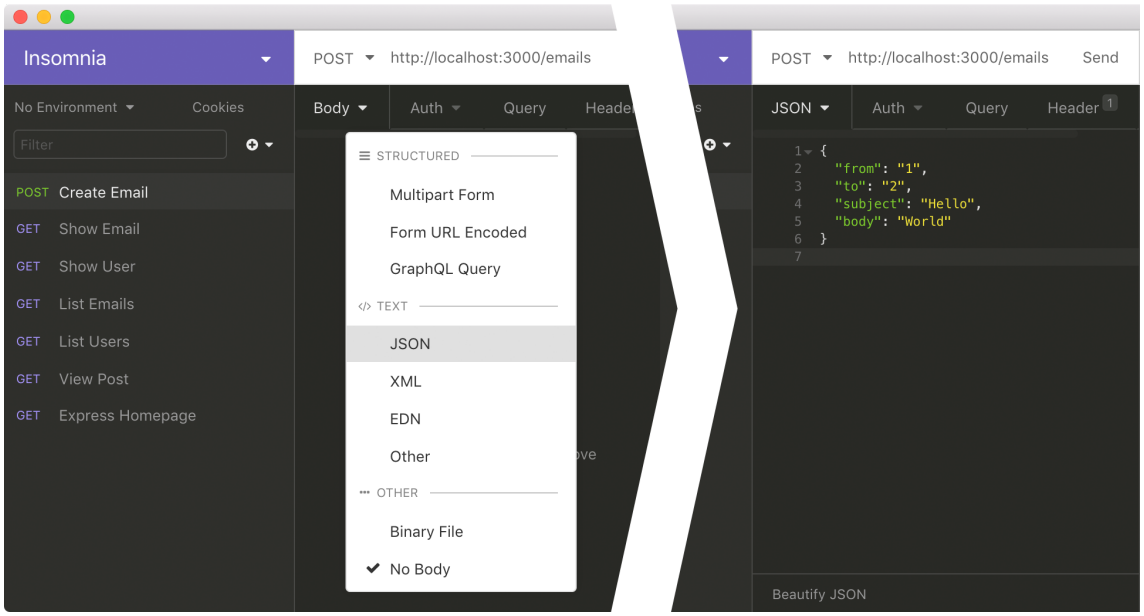


Figure 4.2: *Including a JSON request body in Insomnia.*

After changing the body type, paste in the JSON-formatted email contents you were using in `telnet`. Insomnia will take the initiative and automatically include a `Content-Length` and `Content-Type` header, so don't add your own.

Try sending the Insomnia request. The response should look just like it did in `telnet`.

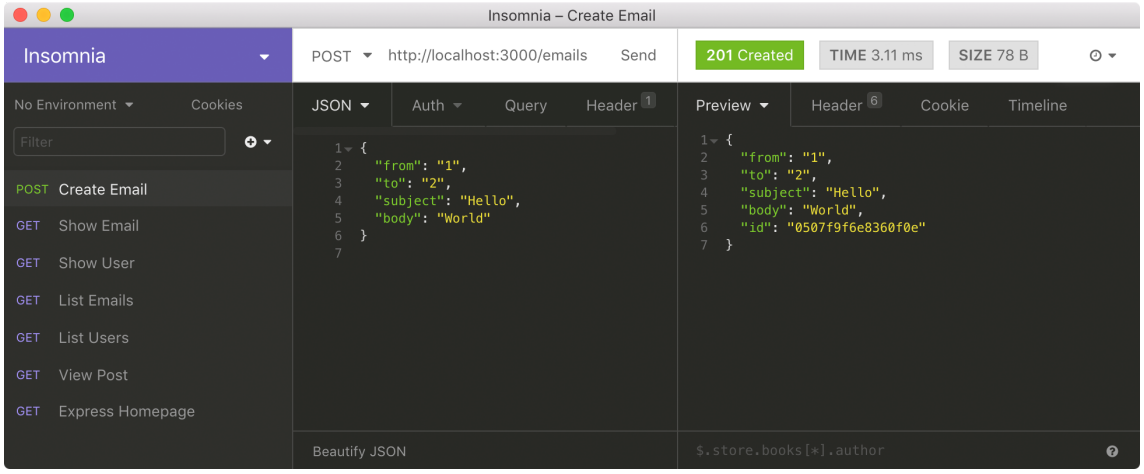


Figure 4.3: *Creating a new email from Insomnia.*

Update and Delete

Now that Pony Express supports creating and viewing emails, let's finish up the **CRUD** (Create-Read-Update-Delete) functionality with routes to update or delete an existing email in `routes/emails.js`:

routes/emails.js

```
  [···]

  let createEmailRoute = async (req, res) => { ... };

+ let updateEmailRoute = async (req, res) => {
+   let body = await readBody(req);
+   let email = emails.find(email => email.id === req.params.id);
+   Object.assign(email, JSON.parse(body));
+   res.status(200);
+   res.send(email);
+ };

  let emailsRouter = express.Router();

  emailsRouter.get('/', getEmailsRoute);
  emailsRouter.get('/:id', getEmailRoute);
  emailsRouter.post('/', createEmailRoute);
+ emailsRouter.patch('/:id', updateEmailRoute);

  [···]
```

`updateEmailRoute()` is almost exactly the same as `createEmailRoute()`, but rather than adding a new entry to `emails`, it mutates the existing email entry with `Object.assign()`. That's generally a bad practice, but we just used mutation to keep things simple. No sweat, production backends would typically use a database anyway.

In Insomnia, make a new request to update the subject of an existing email, such as `PATCH /emails/1`. The request body should be set to JSON and look like:

```
{
  "subject": "I've been changed!"
}
```

After sending the request, follow up with a request to `GET /emails/1` to confirm that the subject changed. Keep in mind that these changes aren't backed by a database, so every time the backend restarts, the fixture data — `emails` and `users` — will be reset. Since `nodemon` restarts every time the code changes, those updates will disappear frequently.

There's just one more route to go! Add a route for deleting an email:

routes/emails.js

```
  [···]

  let updateEmailRoute = async (req, res) => { ... };

+ let deleteEmailRoute = (req, res) => {
+   let index = emails.findIndex(email => email.id === req.params.id);
+   emails.splice(index, 1);
+   res.sendStatus(204);
+ };

  let emailsRouter = express.Router();

  [···]
  emailsRouter.patch('/:id', updateEmailRoute);
+ emailsRouter.delete('/:id', deleteEmailRoute);

  [···]
```

The `deleteEmailRoute()` function looks up an email by ID, then mutates the `emails` array to remove it. Again, mutation is generally bad practice, but is used for simplicity.

When deleting a resource, it's conventional to respond with the `204 No Content` status code and omit the response body. Express includes a shorthand method to simultaneously set the status code and end the response, `res.sendStatus()`.

Create a new request in Insomnia for deleting an email, such as `DELETE /emails/1`. To verify that the email was successfully deleted, follow up with a `GET /emails` request and ensure email #1 is no longer listed.

Express `.route()` Method

CRUD endpoints like `/emails` usually have a total of five routes. Because the route path is either `"/"` or `"/:id"`, some developers prefer to use the Express Router's `.route()` method to cut down the repetition:

routes/emails.js

```
[···]

- emailsRouter.get('/', getEmailsRoute);
- emailsRouter.get('/:id', getEmailRoute);
- emailsRouter.post('/', createEmailRoute);
- emailsRouter.patch('/:id', updateEmailRoute);
- emailsRouter.delete('/:id', deleteEmailRoute);

+ emailsRouter.route('/')
+   .get(getEmailsRoute)
+   .post(createEmailRoute)
+ ;
+
+ emailsRouter.route('/:id')
+   .get(getEmailRoute)
+   .patch(updateEmailRoute)
+   .delete(deleteEmailRoute)
+ ;

[···]
```

The `.route()` method accepts the path shared between routes — such as `.get()` and `.post()` — and provides chaining syntax. This syntax is completely optional: in fact, it is often preferable to stick with the prior format because it keeps version control commits tidier when adding or removing routes. The `.route()` method can occasionally make decoupling routes unnecessarily noisy.

The choice is yours!

Go Further

That was a long chapter, so you deserve a break. There are no challenges, except to celebrate completing the first module!

Part II

Middleware

Chapter 5

Middleware

Often the same behavior needs to be added to a group of routes. For example, most backends log every incoming request to the terminal for debugging and production audits. How could we add logging to Pony Express?

Right now, it's simple enough: we just prepend `console.log()` to each route function. For example, we could start in `routes/emails.js`:

```
routes/emails.js

  [···]

  let getEmailsRoute = (req, res) => {
+   console.log('GET /emails');
    [···]
  };

  let getEmailRoute = (req, res) => {
+   console.log('GET /emails/' + req.params.id);
    [···]
  };

  let createEmailRoute = async (req, res) => {
+   console.log('POST /emails');
    [···]
  };

  [···]
```

Well, that's awful. `console.log()` is basically copy-paste with minor changes, so if we ever want to change the logging style, we would need to update each instance of `console.log()`. We can solve that partly by moving the duplication into a function, but we will still need to invoke that function in every route.

Go ahead and delete all those `console.log()` statements from `routes/emails.js`. How can we prevent this duplication and add logging behavior to all routes without modifying them?

Cross Cutting with Middleware

Express provides a method — `app.use()` — to insert a function that runs before any routes below it. Let's try it in `index.js`:

index.js

```
  [···]

  let app = express();

+ let logger = (req, res, next) => {
+   console.log(req.method + ' ' + req.url);
+ };

+ app.use(logger);
  app.use('/users', usersRouter);
  app.use('/emails', emailsRouter);

  [···]
```

Notice the signature of the `logger()` function. Like a route function, it receives a request and response object, but it also receives a third argument called `next`. Any function with this signature is called **middleware**.

When a request comes in, the `logger()` middleware function runs before any of the routers added below it with `app.use()`. These functions are called middleware because they are sandwiched between each other, and the collective sandwich of these middleware functions is called the **middleware stack**.

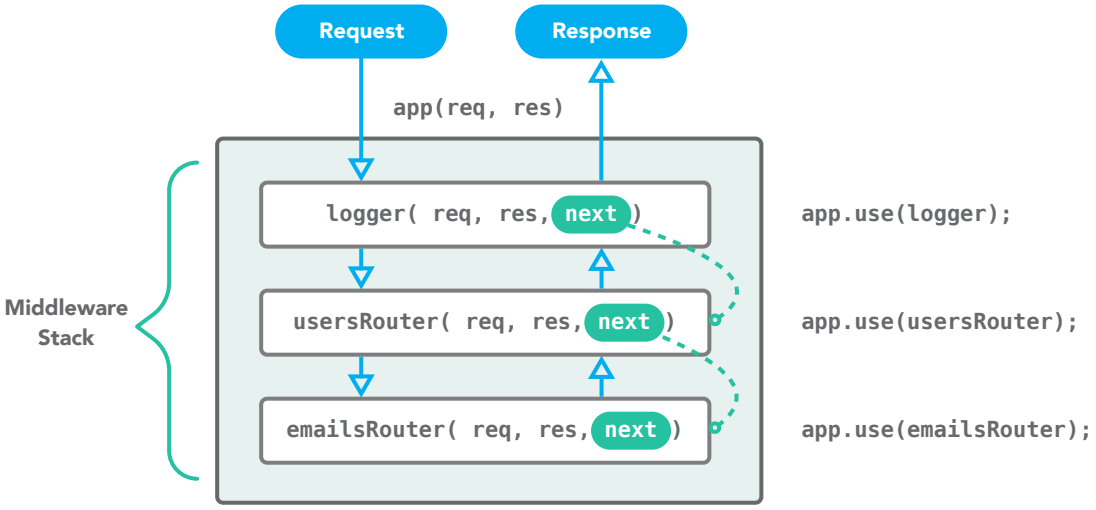


Figure 5.1: *Each middleware function in the stack gets to run before those below it.*

You may not have realized it, but there were already a couple layers in your middleware stack: `usersRouter()` and `emailsRouter()` are middleware functions! Every instance of `app.use()` adds a new layer to the bottom of the stack.

Hop into Insomnia and try a few requests like `GET /users` and `GET /emails`. In the terminal, the backend now prints out the request method and path for any route! However, Insomnia seems to be hanging:

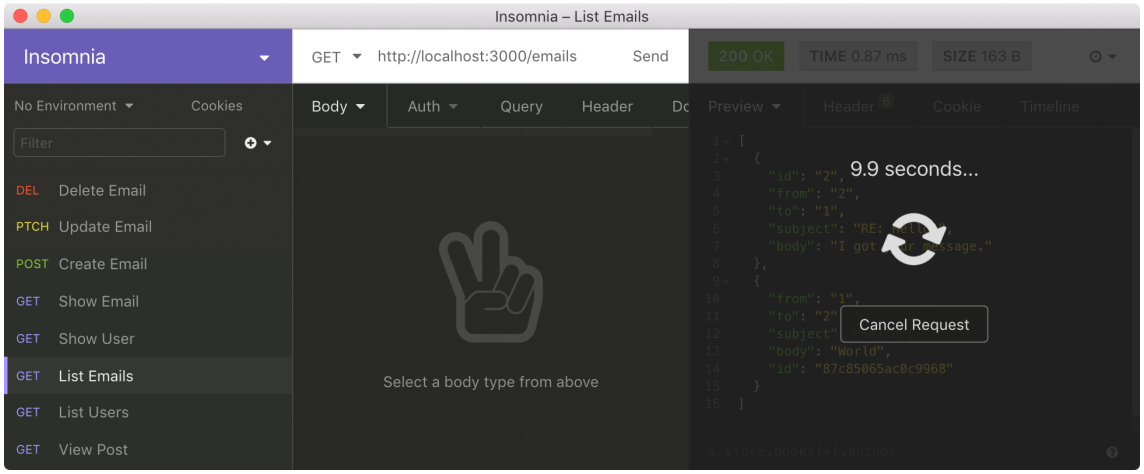


Figure 5.2: *Looks like the request is hanging.*

What's going on? Middleware functions have a lot of power: not only can they be inserted before routes, but they can decide whether to continue to the routes or skip them altogether! To continue to the routes — the next layer in our middleware stack — the middleware must invoke the third argument it received, `next()`:

index.js

```
  [···]

  let logger = (req, res, next) => {
    console.log(req.method + ' ' + req.url);
+   next();
  };

  [···]
```

Try a few requests with Insomnia. The backend should still log each request, but now the routes should behave as they did before the hang.

Way to go, you wrote your first middleware function! Middleware is both a general design pattern and a concrete feature in backend libraries like Express. Express Middleware helps us reuse complex behaviors — sometimes called **cross cutting concerns** — across routes. Since middleware tends to be entirely decoupled from the routes, it's incredibly easy to reuse middleware in other projects. In fact, let's move `logger()` into a new file called `lib/logger.js`:

lib/logger.js

```
+ let logger = (req, res, next) => {
+   console.log(req.method + ' ' + req.url);
+   next();
+ };
+
+ module.exports = logger;
```

Don't forget to wire it up in `index.js`:

index.js

```
  const express = require('express');
+ const logger = require('./lib/logger');

  [···]

- let logger = (req, res, next) => {
-   console.log(req.method + ' ' + req.url);
-   next();
- };

  [···]
```

Passing Data to Routes

You know what else is irritating? Parsing JSON-formatted request bodies in `createEmailRoute()` and `updateEmailRoute()`! Let's make a middleware function to do that instead. Create a new file called `lib/json-body-parser.js`:

lib/json-body-parser.js

```
+ const readBody = require('./read-body');
+
+ let jsonBodyParser = async (req, res, next) => {
+   let body = await readBody(req);
+   let json = JSON.parse(body);
+   next();
+ };
+
+ module.exports = jsonBodyParser;
```

Unlike our `logger()` middleware, `jsonBodyParser()` does some work that needs to be passed to the routes. How should we feed the parsed JSON to the routes in the next layer? In Express, it's common to add a property to the request object. We'll put the parsed JSON body in `req.body`:

lib/json-body-parser.js

```
  [...]

  let jsonBodyParser = async (req, res, next) => {
    let body = await readBody(req);
-   let json = JSON.parse(body);
+   req.body = JSON.parse(body);
    next();
  };

  [...]
```

Now any route that comes after `jsonBodyParser()` can access the JSON-formatted request body in `req.body`. Where should `jsonBodyParser()` go in the middleware stack? We could try adding it to `index.js` like this:

index.js

```
  const express = require('express');
  const logger = require('./lib/logger');
+ const jsonBodyParser = require('./lib/json-body-parser');

  [···]

  app.use(logger);
+ app.use(jsonBodyParser);
  app.use('/users', usersRouter);
  app.use('/emails', emailsRouter);

  [···]
```

Since all the `/users` and `/emails` routes come after `jsonBodyParser()` runs, we can drop the `readBody()` calls from `createEmailRoute()` and `updateEmailRoute()` in `routes/emails.js`:

routes/emails.js

```
  const express = require('express');
- const readBody = require('../lib/read-body');
  const generateId = require('../lib/generate-id');
  const emails = require('../fixtures/emails');

  [···]

  let createEmailRoute = async (req, res) => {
-   let body = await readBody(req);
-   let newEmail = { ...JSON.parse(body), id: generateId() };
+   let newEmail = { ...req.body, id: generateId() };
    emails.push(newEmail);
    res.status(201);
    res.send(newEmail);
  };

  let updateEmailRoute = async (req, res) => {
-   let body = await readBody(req);
    let email = emails.find(email => email.id === req.params.id);
-   Object.assign(email, JSON.parse(body));
+   Object.assign(email, req.body);
    res.status(200);
    res.send(email);
  };

  [···]
```

Retry your Insomnia requests for `POST /emails` and `PATCH /emails/1`. They should work just as before!

Route Middleware

Sadly not all is well. Try sending a `GET /emails` request with Insomnia. The request seems to be hanging again because an exception is blowing everything up:

```
GET /emails
(node:44439) UnhandledPromiseRejectionWarning:
  SyntaxError: Unexpected end of JSON input
    at JSON.parse (<anonymous>)
    at jsonBodyParser (lib/json-body-parser.js:5:19)
    [···]
```

What's going on? Well, not every route expects a request body, much less a JSON-formatted body. But `jsonBodyParser()` runs before every single route as though a JSON-formatted request body is guaranteed. `GET` requests don't have a body, so `JSON.parse()` is trying to parse an empty string.

There are a few approaches to fix this bug. The typical solution is to make `jsonBodyParser()` a bit more robust to edge cases with some `if...else` statements. However, apart from making our code uglier, it only postpones other bugs that will emerge because it won't solve the underlying design problem: only *two* routes in our backend expect JSON-formatted request bodies!

Inserting middleware with `app.use()` is a bit like using global variables: tempting and easy, but deadly to reusable software. With few exceptions, "global middleware" is a bad design choice because it is more difficult to "opt-out" of middleware in a few routes than it is to "opt-in" where it's needed.

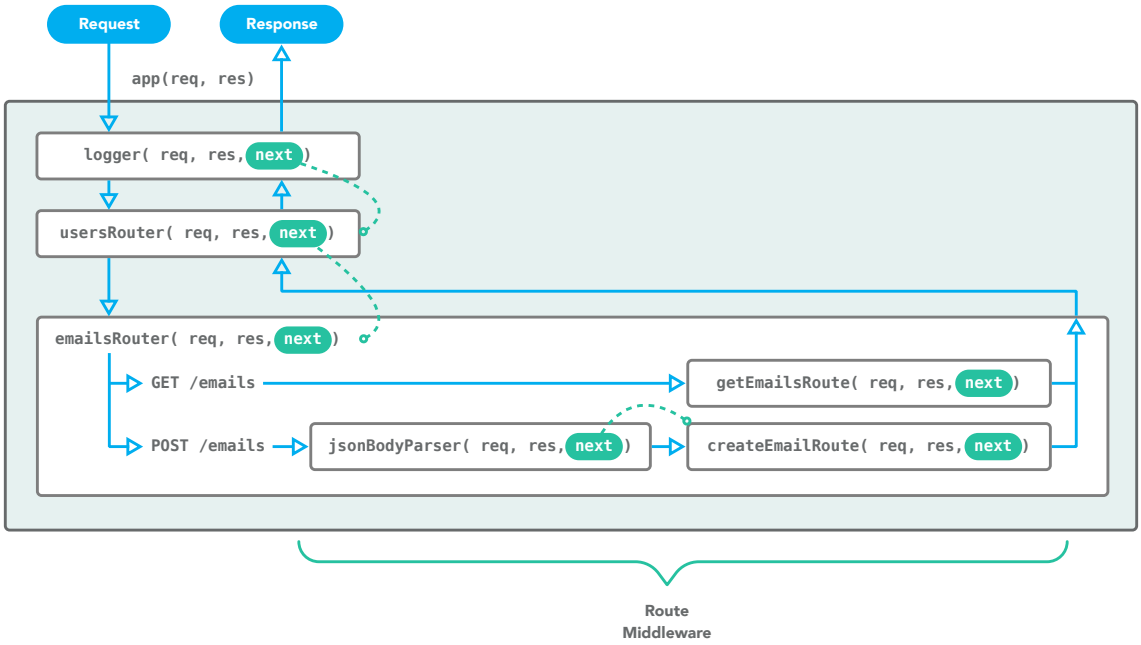


Figure 5.3: *Route middleware is like a personalized stack for just this route.*

Instead of adding global middleware with `app.use()`, we can specify middleware for individual routes with `.get()` and its siblings. Let's try it in `routes/emails.js`:

routes/emails.js

```
  const express = require('express');
+ const jsonBodyParser = require('../lib/json-body-parser');
  const generateId = require('../lib/generate-id');
  const emails = require('../fixtures/emails');

  [···]

  emailsRouter.route('/')
    .get(getEmailsRoute)
-   .post(createEmailRoute)
+   .post(jsonBodyParser, createEmailRoute)
  ;

  emailsRouter.route('/:id')
    .get(getEmailRoute)
-   .patch(updateEmailRoute)
+   .patch(jsonBodyParser, updateEmailRoute)
    .delete(deleteEmailRoute)
  ;

  [···]
```

You can add as many middleware functions for a route as you want. They will execute from left to right, so make sure your route function comes last. To get our code working again, let's nuke `jsonBodyParser()` from the global middleware stack in `index.js`:

index.js

```
  const express = require('express');
  const logger = require('./lib/logger');
- const jsonBodyParser = require('./lib/json-body-parser');

  [···]

  app.use(logger);
- app.use(jsonBodyParser);
  app.use('/users', usersRouter);
  app.use('/emails', emailsRouter);

  [···]
```

Try a few requests again in Insomnia, like `POST /emails`, `PATCH /emails/1` and `GET /emails`. They should all be working as before!

Middleware is Everywhere

Middleware is a fundamental design pattern that emerges in codebases other than the backend. But particularly in a backend, middleware allows us to reuse complex behaviors across many routes. Because we don't have to worry too much about interactions between middleware, the middleware pattern tends to insulate existing code from change — a powerful characteristic for preventing regressions.

It is generally better to add middleware to just the route or group of routes that needs the middleware's functionality. That's because it is always easier for a route to opt-in to middleware than it is to opt-out of global middleware. There are a few compelling exceptions which we'll cover in upcoming chapters.

As mentioned earlier, `usersRouter()` and `emailsRouter()` are middleware functions too. An Express Router is a middleware function that looks for a matching route in its map. If it finds a match, it runs the route, but if no routes matched, it implicitly invokes the `next()` middleware in the stack. Here's what our handwritten `router()` function would like as middleware:

```
let router = (req, res, next) => {
  let route = req.method + ' ' + req.url;
  let handler = routes[route];

  if (handler) {
    handler(req, res);
  } else {
    next();
  }
};

app.use(router);
```

That's not all: route functions like `createEmailRoute()` are also middleware! Although they only list two parameters — `(req, res) =>` — they also receive `next` like any other middleware function.

```
- let createEmailRoute = async (req, res) => {
+ let createEmailRoute = async (req, res, next) => {
    [···]
  };
```

In JavaScript, the number of arguments used to invoke a function doesn't need to match the number of parameters listed in the function definition. Route functions are generally the end of the line for an HTTP request, so they rarely need the last `next` argument.

Middleware really is everywhere!

Go Further

Error Handling Middleware

Pony Express is happy — that is, it only deals with the happy paths. But not all paths are happy: what happens if a client requests an email that doesn't exist, such as `GET /emails/4`?

At the moment, it responds with `200 OK` and an empty body. That response is unconventional and likely to frustrate frontend developers. Instead, the backend should respond the same way it does to any path that doesn't exist: with `404 Not Found`.

This response logic tends to get duplicated across many routes, but there are a few ways to DRY it up. One way is to throw a custom error from the route:

routes/emails.js

```
  [···]

+ class NotFound extends Error {
+   constructor(message) {
+     super(message);
+     this.name = 'NotFound';
+   }
+ }

  let getEmailRoute = (req, res) => {
    let email = emails.find(email => email.id === req.params.id);
+   if (!email) { throw new NotFound(); }
    res.send(email);
  };

  [···]
```

Exceptions have some unique design advantages: because they immediately interrupt the route and all middleware below it, they help keep `if...else` statements to a minimum. Moreover, exceptions naturally "bubble up" until caught, so it's straightforward to handle exceptions in one place.

Express supports a special type of middleware — **error handling middleware** — that runs when a layer above it in the middleware stacks throws an error. The only difference between regular middleware and error handling middleware is the addition of a fourth argument called `err`:

```
let notFoundHandler = (err, req, res, next) => {
  if (err instanceof NotFound) {
    // Respond to the client
    [···]
    // Extinguish the error
    next();
  } else {
    // Let someone else deal with the error
    next(err);
  }
};
```

Throw a `NotFound` error from all the `/users` and `/emails` routes that could potentially look up a non-existent resource, then add one error handling middleware that catches `NotFound` errors and responds with a `404 Not Found` status code and body of your choosing.

Chapter 6

Common Middleware

Middleware is pretty awesome. It's such a good design pattern for code reuse that the most common backend behaviors are already coded up and can be added with just a few lines of code.

We will use very few third-party libraries in Pony Express, but these particular packages are staples of the Express community. Most of the middleware in this chapter used to be bundled with Express, but over time they have been extracted to separate packages. Thus, unlike many libraries, the Express core has *shrunk* in newer releases.

Logging with Morgan

Let's start by fancying up our request logger. The Express team maintains logging middleware called Morgan that will easily replace our own logger:

```
$ npm install morgan
```

We could delete `lib/logger.js`, but to show how easy it is to swap in third-party middleware without influencing dependent files, let's modify `lib/logger.js`:

lib/logger.js

```
+ const morgan = require('morgan');

- let logger = (req, res, next) => {
-   console.log(req.method + ' ' + req.url);
-   next();
- };
+ let logger = morgan('tiny');

  module.exports = logger;
```

Send a few requests with Insomnia. There should be some even fancier log statements in the terminal!

```
GET /users 200 202 - 4.171 ms
GET /emails 200 165 - 0.555 ms
GET /users/1 200 66 - 0.520 ms
GET /emails/1 200 80 - 0.304 ms
POST /emails 201 78 - 1.701 ms
PATCH /emails/1 200 92 - 0.404 ms
DELETE /emails/1 204 - - 0.407 ms
```

It's easy to swap in a middleware upgrade because all middleware has the same signature: `(req, res, next) =>`.

By the way, is it bad that `logger()` is part of the global middleware stack? For most backends, every single request should be logged with a consistent format. Consequently, logging is a canonical example of when to add middleware globally.

Body Parser

What other handwritten middleware could we obliterate and replace with a third-party package? Well, we wrote a fair amount of code for parsing JSON-formatted request bodies, and we can drop in Express's own body-parser package:

```
$ npm install body-parser
```

The `body-parser` package includes middleware for parsing several body formats, including JSON:

routes/emails.js

```
  const express = require('express');
- const jsonBodyParser = require('../lib/json-body-parser');
+ const bodyParser = require('body-parser');
  [···]

  emailsRouter.route('/')
    .get(getEmailsRoute)
-   .post(jsonBodyParser, createEmailRoute)
+   .post(bodyParser.json(), createEmailRoute)
  ;

  emailsRouter.route('/:id')
    .get(getEmailRoute)
-   .patch(jsonBodyParser, updateEmailRoute)
+   .patch(bodyParser.json(), updateEmailRoute)
    .delete(deleteEmailRoute)
  ;

  [···]
```

Test out your `POST /emails` and `PATCH /emails/1` requests in Insomnia. They should work exactly as before! Unless you're feeling sentimental, go ahead and delete `lib/read-body.js` and `lib/json-body-parser.js`.

Middleware Factories

What's up with the syntax for `bodyParser.json()`? It's extremely common to configure a middleware function before inserting it. For example, we could prevent a client from sending a huge JSON request body:

```
// Inline middleware configuration
router.post('/',
  bodyParser.json({ limit: '100kb' }),
  createEmailRoute
);

// Reusing configured middleware
let jsonBodyParser = bodyParser.json({ limit: '100kb' });

router.post('/', jsonBodyParser, createEmailRoute);
router.patch('/:id', jsonBodyParser, updateEmailRoute);
```

You might recognize `.json()` as an example of the Factory design pattern. Specifically, `bodyParser.json()` is a **Middleware Factory** function.

This isn't the first example we've seen: `morgan('tiny')` is also a Middleware Factory, and we passed it an argument — `'tiny'` — to configure the logging format. Middleware Factories make middleware easy to customize without forcing all instances of that particular middleware to share the same configuration.

The most powerful example of a Middleware Factory we've seen is `express.Router()`. That's right! Our `usersRouter` and `emailsRouter` are middleware functions, and `express.Router()` is a Middleware Factory that helped us quickly build complex router logic.

JavaScript excels in this style of functional programming, so we'll try writing our own Middleware Factory in an upcoming chapter.

Compression

We're not done with easy wins yet! A lot of the HTTP traffic between a client and backend is unnecessarily bulky, but we can add compression support — both for the request body and the response body — with the compression package:

```
$ npm install compression
```

It takes just a couple lines in `index.js` to add compression support to the entire backend:

index.js

```
  const express = require('express');
  const logger = require('./lib/logger');
+ const compress = require('compression');

  [···]

  app.use(logger);
+ app.use(compress());
  app.use('/users', usersRouter);
  app.use('/emails', emailsRouter);

  [···]
```

Just like that, our backend can decompress requests and compress responses — quick performance wins for the win! Like logging, compression is another good example of when to use global middleware.

To test out compression, add the following request header to your `GET /emails` request in Insomnia:

```
Accept-Encoding: gzip
```

This tells the backend that Insomnia understands gzipped responses and to compress the response if the backend also supports gzip.

Try sending the request. To verify that Express compressed the response, check the Headers tab in the response section. When a server sends a compressed response, it should include a `Content-Encoding: gzip` header:

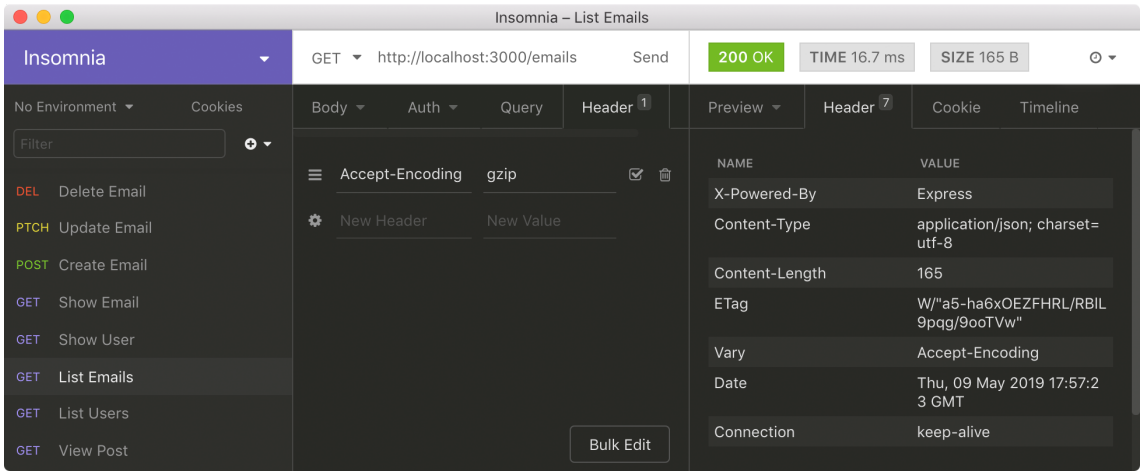


Figure 6.1: *Enabling compression with the `Accept-Encoding` request header.*

Hmm, the `Accept-Encoding` header tells the backend to compress the response, so why doesn't the response have a `Content-Encoding` header? By default, the compression middleware only compresses response bodies when the size is beyond a threshold like 1 kb. It's a good default, but just to make sure we wired everything up correctly, let's temporarily disable the threshold for `compress()`:

index.js

```
[...]

app.use(logger);
- app.use(compress());
+ app.use(compress({ threshold: 0 }));
[...]
```

Retry the `GET /emails` request in Insomnia. This time, the response headers should include a `Content-Encoding: gzip` header.

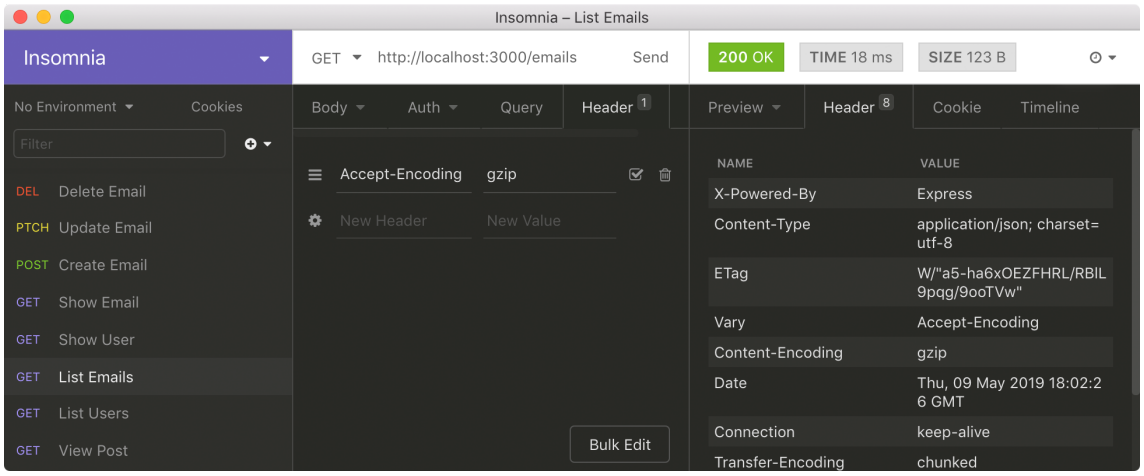


Figure 6.2: *The response is compressed with gzip.*

Compression is working! 1 kb is still a good default, so let's comment out the threshold:

index.js

```
[···]

app.use(logger);
- app.use(compress({ threshold: 0 }));
+ app.use(compress(/* { threshold: 0 } */));
[···]
```

Serving a Frontend

Right now our backend is a pure API, but it would be nice to show a basic webpage when a user opens http://localhost:3000 in their browser. In your project folder, make a directory for frontend files called `public/`, then create a file called `public/index.html`:

public/index.html

```
+ <!DOCTYPE html>
+ <html lang="en">
+ <head>
+   <meta charset="UTF-8">
+   <title>Pony Express</title>
+ </head>
+ <body>
+   <h1>Pony Express</h1>
+ </body>
+ </html>
```

We could add individual routes to respond to requests like `GET /index.html`, but as the frontend grows to include CSS and images, we will need to add more routes. Instead, we could write a single middleware function that examines all incoming requests and maps the URL to a corresponding file in the `public` directory.

You guessed it, there's a middleware package for that: serve-static.

```
$ npm install serve-static
```

The `serve-static` package provides a middleware factory so we can configure which directory to serve files from in `index.js`:

index.js

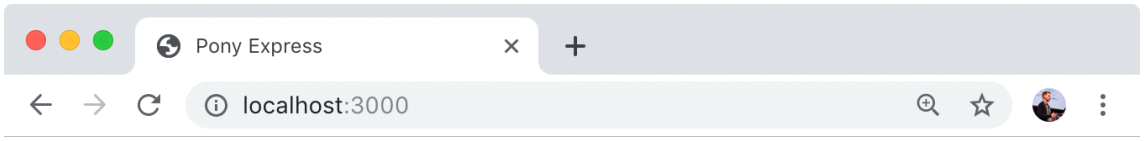
```
  const express = require('express');
  const logger = require('./lib/logger');
  const compress = require('compression');
+ const serveStatic = require('serve-static');

  [···]

  app.use(logger);
  app.use(compress(/* { threshold: 0 } */));
+ app.use(serveStatic('./public'));
  app.use('/users', usersRouter);
  app.use('/emails', emailsRouter);

  [···]
```

Try opening http://localhost:3000 or http://localhost:3000/index.html in your browser. Both should show your homepage instead of a 404 message, and your Insomnia requests should still work too.



Pony Express

Figure 6.3: *Pony Express now has an impressive homepage.*

There's one caveat: the path `./public` is a relative path, but it's not relative to the project directory — it's relative to the terminal's present working directory. In production environments, it's common to boot up a Node server from somewhere other than the project directory, so let's calculate the full path to `./public` based on the project directory:

index.js

```
+ const path = require('path');
  const express = require('express');
  [···]

  app.use(compress(/* { threshold: 0 } */));
- app.use(serveStatic('./public'));
+ app.use(serveStatic(path.join(__dirname, 'public')));
  [···]
```

File Uploads with Multer

Our users may want to include file attachments when creating their emails. Unfortunately, a JSON-formatted request body can't include file uploads. However, there is another well-supported body format that does: **multipart form data**. Like a simple JSON object, multipart form data can represent a dictionary of keys and values: the keys are strings, and the values can be a string or a file.

Create a new directory called `uploads/` in your project. This is where email attachments will be stored after upload. Since `body-parser` doesn't support multipart form data, most developers turn to the multer package:

```
$ npm install multer
```

Like `body-parser`, `multer` is a middleware factory and should only be added to routes that need to support multipart request bodies. Let's add support to `createEmailRoute()` in `routes/emails.js`:

routes/emails.js

```
+ const path = require('path');
  const express = require('express');
  const bodyParser = require('body-parser');
+ const multer = require('multer');
  const generateId = require('../lib/generate-id');
  const emails = require('../fixtures/emails');

+ let upload = multer({ dest: path.join(__dirname, '../uploads') });

  [···]

  emailsRouter.route('/')
    .get(getEmailsRoute)
-   .post(bodyParser.json(), createEmailRoute)
+   .post(
+     bodyParser.json(),
+     upload.array('attachments'),
+     createEmailRoute
+   )
  ;

  [···]
```

We configured an instance of `multer` middleware to handle any files uploaded with the request and save them in the `uploads/` directory we created earlier. As with `serve-static`, it's a good idea to determine the absolute path to the `uploads/` directory.

As mentioned last chapter, a route can have multiple middleware functions that run before it. The `createEmailRoute()` now has two: a JSON body parser and a multipart body parser. Since these two middleware functions only run if they recognize the `Content-Type` header, only one will assign `req.body`. That means clients can submit either format!

The `createEmailRoute()` function can access the uploaded files in `req.files`:

routes/emails.js

```
  [...]

  let createEmailRoute = async (req, res) => {
+   let attachments = req.files.map(file => file.filename);
-   let newEmail = { ...req.body, id: generateId() };
+   let newEmail = { ...req.body, id: generateId(), attachments };
    emails.push(newEmail);
    res.status(201);
    res.send(newEmail);
  };

  [...]
```

Hop back to Insomnia and duplicate your existing request for `POST /emails`, but change the request body type to "Multipart Form" and add a key-value pair for each field of an email:

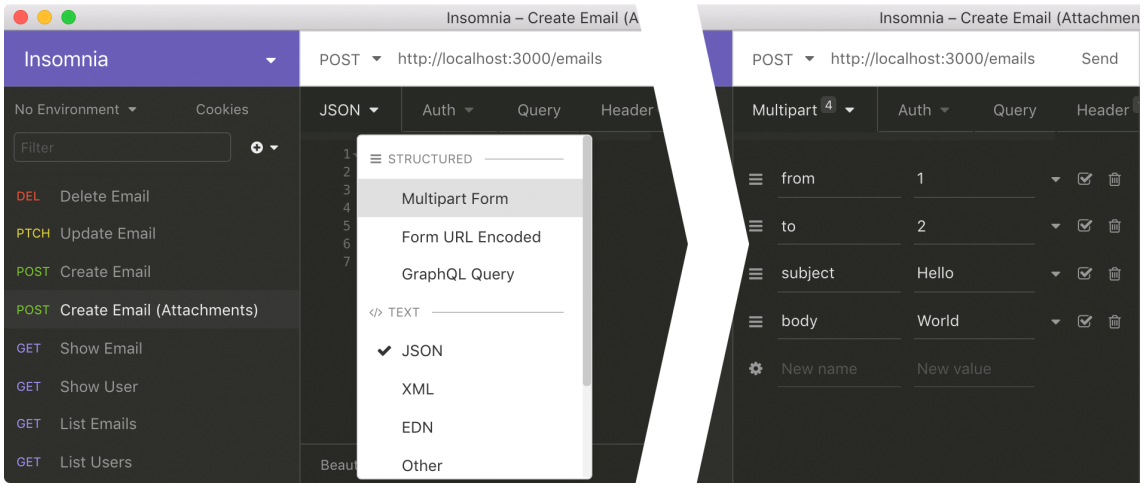


Figure 6.4: *Configuring a multipart request in Insomnia.*

In Insomnia, add one more key-value pair with a key of "attachments" and change the value type to "File":

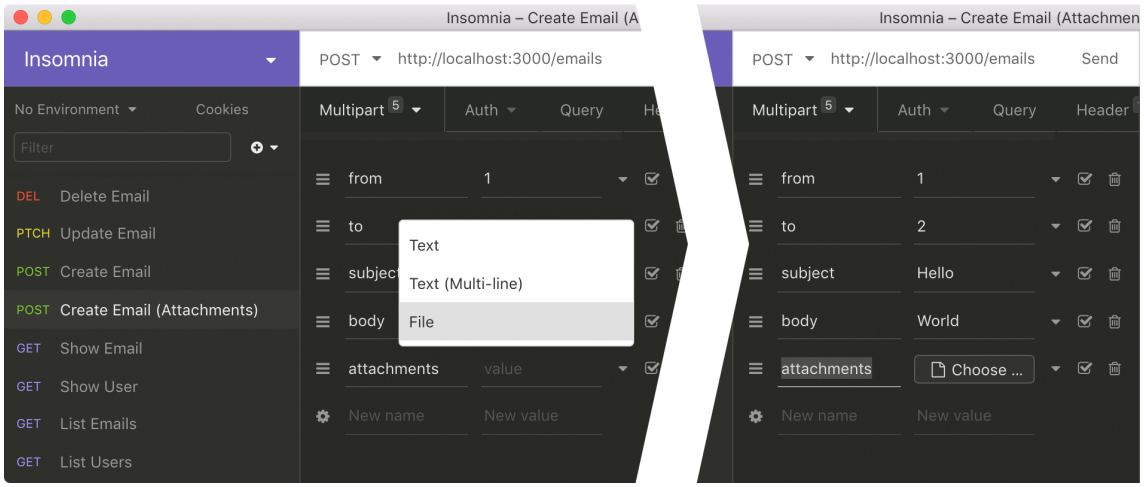


Figure 6.5: *Uploading a file with a multipart form.*

Click "Choose..." and pick a file on your hard drive, such as a small image. Want to upload more than one attachment with the request? Just add a key-value pair per file with the same key of "attachments".

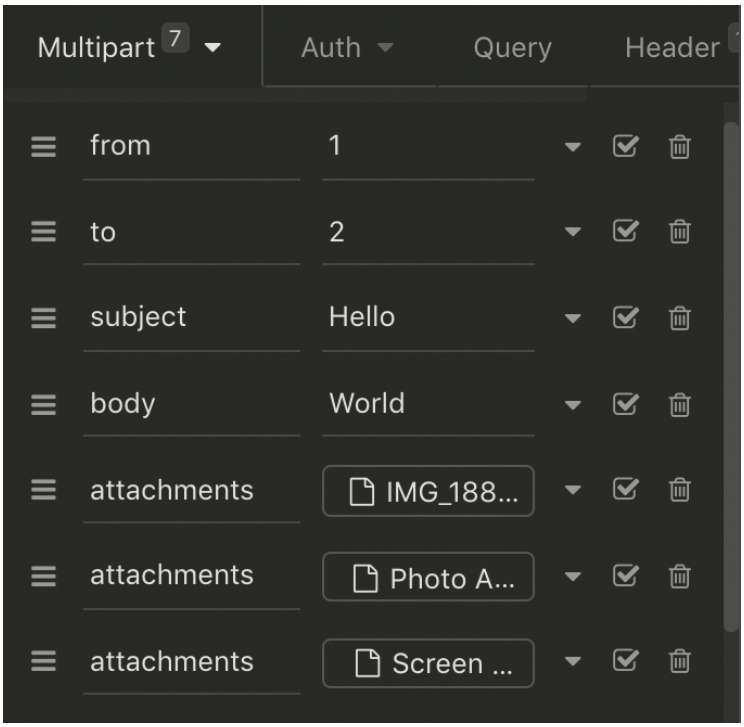


Figure 6.6: *Multipart forms can include several files under the same key.*

Try sending the request and check the `uploads/` directory. If you uploaded three files, you should see three randomly named files in `uploads/` and the JSON response should include an `"attachments"` array that corresponds with those filenames:

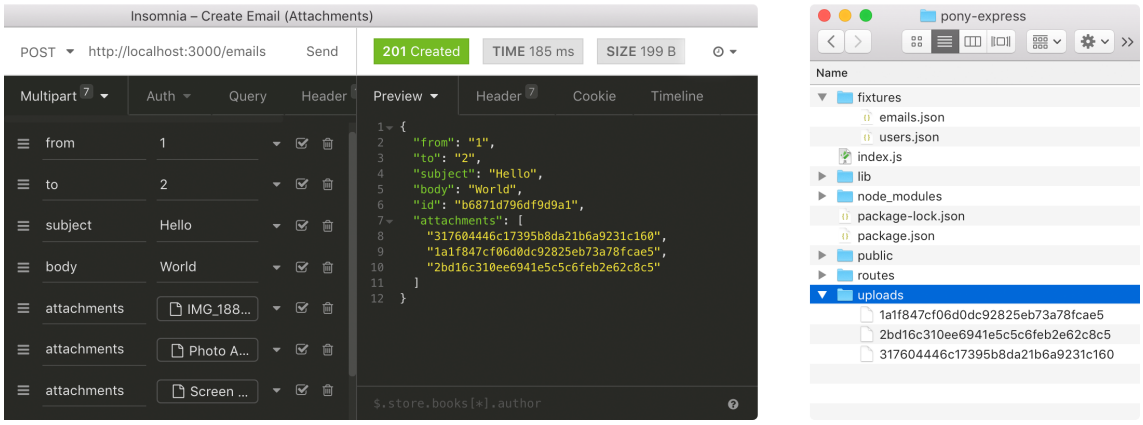


Figure 6.7: *Uploaded files are stored in `uploads/` and listed in the response.*

Serving Static Files with a Path Prefix

If someone wants to view an email attachment, how do they retrieve it from the back-end API? We could add a new route like `GET /uploads/31760444…` for retrieving email attachments. No need to hand-code those routes, we can use `serve-static` again to serve up any files in the `uploads/` directory. Pop back to `index.js`:

index.js

```
[···]

app.use(logger);
app.use(compress(/* { threshold: 0 } */));
app.use(serveStatic(path.join(__dirname, 'public')));
+ app.use('/uploads', serveStatic(path.join(__dirname, 'uploads')));
app.use('/users', usersRouter);
app.use('/emails', emailsRouter);

[···]
```

Unlike serving the `public/` directory, this time we specified a path prefix as the first argument to `app.use()`. Without the prefix, the route would instead be `GET /31760444…`.

Create a new request in Insomnia to test a route like `GET /uploads/31760444…`. Depending on the file type you uploaded, the response will probably look like binary gobbledygook. Not bad!

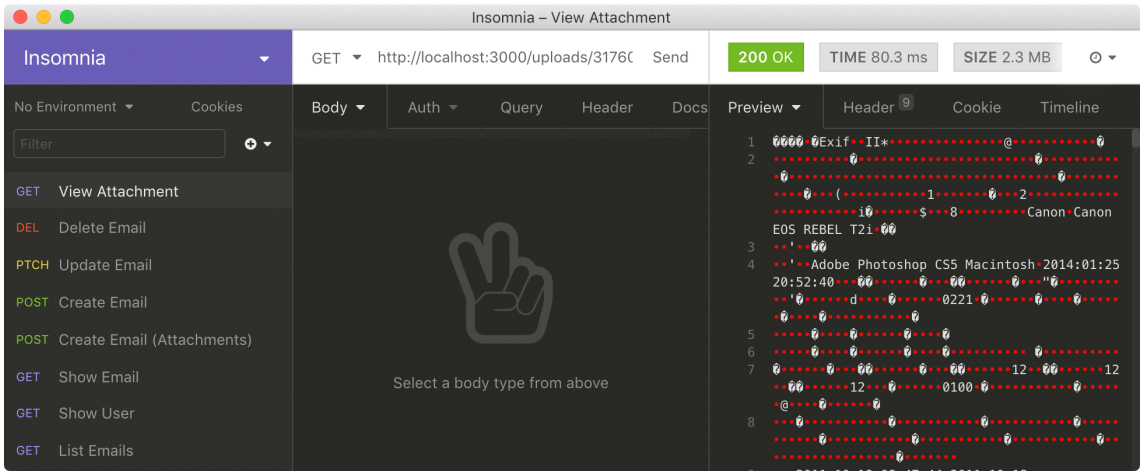


Figure 6.8: *Viewing a previously uploaded email attachment.*

Handling file uploads can be tedious in many backend frameworks, but thanks to the middleware pattern it only takes a few lines of code.

Accepting Multiple Body Types

There's one last thing: when we added multipart support, we broke our old JSON version of `createEmailRoute()`. Try running your old `POST /emails` request in Insomnia with a JSON body. The request hangs, and there's a stack trace in the terminal:

```
(node:48981) UnhandledPromiseRejectionWarning:
  TypeError: Cannot read property 'map' of undefined
    at createEmailRoute (routes/emails.js:20:31)
```

It looks like `req.files` is `undefined` instead of an empty array, what's going on?

`body-parser` and `multer` can play nicely with each other because they only run when the incoming request body has a particular `Content-Type` header. Since the `req.files` property is specific to `multer`, it will be `undefined` when the request body is JSON-formatted.

That's a simple fix: use the `||` operator to swap in an empty array just in case:

routes/emails.js

```
[···]

let createEmailRoute = async (req, res) => {
-   let attachments = req.files.map(file => file.filename);
+   let attachments = (req.files || []).map(file => file.filename);
    [···]
};

[···]
```

While we're at it, let's also send the full URL to each attachment instead of just the filename:

routes/emails.js

```
[···]

let createEmailRoute = async (req, res) => {
-   let attachments = (req.files || []).map(file => file.filename);
+   let attachments = (req.files || []).map(file =>
+     '/uploads/' + file.filename
+   );
    [···]
};

[···]
```

Test out the JSON and multipart versions of the `POST /emails` request with Insomnia. Both should work, and the response should always include an `"attachments"` property!

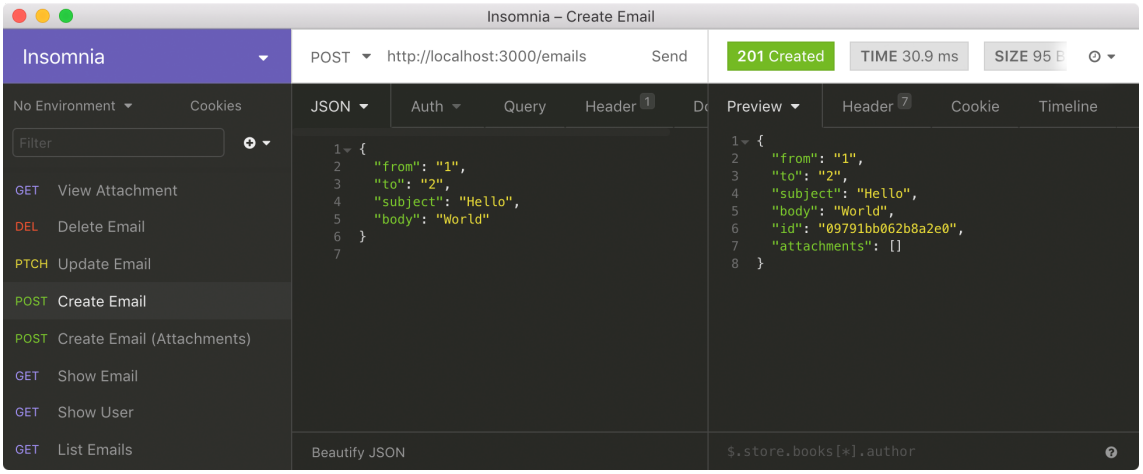


Figure 6.9: *`POST /emails` works with JSON or multipart request bodies.*

Good middleware doesn't make too many assumptions about what other middleware is or is not doing. With just a few lines of code, we added logging, JSON body support, compression, a frontend and file upload support! That's the hallmark of middleware: powerful, reusable behaviors that can be layered without conflicts.

Go Further

Third-party middleware dramatically expands what backends can do with just a few lines of code. These challenges would have been overwhelming last chapter, but now they're just a small stretch!

URL Encoded Bodies

JSON and multipart forms are pretty awesome, but many clients are used to submitting URL-encoded bodies like this:

```
POST /emails HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 36

from=1&to=2&subject=Hello&body=World
```

URL-encoded forms are commonly included as a body or at the end of the URL. Like multipart forms, they can encode a simple dictionary.

In Insomnia, duplicate your `POST /emails` request and change the body type to "Form URL Encoded". In the Timeline panel, you can see that Insomnia sets the `Content-Type` header and generates an ampersand-separated list of key-value pairs.

The `createEmailRoute()` and `updateEmailRoute()` functions don't understand URL-encoded forms. Add support with the `body-parser` module. Remember, routes can have more than one middleware function!

PATCH Things Up

The `PATCH /emails/<id>` route for updating existing emails doesn't accept multipart forms like `POST /emails`. Add support and pay close attention to edge cases: how do you want to handle older attachments that were previously uploaded?

MIME Types

When attachments are uploaded, the filenames are replaced with random ones, and the file extension is dropped for security reasons. Unfortunately, that means when someone tries to load an attachment in the browser, Express doesn't know what to set the `Content-Type` response header to. Consequently, the browser downloads the attachment rather than displaying it as, for example, an image or PDF.

There are a couple approaches to fix this:

1. The quick fix is to preserve the file extension: when a client requests an attachment, Express will infer the MIME type from the file extension. However, this is generally considered a dangerous practice.
2. The better fix is to adjust `createEmailRoute()` to track not just the filename, but also the file's `.mimetype` property. When a client requests an email attachment, `serveStatic()` should search the list of emails for the attachment to find its corresponding MIME type and add a `Content-Type` header.

   To inject this behavior, take a look at the `setHeaders()` option for `serve-static`.

Part III

Authentication & Authorization

Chapter 7

Basic Authentication

Right now, Pony Express allows anyone to send and view emails. How can we lock things down so that only registered users can access Pony Express? To identify which user is making the request, the backend needs to support user authentication.

Authentication and authorization are two inherently separate concerns that often get tangled together. **Authentication** deals with verifying that a user is who they claim they are. **Authorization** specifies what that verified user is allowed to do. Strictly separating authentication and authorization will pave the way for effortless extension, like supporting a new authentication method. But a small design shortcut can obscure dangerous security flaws and frustrate other developers.

In the next few chapters, we won't cover any particular Node packages, but will instead focus on good boundaries and design patterns that lay the groundwork for resilient authentication and authorization code, regardless of your tooling choices.

Authorization Header

Let's start with the simplest form of user authentication: submitting a username and password. Despite the misleading name, the conventional way to send credentials with a request is to include an `Authorization` header.

Here are some examples of valid `Authorization` header formats:

```
Authorization: Basic bnliYmxyOmFscHM=
Authorization: Bearer eyJhbGci.eyJ1c2V.CTjnJfQk
```

The `Authorization` header begins with a word indicating which authentication strategy to use. The strategy is not optional, although many production codebases abuse the standard by omitting the strategy. There are a handful of standardized authentication methods available: we'll start with "Basic", which is designed for username and password credentials, and in the next chapter we'll tackle "Bearer".

Everything after the strategy type "Basic" represents the username and password: `bnliYmxyOmFscHM=`. The credentials are formatted as `username:password`, then converted from ASCII to Base 64. This example represents the string `nybblr:alps`, where `nybblr` is the username and `alps` is the password.

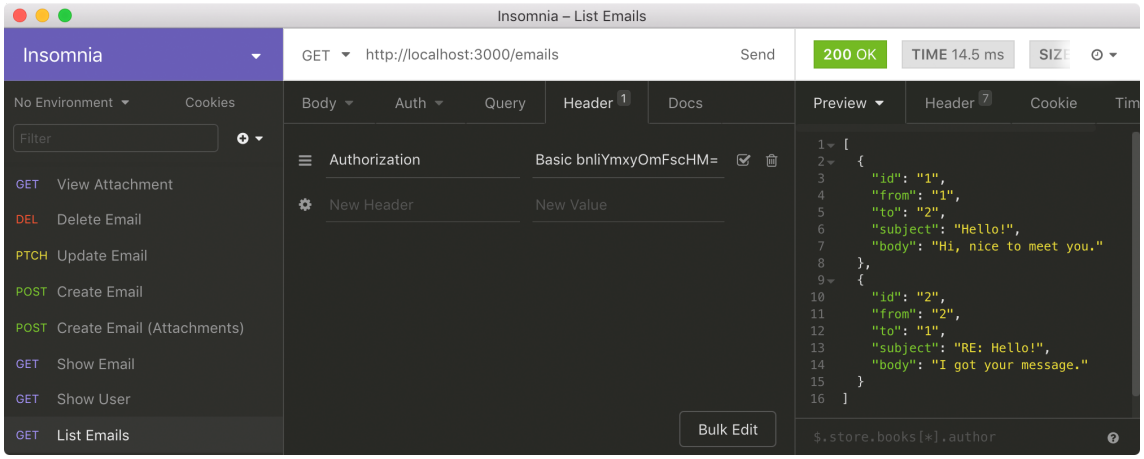


Figure 7.1: *Authenticating with the `Authorization` header.*

In Insomnia, add an `Authorization` header to your `GET /emails` request and set the value to the following:

```
Basic bnliYmxyOmFscHM=
```

Make sure you are adding a request header, not using Insomnia's "Auth" tab. If you want to authenticate as a different user, you can use Node's `Buffer()` constructor to convert a string to Base 64:

```
Buffer.from('nybblr:alps').toString('base64');
> "bnliYmxyOmFscHM="
Buffer.from('flurry:redwood').toString('base64');
> "Zmx1cnJ5OnJlZHdvb2Q="
```

Doing that Base 64 logic is a bit cryptic, so let's delete the handwritten `Authorization` header and select "Basic Auth" in Insomnia's "Auth" tab. Here, simply type out the username and password, and Insomnia will automatically generate the same Base 64 encoded `Authorization` header.

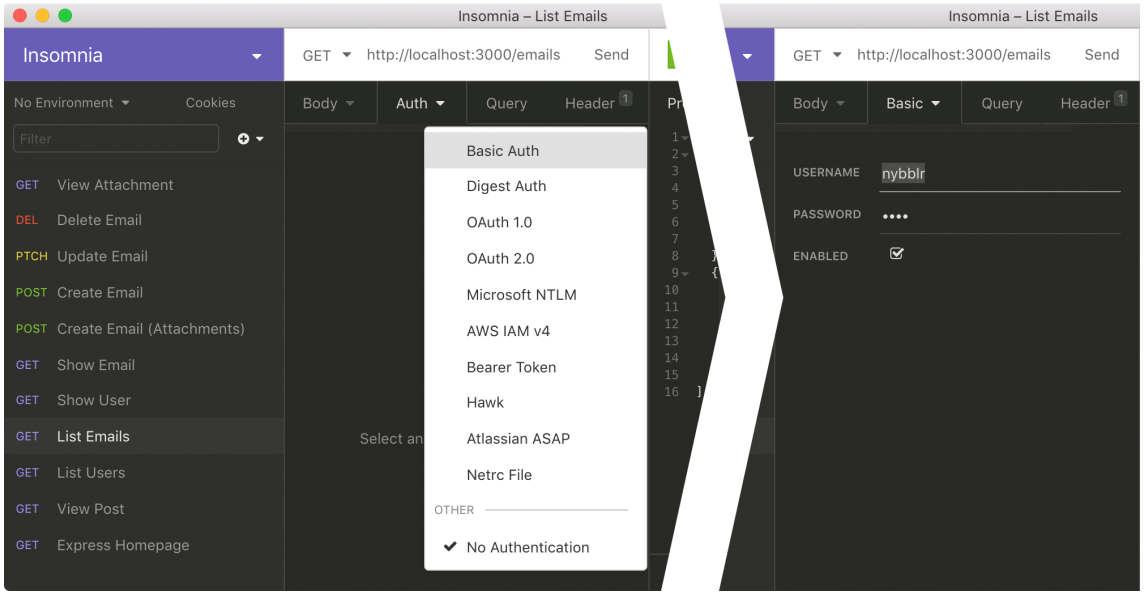


Figure 7.2: *Insomnia's Auth tab supports several authentication methods.*

To the backend, nothing changed — Insomnia's "Auth" tab is just a convenient way to generate the `Authorization` header.

Handling Authentication with Middleware

Now that we're including credentials with the request, we need to add support to the backend. Since most requests should require authentication, middleware is a great fit. We'll start writing our own middleware in a new file, `lib/basic-auth.js`:

lib/basic-auth.js

```
+ let basicAuth = (req, res, next) => {
+   let header = req.headers.authorization || '';
+   let [type, payload] = header.split(' ');
+
+   console.log(type, payload);
+
+   next();
+ };
+
+ module.exports = basicAuth;
```

Add it to the global middleware stack in `index.js`:

index.js

```
  [...]
  const compress = require('compression');
  const serveStatic = require('serve-static');
+ const basicAuth = require('./lib/basic-auth');

  [...]
  app.use(serveStatic(path.join(__dirname, 'public')));
  app.use('/uploads', serveStatic(path.join(__dirname, 'uploads')));
+ app.use(basicAuth);
  app.use('/users', usersRouter);
  app.use('/emails', emailsRouter);

  [...]
```

The placement is important: the user should be identified from their credentials before the `/emails` or `/users` routes run, but authentication isn't necessary for serving frontend files or email attachments. Arguably, `/uploads` could use authentication — we will leave that challenge to you!

We added `basicAuth()` to the global middleware stack. That's usually not a great choice, but most backends should default to requiring authentication for all routes. The downside is that we'll need to be especially careful when we implement `basicAuth()` to make sure it behaves well with other middleware.

Try running an authenticated `GET /emails` request with Insomnia. In the terminal, you should see `Basic` and `bnliYmxyOmFscHM=` logged.

Now is a good time to check the authentication strategy: if the type is "Basic", `basicAuth()` should come to life. Otherwise, it should quietly ignore the header and move on to the next middleware:

lib/basic-auth.js

```
let basicAuth = (req, res, next) => {
  let header = req.headers.authorization || '';
  let [type, payload] = header.split(' ');

-   console.log(type, payload);
+   if (type === 'Basic') {
+     let credentials = Buffer.from(payload, 'base64').toString('ascii');
+     let [username, password] = credentials.split(':');
+     console.log(username, password);
+   }

  next();
};

[···]
```

Retry an authenticated `GET /emails` request; the username and password should be logged in the terminal.

Middleware shouldn't be tightly coupled to its position in the middleware stack, nor should it be too eager to blow up. `basicAuth()` should only blow up if the request opts-in to "Basic" authentication, but the credentials are wrong. To determine if the credentials are correct, we'll search the array of users by username and password:

lib/basic-auth.js

```
+ const users = require('../fixtures/users');
+
+ let findUserByCredentials = ({ username, password }) =>
+   users.find(user => user.username === username
+                   && user.password === password);

  let basicAuth = (req, res, next) => {
    [···]

    if (type === 'Basic') {
      let credentials = Buffer.from(payload, 'base64').toString('ascii');
      let [username, password] = credentials.split(':');
-     console.log(username, password);

+     let user = findUserByCredentials({ username, password });
+     console.log(user);
    }

    next();
  };

  [···]
```

Graceful Global Middleware

There are many ways authentication could go wrong, so let's tabulate a few scenarios:

1. The user doesn't include an `Authorization` header at all. In this case, `basicAuth()` should do nothing and continue to the next middleware.
2. The user includes an `Authorization` header, but the type isn't `Basic`. Again, `basicAuth()` should do nothing and continue to the next middleware.
3. The user includes an `Authorization` header for `Basic`, and the credentials are correct. `basicAuth()` should continue to the next middleware, but first store information about the authenticated user on the request object for subsequent middleware.
4. The user includes an `Authorization` header for `Basic`, but the credentials are wrong. This time, `basicAuth()` should halt the request, respond with a `401 Unauthorized` status code, and it should *not* continue to the next middleware.

To handle the last scenario which should *not* advance to the next middleware, let's use an early return to short circuit `basicAuth()` before it invokes `next()`:

lib/basic-auth.js

```
[···]

let basicAuth = (req, res, next) => {
  [···]

  if (type === 'Basic') {
    [···]
    let user = findUserByCredentials({ username, password });
-   console.log(user);

+   if (user) {
+     req.user = user;
+   } else {
+     res.sendStatus(401);
+     return;
+   }
  }

  next();
};

[···]
```

The most common symptom of global middleware is nested `if...else` statements. It's worth the added complexity in this particular instance, but branch complexity is one of the most compelling reasons to keep global middleware to a minimum.

Try replicating each of the four scenarios from Insomnia. They should all be working! Bad credentials should trigger a `401 Unauthorized` status code:

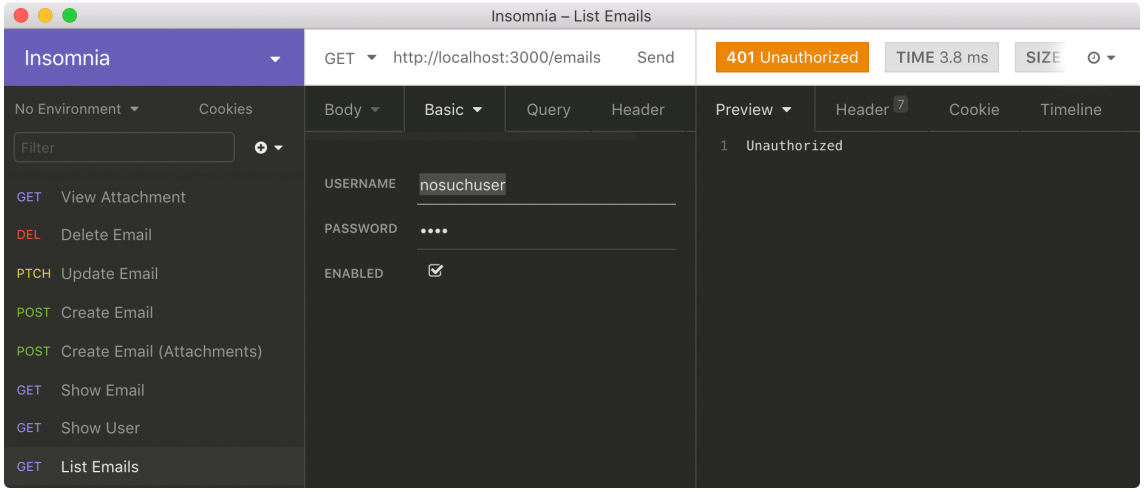


Figure 7.3: *No emails for you.*

Requiring Authentication

Well, except for one thing: if you remove the `Authorization` header, the backend still responds to any request. Shouldn't credentials be mandatory? What's the point of supporting authentication if it's optional?

We could make `basicAuth()` a bit more opinionated and let it blow up if credentials aren't included with the request, but this leads to brittle middleware. `basicAuth()` has one responsibility: to *verify* "Basic" credentials. It shouldn't also have the responsibility of blowing up if *none are included*!

Instead, let's delegate that responsibility to a new middleware. Create a new file, `lib/require-auth.js`:

lib/require-auth.js

```
+ let requireAuth = (req, res, next) => {
+   if (req.user) {
+     next();
+   } else {
+     res.sendStatus(401);
+   }
+ };
+
+ module.exports = requireAuth;
```

`requireAuth()` is dead simple: if the request looks authenticated because the `req.user` property has been set, continue to the next middleware. If the request appears unauthenticated, respond with `401 Unauthorized` and don't invoke the next middleware.

Where should we add `requireAuth()`? There are a couple options, but for now let's require authentication on a router-by-router basis. Add `requireAuth()` to `routes/users.js`:

routes/users.js

```
  const express = require('express');
+ const requireAuth = require('../lib/require-auth');
  const users = require('../fixtures/users');

  [···]

  let usersRouter = express.Router();

+ usersRouter.use(requireAuth);

  usersRouter.get('/', getUsersRoute);
  usersRouter.get('/:id', getUserRoute);

  [···]
```

Surprise! A router generated by `express.Router()` has its own middleware stack that runs before any of its routes. Global middleware is generally bad, but repeating middleware per route is tedious, so router-level middleware often hits the sweet spot.

Let's also add `requireAuth()` to `routes/emails.js`:

routes/emails.js

```
  [···]
  const multer = require('multer');
+ const requireAuth = require('../lib/require-auth');
  const generateId = require('../lib/generate-id');
  const emails = require('../fixtures/emails');

  [···]

  let emailsRouter = express.Router();

+ emailsRouter.use(requireAuth);

  emailsRouter.route('/')
    [···]
```

Now all our `/users` and `/emails` routes are protected: a user must submit valid credentials to get a response from the server. Try a few Insomnia requests with authentication disabled:

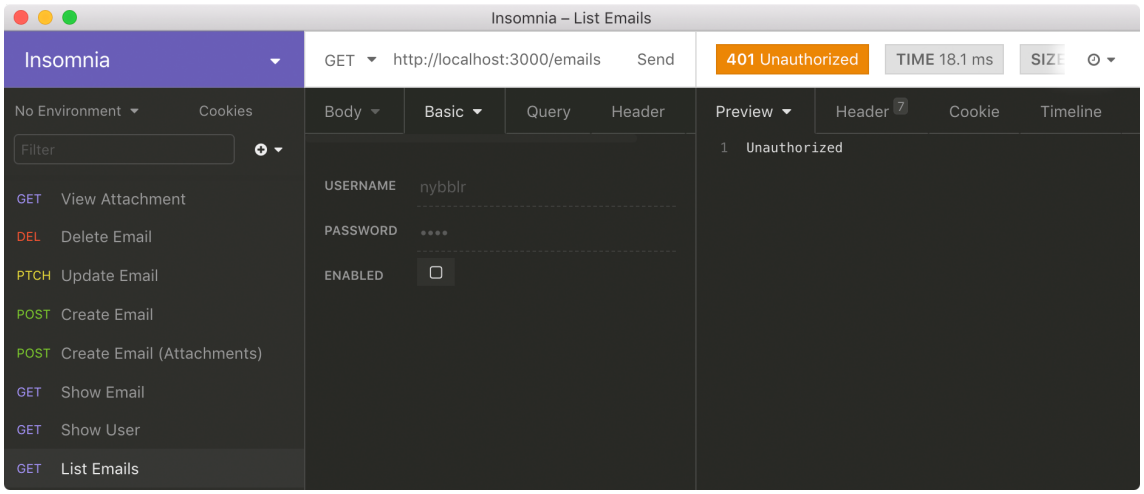


Figure 7.4: *No credentials, no response.*

Splitting authentication into `basicAuth()` and `requireAuth()` might seem unnecessarily formal, but it sets the stage to effortlessly support other authentication strategies. Moreover, the backend now supports authenticated routes and guest routes: while some routes always require a credentialed user, other routes can be accessed without credentials as a guest.

Creating a Middleware Factory

Basic authentication is feature complete, but there's one last design problem to tackle: `basicAuth()` is hardwired to look up a user by credentials. That lookup logic is specific to our backend, but good middleware is often easy to reuse in other projects. How could we remove `findUserByCredentials()` from this module so `basicAuth()` can be used in other backend codebases?

The implementation of `findUserByCredentials()` is essentially a configuration detail of `basicAuth()`. Middleware factories like `serveStatic()` and `morgan()` supported configuration, so what if we turned `basicAuth()` into our own middleware factory to inject an implementation for `findUserByCredentials()`?

Arrow functions in JavaScript make this style of functional programming a cinch. We just need to modify the function signature in `lib/basic-auth.js`:

```
lib/basic-auth.js

  [···]

- let basicAuth = (req, res, next) => {
+ let basicAuth = (findUserByCredentials) => (req, res, next) => {
    [···]
  };

  [···]
```

If you're new to functional programming or arrow functions with implicit returns, this might look a typo. We made a function `(findUserByCredentials) =>` that returns our `basicAuth()` middleware function. But this time, the rules of variable scoping allow `basicAuth()` to *memorize* the `findUserByCredentials` argument.

The `basicAuth()` middleware is essentially gathering arguments from two invocations instead of one! This pattern — gathering arguments over multiple invocations — is called "currying."

Currying and Middleware Factories

Currying can often be disguised by syntax, but with arrow functions and implicit returns it's pretty easy to spot:

```
// Typical function that
// accepts 3 arguments at once
let add = (a, b, c) => {
  return a + b + c;
};

add(1, 2, 3);
> 6

// Curried function that
// accepts 1 argument at a time
let add = (a) => (b) => (c) => {
  return a + b + c;
};

add(1)(2)(3);
> 6
```

What's the point of requiring multiple invocations when it can be done with one? Thanks to the rules of variable scoping, we can stop at any invocation and save the resulting function — a **partially applied function** — to a variable so it can be reused.

```
let addOne = add(1);
let addOneTwo = addOne(2);

add(1)(2)(3);
> 6
addOne(2)(3);
> 6
addOneTwo(3);
> 6
```

Currying is an elegant way to generate functions that must have a particular signature, such as `(req, res, next) =>`, but need some extra configuration first. Strictly speaking, currying refers to accepting one argument at a time, but in JavaScript it's more common to leave arguments grouped together.

Now that `basicAuth()` is a middleware factory and expects to receive the `findUserByCredentials()` function as an argument, let's move it into a new file called `lib/find-user.js`:

lib/find-user.js

```
+ const users = require('../fixtures/users');
+
+ let findUserByCredentials = ({ username, password }) =>
+   users.find(user => user.username === username
+                   && user.password === password);
+
+ exports.byCredentials = findUserByCredentials;
```

Don't forget to delete the code you just moved from `lib/basic-auth.js`:

lib/basic-auth.js

```
- const users = require('../fixtures/users');
-
- let findUserByCredentials = ({ username, password }) =>
-   users.find(user => user.username === username
-                   && user.password === password);

  let basicAuth = (findUserByCredentials) => (req, res, next) => {
    [···]
```

In `index.js`, we'll pass `findUserByCredentials()` to `basicAuth()`:

index.js

```
  [···]
  const serveStatic = require('serve-static');
  const basicAuth = require('./lib/basic-auth');
+ const findUser = require('./lib/find-user');

  [···]

  app.use('/uploads', serveStatic(path.join(__dirname, 'uploads')));
- app.use(basicAuth);
+ app.use(basicAuth(findUser.byCredentials));
  [···]
```

Test a few requests in Insomnia one last time to ensure you didn't break anything.

That subtle change comes with huge design wins! Middleware factories elegantly decouple backend code with almost no consequences for API simplicity. For a complex feature like authentication, we need all the wins we can get.

Overall, adding authentication to our backend wasn't bad at all because:

1. We used *stateless* authentication instead of introducing cookies and sessions.
2. We added authentication support using middleware.
3. We kept `basicAuth()` as chill as possible and left enforcement to `requireAuth()`.
4. We decoupled user lookup logic by turning `basicAuth()` into a middleware factory.

With the right design patterns, complex behaviors arise from straightforward code.

Go Further

Hashing Passwords

For simplicity's sake, we are storing user passwords as plain text in `fixtures/users.json`, but that would be catastrophically unsafe in a production app. Instead, we should hash the passwords with a library like bcrypt and store the hashed password instead:

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

/* Tune how long it takes to hash password.
   The longer, the more secure. */
const saltRounds = 10;

/* Generate a hashed version of the password.
   This is what should be stored. */
let hashed = await bcrypt.hash('alps', saltRounds);
```

The hashed password looks like gobbledygook, but most importantly it can't be reversed to determine the user's original password.

```
> console.log(hashed)

"$2b$10$n77SvPvP/o4eU21J4.cQLfbcqM"
```

You'll need to replace the passwords in `fixtures/users.json` with hashed versions.

When a user tries to authenticate with their password, use the `bcrypt.compare()` method to compare it with the hashed password that's stored in `fixtures/users.json`:

```
/* Compare submitted password with the
   hashed password that's stored. */
if (await bcrypt.compare('alps', hashed)) {
  console.log('Password is correct!');
} else {
  console.log('Wrong password.');
}
```

If you make these changes in the right place, you'll be able to follow the next chapter as-is. Otherwise, you'll end up duplicating the hashing logic. Choose wisely!

Chapter 8

Authentication with JSON Web Tokens

Sending credentials with each HTTP request is straightforward, but as backends grow, the surface area for security vulnerabilities also grows. By sending credentials with every request, an attacker has plenty of opportunities to compromise user credentials.

Security isn't the only downside to sending credentials with each request — it also handicaps architecture and scaling options. Here are a few examples:

- Every endpoint or server must first verify credentials for each request. That can quickly become a performance bottleneck: password hashing algorithms like bcrypt are secure because they are *designed* to be time consuming — ¼ to one second. That will noticeably delay every request.
- It's harder to scale backend services across separate servers because each server must support user authentication, creating a central bottleneck.
- The backend can't easily track which devices have used the account or allow the user to audit and revoke access without resetting their password. Likewise, it's difficult to grant restricted access to certain devices.
- It's difficult to provide alternative authentication methods — such as Single Sign On (SSO) services — without drastically changing how clients interact with the API.

None of these are deal breakers early on, and premature optimization is a dangerous trap. But luckily there's an easy way to delay these pain points *and* simplify backend authentication!

Proof of Verification

To cross country borders at an airport, you must prove your identity and citizenship. One way to do that would be to carry your birth certificate and government-issued ID with you at all times. However, it is time consuming to verify these documents, and border control would need access to your home country's citizen database, plus the expertise to verify those documents.

Instead, you present a passport at the border to prove your identity and citizenship. A passport has security features that make it difficult to tamper with and relatively fast to verify.

The actual documents still need to be verified, but only when you pick up the passport. That process could take weeks to months, but it only needs to happen every ten years. If your passport is stolen, it can be invalidated without compromising your birth certificate and government-issued ID.

In other words, a passport is *proof* that your documents were verified.

JSON Web Tokens

Like a passport, a **JSON Web Token** (JWT, or simply "token" in this chapter) is a tamper-resistant document that proves you have verified your identity using credentials like a username and password. To make authenticated HTTP requests, a client submits their username and password once to be issued a JWT. On all subsequent HTTP requests, the client includes the JWT instead of credentials.

To support token authentication, we need to build two pieces:

1. A "passport office" where a client exchanges credentials for a JWT.
2. Middleware like `basicAuth()` that checks for a valid JWT with every HTTP request.

Issuing Tokens

First we'll create a "passport office" at `POST /tokens`. Create a new router in `routes/tokens.js`:

routes/tokens.js

```
+ const express = require('express');
+ const bodyParser = require('body-parser');
+ const findUser = require('../lib/find-user');
+
+ let createTokenRoute = (req, res) => {
+   let credentials = req.body;
+   let user = findUser.byCredentials(credentials);
+   console.log(user);
+ };
+
+ let tokensRouter = express.Router();
+
+ tokensRouter.post('/', bodyParser.json(), createTokenRoute);
+
+ module.exports = tokensRouter;
```

Mount the `tokensRouter` in `index.js`:

index.js

```
  [···]

+ const tokensRouter = require('./routes/tokens');
  const usersRouter = require('./routes/users');
  const emailsRouter = require('./routes/emails');

  [···]
  app.use('/uploads', serveStatic(path.join(__dirname, 'uploads')));
+ app.use('/tokens', tokensRouter);
  app.use(basicAuth(findUser.byCredentials));
  [···]
```

Create a new Insomnia request to `POST /tokens` and include a JSON-formatted request body with a username and password:

```
{
  "username": "nybblr",
  "password": "alps"
}
```

The request will hang, but if the credentials are correct, the terminal should print out the matching user. So once a user proves their identity, what should the backend respond with? A newly created token! Add an `if...else` statement to `routes/tokens.js`:

routes/tokens.js

```
  [...]

  let createTokenRoute = (req, res) => {
    let credentials = req.body;
    let user = findUser.byCredentials(credentials);
-   console.log(user);

+   if (user) {
+     let token = 'I am user ' + user.id;
+     res.status(201);
+     res.send(token);
+   } else {
+     res.sendStatus(422);
+   }
  };

  [...]
```

If the credentials are incorrect, it's conventional to reply with `422 Unprocessable Entity`. If they are correct, we use the `201 Created` status code. But what does a token actually look like? A token can be anything that identifies the user, such as the string `"I am user 1"` or an object like `{ userId: "1" }`. For a token to be useful from a security perspective, it must be difficult to forge.

Since Pony Express validated the credentials, it should be the only party that can issue genuine tokens. In other words, Pony Express needs to *digitally sign* the tokens it issues. When a client presents this token on a future HTTP request, the backend can quickly tell if the token is authentic.

Signing Tokens

To generate a cryptographically signed token — one that can't be counterfeited — we'll use the jsonwebtoken Node package:

```
$ npm install jsonwebtoken
```

This library takes care of the tricky security details behind JSON Web Tokens. To issue a new token, use the `jwt.sign()` method:

routes/tokens.js

```
  const express = require('express');
  const bodyParser = require('body-parser');
+ const jwt = require('jsonwebtoken');
  const findUser = require('../lib/find-user');

+ const signature = '1m_s3cure';
+
+ let createToken = (user) =>
+   jwt.sign(
+     { userId: user.id },
+     signature,
+     { expiresIn: '7d' }
+   );

  let createTokenRoute = (req, res) => {
    [···]
    if (user) {
-     let token = 'I am user ' + user.id;
+     let token = createToken(user);
      res.status(201);
      res.send(token);
    } else {
      res.sendStatus(422);
    }
  };

  [···]
```

`jwt.sign()` takes three arguments:

1. `payload`: A plain ol' JavaScript object with identifying information about the user. The `payload` is like the picture page of a passport: it should have enough details to look up the user's full profile, but not so much that the token gets long. After all,

the token will need to be sent with every single HTTP request. Most of the time, the user's ID is enough.
2. `signature`: A secret key that only the backend should know. This is the backend's signature for signing all new tokens, so if the signature is compromised, JWTs can be forged!
3. `options`: An object of configuration options, such as how long the token is considered valid. We specified that the token should expire in seven days. After seven days, the user will need to request a new JSON Web Token by resubmitting their username and password.

Dissecting a Token

Send a request to `POST /tokens` with Insomnia. The response should look like a long string of random characters with a couple periods.

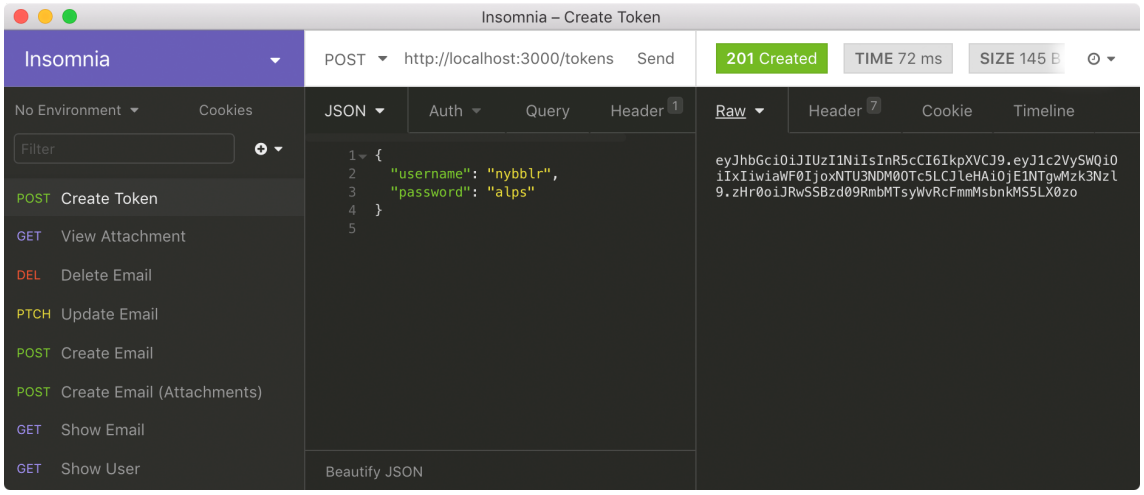


Figure 8.1: *The department of tokens is open.*

The string isn't complete gobbledygook. Like the arguments to `jwt.sign()`, the string has three sections separated by periods:

1. The first section is a Base 64 string of the algorithm used to generate the token. With the default algorithm options, the first section encodes a JSON string like this:

```
> atob( token.split('.')[0] )

  {"alg":"HS256","typ":"JWT"}
```

2. The middle section is a Base 64 string of the payload — the first argument to `jwt.sign()` — along with an issue time and expiration time. The decoded string looks like this:

```
> atob( token.split('.')[1] )

{"userId":"1","iat":1554742821,"exp":1555347621}
```

3. The last section is a cryptographic signature that proves the first two sections haven't been tampered with. Only the backend can generate an authentic digital signature since it knows the secret signing key.

Luckily, you don't have to write the code to generate or parse the token. The client simply presents it with every subsequent HTTP request.

Accepting JSON Web Tokens

Now that the "passport office" is up and running, the backend needs to support tokens as an alternative authentication method.

Rather than include an `Authorization` header with type "Basic" on each request, the client will use the "Bearer" method. In Insomnia, duplicate your `GET /emails` request and change the Auth type from "Basic Auth" to "Bearer Token". Copy-paste a freshly generated JWT from your `POST /tokens` request into the "token" field.

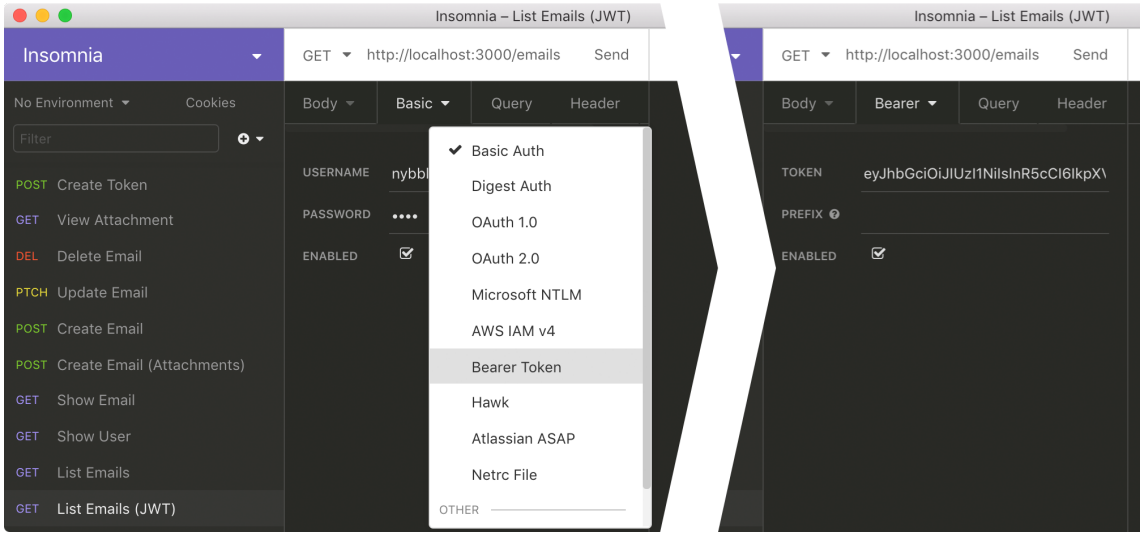


Figure 8.2: *Authenticating with a JSON Web Token.*

Try sending the request. In the Timeline tab, we see that Insomnia automatically adds an `Authorization` header to the request.

We're still getting a `401 Unauthorized` status code because the backend doesn't support "Bearer" authentication, but it's not hard to add. Token authentication will be structurally identical to `basicAuth()`. Let's create our own `tokenAuth()` middleware in `lib/token-auth.js`:

lib/token-auth.js

```
+ const jwt = require('jsonwebtoken');
+
+ const signature = '1m_s3cure';
+
+ let tokenAuth = (req, res, next) => {
+   let header = req.headers.authorization || '';
+   let [type, token] = header.split(' ');
+
+   if (type === 'Bearer') {
+     let payload = jwt.verify(token, signature);
+
+     console.log(payload);
+   }
+
+   next();
+ };
+
+ module.exports = tokenAuth;
```

Mount `tokenAuth()` just below the `tokensRouter` in `index.js`:

index.js

```
  [···]
  const basicAuth = require('./lib/basic-auth');
+ const tokenAuth = require('./lib/token-auth');
  const findUser = require('./lib/find-user');

  [···]
  app.use('/tokens', tokensRouter);
+ app.use(tokenAuth);
  app.use(basicAuth(findUser.byCredentials));
  [···]
```

Try sending a JWT-authenticated request to `GET /emails`. The backend should log an object with a `userId` property:

```
{ userId: '1', iat: 1557434979, exp: 1558039779 }
GET /emails 401 12 - 9.393 ms
```

That's the payload of your JWT — now Pony Express knows which user sent the request!

How do we know this token is authentic and was issued by Pony Express? If the token is tampered with, `jwt.verify()` will throw an exception. Otherwise, it will decode the Base 64 encoding on the payload, parse it as JSON and return it.

As with `basicAuth()`, we should look up the user's details and store them in `req.user` for the routes. To keep our middleware reusable, let's pass the payload object to a function called `findUserByToken()` that searches by the `userId` property:

lib/token-auth.js

```
  const jwt = require('jsonwebtoken');
+ const users = require('../fixtures/users');

  const signature = '1m_s3cure';

+ let findUserByToken = ({ userId }) =>
+   users.find(user => user.id === userId);

  let tokenAuth = (req, res, next) => {
    [···]

    if (type === 'Bearer') {
      let payload = jwt.verify(token, signature);

-     console.log(payload);

+     let user = findUserByToken(payload);
+     if (user) {
+       req.user = user;
+     } else {
+       res.sendStatus(401);
+       return;
+     }
    }

    next();
  };

  [···]
```

If there is a matching user, `tokenAuth()` will store the user in `req.user` to signal to `requireAuth()` that the user successfully authenticated. Try your JWT-authenticated `GET /emails` request — your old "Basic" authentication should still work too!

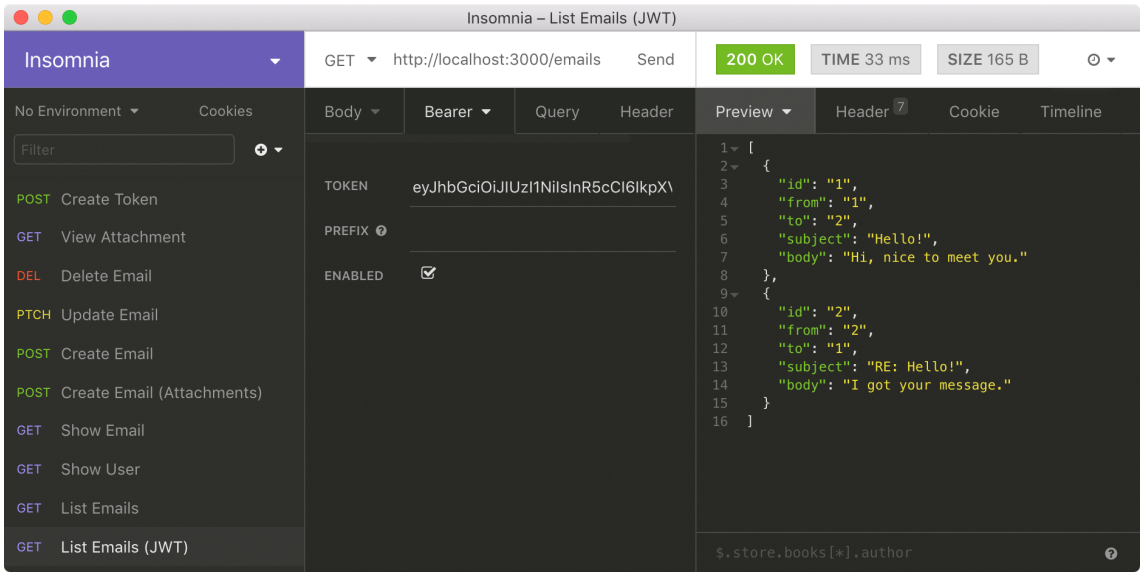


Figure 8.3: *The backend supports Bearer authentication with JWTs.*

Dealing with Invalid Tokens

There's one edge case we need to handle. In the "Auth" tab for your `GET /emails` request, try tampering with the token string — for example, delete a character from the end of the token and retry the request.

The backend should spit out a long stack trace in the terminal, such as a `JsonWebTokenError`:

```
GET /emails 500 1553 - 5.903 ms
JsonWebTokenError: invalid signature
    at node_modules/jsonwebtoken/verify.js:133:19
```

That's mostly good: the `jsonwebtoken` module noticed that the token was tampered with. However, the server shouldn't crash with `500 Internal Server Error` — to other developers, that indicates a bug. Servers should make every effort to handle expected errors, so let's catch the error and respond with `401 Unauthorized` instead:

lib/token-auth.js

```
  [···]

  let tokenAuth = (req, res, next) => {
    [···]

    if (type === 'Bearer') {
-     let payload = jwt.verify(token, signature);
+     let payload;
+     try {
+       payload = jwt.verify(token, signature);
+     } catch(err) {
+       res.sendStatus(401);
+       return;
+     }

      let user = findUserByToken(payload);
      [···]
    }

    next();
  };

  [···]
```

Retry the `GET /emails` request with the tampered token. The server should politely respond with a `401 Unauthorized` status code instead of `500 Internal Server Error`.

Decoupling with Middleware Factories

Like `basicAuth()`, the `tokenAuth()` function is due for some refactoring. Move `findUserByToken()` into `lib/find-user.js`:

lib/find-user.js

```
  [···]

  exports.byCredentials = findUserByCredentials;

+ let findUserByToken = ({ userId }) =>
+   users.find(user => user.id === userId);

+ exports.byToken = findUserByToken;
```

Pass the `findUser.byCredentials()` function to `tokenAuth()` in `index.js`:

index.js

```
  [···]
  app.use('/tokens', tokensRouter);
- app.use(tokenAuth);
+ app.use(tokenAuth(findUser.byToken));
  app.use(basicAuth(findUser.byCredentials));
  [···]
```

Last, delete the code for `findUserByToken()` from `lib/token-auth.js` and make `tokenAuth()` a middleware factory:

lib/token-auth.js

```
  const jwt = require('jsonwebtoken');
- const users = require('../fixtures/users');

  const signature = '1m_s3cure';

- let findUserByToken = ({ userId }) =>
-   users.find(user => user.id === userId);

- let tokenAuth = (req, res, next) => {
+ let tokenAuth = (findUserByToken) => (req, res, next) => {
    [···]
  };

  [···]
```

Try a few requests from Insomnia just to make sure everything still works with both authentication methods.

The backend now seamlessly supports two authentication methods: username and password with "Basic", and JSON Web Tokens with "Bearer". Because we designed `basicAuth()` to gracefully ignore anything other than "Basic" authentication and left enforcement to `requireAuth()`, we were able to support a second authentication mechanism like `tokenAuth()` with one line of middleware. Most importantly, we didn't need to *modify* existing code and potentially introduce security regressions!

Go Further

Environment Variables

Source code is not a safe place to hide sensitive information. "Magic values" like `signature` are hardcoded into the backend's source code, so anyone who gets a copy of the code will know the `signature` and can quietly issue valid tokens to impersonate any user.

It's a better idea to extract the `signature` into an environment variable so it never appears in the source code. This variable needs to be specified every time the backend is booted:

```
$ SIGNATURE=1m_s3cure npx nodemon index.js
```

Node provides `process.env` to access these environment variables:

```
const signature = process.env.SIGNATURE;
```

During development, it can be tedious to list out environment variables just to boot up the server. The `dotenv` module lets you list out these variables in a separate file called `.env`:

.env

```
SIGNATURE=1m_s3cure
```

This `.env` file should never be committed to source code. To load it, the backend needs to run the `dotenv` module as early as possible:

index.js

```
require('dotenv').config();

/* From now on, any variables listed in .env
   are available as environment variables. */
console.log(process.env.SIGNATURE);
> "1m_s3cure"
```

Nothing else needs to change! Boot up the backend as before, without listing all the environment variables:

```
$ npx nodemon index.js
```

Find another “magic value” to extract from the source code — for example, the `expiresIn` value — and load it from an environment variable instead.

Chapter 9

Authorization Design Patterns

Pony Express supports two authentication methods, but it's still not particularly secure: any client with valid user credentials can do anything, such as deleting another user's emails.

Authentication deals with verifying that a user is who they claim to be. **Authorization** specifies what that verified user is allowed to do. Is user #1 allowed to edit an email they drafted? Can user #2 spy on an email they didn't author or receive?

Authentication is pretty straightforward to add to a backend, but defining authorization rules — who can do what — is a much more granular question that will need to evolve as the backend grows.

Adding Authorization to a Route

Modify your Insomnia request for `PATCH /emails/1` and `DELETE /emails/1` to use `Basic` authentication with credentials for user #3. A user shouldn't be able to edit an email they didn't author or delete an email not addressed to them, but both of these requests currently work.

Let's add authorization logic directly to a couple `/emails` routes in `routes/emails.js`:

routes/emails.js

```
  [···]

  let updateEmailRoute = async (req, res) => {
    let email = emails.find(email => email.id === req.params.id);
+   let user = req.user;
+   if (user.id === email.from) {
      Object.assign(email, req.body);
      res.status(200);
      res.send(email);
+   } else {
+     res.sendStatus(403);
+   }
  };

  let deleteEmailRoute = (req, res) => {
+   let email = emails.find(email => email.id === req.params.id);
+   let user = req.user;
+   if (user.id === email.to) {
      let index = emails.findIndex(email => email.id === req.params.id);
      emails.splice(index, 1);
      res.sendStatus(204);
+   } else {
+     res.sendStatus(403);
+   }
  };

  [···]
```

Retry your `PATCH /emails/1` and `DELETE /emails/1` requests with credentials for user #3. This time, both should respond with a `403 Forbidden` status code.

Authorization Design Flaws

We're done! But from a design perspective, this is horrible:

- We must duplicate this exact change — *structural* duplication — in each route.
- The authorization logic obscures the route's primary responsibility.
- The extra flow control is easy to mess up in routes with early returns.
- The response logic is duplicated, but should be consistent throughout the app.
- It's difficult to unit test the route or access rules since they are tangled with each other.

Our goal is to eliminate this structural duplication while making the code readable. It's only fair to warn you that the in-between steps won't be pretty, so roll up your sleeves and get ready for some dirt!

Extracting Authorization to Middleware

A great way to extract an `if...else` statement from a route is by moving it into middleware. Let's add a dedicated middleware function for both routes whose sole responsibility is to guard the route:

routes/emails.js

```
  [···]

  let updateEmailRoute = async (req, res) => {
    let email = emails.find(email => email.id === req.params.id);
-   let user = req.user;
-   if (user.id === email.from) {
    Object.assign(email, req.body);
    res.status(200);
    res.send(email);
-   } else {
-     res.sendStatus(403);
-   }
  };

+ let authorizeUpdateEmailRoute = (req, res, next) => {
+   let email = emails.find(email => email.id === req.params.id);
+   let user = req.user;
+   if (user.id === email.from) {
+     next();
+   } else {
+     res.sendStatus(403);
+   }
+ };

  [···]
```

Do the same for `deleteEmailRoute()`:

routes/emails.js

```
  [···]

  let deleteEmailRoute = (req, res) => {
-   let email = emails.find(email => email.id === req.params.id);
-   let user = req.user;
-   if (user.id === email.to) {
    let index = emails.findIndex(email => email.id === req.params.id);
    emails.splice(index, 1);
    res.sendStatus(204);
-   } else {
-     res.sendStatus(403);
-   }
  };

+ let authorizeDeleteEmailRoute = (req, res, next) => {
+   let email = emails.find(email => email.id === req.params.id);
+   let user = req.user;
+   if (user.id === email.to) {
+     next();
+   } else {
+     res.sendStatus(403);
+   }
+ };

  [···]
```

Mount each middleware to its respective route:

routes/emails.js

```
[···]

emailsRouter.route('/:id')
  .get(getEmailRoute)
- .patch(bodyParser.json(), updateEmailRoute)
+ .patch(
+   authorizeUpdateEmailRoute,
+   bodyParser.json(),
+   updateEmailRoute
+ )
- .delete(deleteEmailRoute)
+ .delete(
+   authorizeDeleteEmailRoute,
+   deleteEmailRoute
+ )
;

[···]
```

Our routes are short and focused again: after rolling back the changes to `updateEmailRoute()` and `deleteEmailRoute()`, we extracted the authorization logic into dedicated route middleware. But there's still duplication between `authorizeUpdateEmailRoute()` and `authorizeDeleteEmailRoute()`.

Policies and Enforcers

These two middleware functions share the same logic, except for one difference: the access rules. That's because there are two unique roles tangled together: the policy and enforcer.

- The **policy** specifies the actual authorization rules with a simple yes or no answer to the question, "can this user do that?"
- The **enforcer** makes sure that policy is respected: it either continues to the next middleware or responds with `403 Forbidden`.

Let's split `authorizeUpdateEmailRoute()` into these two roles:

routes/emails.js

```
  [···]

+ let updateEmailPolicy = (req) => {
+   let email = emails.find(email => email.id === req.params.id);
+   let user = req.user;
+   return user.id === email.from;
+ };

  let authorizeUpdateEmailRoute = (req, res, next) => {
-   let email = emails.find(email => email.id === req.params.id);
-   let user = req.user;
-   if (user.id === email.from) {
+   if (updateEmailPolicy(req)) {
      next();
    } else {
      res.sendStatus(403);
    }
  };

  [···]
```

Repeat that transformation to `authorizeDeleteEmailRoute()`:

routes/emails.js

```
  [···]

+ let deleteEmailPolicy = (req) => {
+   let email = emails.find(email => email.id === req.params.id);
+   let user = req.user;
+   return user.id === email.to;
+ };

  let authorizeDeleteEmailRoute = (req, res, next) => {
-   let email = emails.find(email => email.id === req.params.id);
-   let user = req.user;
-   if (user.id === email.to) {
+   if (deleteEmailPolicy(req)) {
      next();
    } else {
      res.sendStatus(403);
    }
  };

  [···]
```

The two `authorize*()` functions are almost identical now. The only difference is the policy they enforce. Let's turn one of them into a middleware factory called `enforce()` that can be reused with different policies:

routes/emails.js

```
  [···]

- let authorizeUpdateEmailRoute = (req, res, next) => {
-   if (updateEmailPolicy(req)) {
+ let enforce = (policy) => (req, res, next) => {
+   if (policy(req)) {
      next();
    } else {
      res.sendStatus(403);
    }
  };

  [···]

- let authorizeDeleteEmailRoute = (req, res, next) => { ... };

  [···]
```

Now we can replace the old `authorize*()` functions with `enforce()`:

routes/emails.js

```
  [···]

  emailsRouter.route('/:id')
    .get(getEmailRoute)
    .patch(
-     authorizeUpdateEmailRoute,
+     enforce(updateEmailPolicy),
      bodyParser.json(),
      updateEmailRoute
    )
    .delete(
-     authorizeDeleteEmailRoute,
+     enforce(deleteEmailPolicy),
      deleteEmailRoute
    )
  ;

  [···]
```

The `enforce()` middleware factory belongs in its own module now. Move it to a new file called `lib/enforce.js`:

lib/enforce.js

```
+ let enforce = (policy) => (req, res, next) => {
+   if (policy(req)) {
+     next();
+   } else {
+     res.sendStatus(403);
+   }
+ };
+
+ module.exports = enforce;
```

Don't forget to delete it from `routes/emails.js`:

routes/emails.js

```
  [···]
  const requireAuth = require('../lib/require-auth');
  const generateId = require('../lib/generate-id');
+ const enforce = require('../lib/enforce');
  const emails = require('../fixtures/emails');

  [···]

- let enforce = (policy) => (req, res, next) => { ... };

  [···]
```

Here's a quick recap of that refactor:

1. We added authorization logic to each route.
2. We extracted the authorization structure into dedicated middleware.
3. We split the authorization middleware into policies and enforcers.
4. We replaced both enforcers with a single middleware factory, `enforce()`.

Simplifying Policies

What about the email lookup logic we had to duplicate between `updateEmailRoute()` and `updateEmailPolicy()`? Could we simplify the policy functions so there is less room for duplication bugs?

There are several ways to tackle this, but here's one that's loosely based on a Ruby authorization library called Pundit. Imagine if each route could decide when to apply authorization logic so it could pass along the exact email to authorize:

routes/emails.js

```
[···]

let updateEmailRoute = async (req, res) => {
  let email = emails.find(email => email.id === req.params.id);
+ req.authorize(email);
  Object.assign(email, req.body);
  res.status(200);
  res.send(email);
};

[···]

let deleteEmailRoute = (req, res) => {
+ let email = emails.find(email => email.id === req.params.id);
+ req.authorize(email);
  let index = emails.findIndex(email => email.id === req.params.id);
  emails.splice(index, 1);
  res.sendStatus(204);
};

[···]
```

Let's dream a bit more. Suppose that `req.authorize()` could in turn invoke the appropriate policy and pass along the authenticated user and email. Our policies would become substantially tidier:

routes/emails.js

```
  [···]

- let updateEmailPolicy = (req) => {
-   let email = emails.find(email => email.id === req.params.id);
-   let user = req.user;
-   return user.id === email.from;
- };
+ let updateEmailPolicy = (user, email) =>
+   user.id === email.from;

  [···]

- let deleteEmailPolicy = (req) => {
-   let email = emails.find(email => email.id === req.params.id);
-   let user = req.user;
-   return user.id === email.to;
- };
+ let deleteEmailPolicy = (user, email) =>
+   user.id === email.to;

  [···]
```

Policies that are just a line long? Yes! To make `req.authorize()` a reality, the `enforce()` middleware factory can add the method to the request object in `lib/enforce.js`:

lib/enforce.js

```
let enforce = (policy) => (req, res, next) => {
-   if (policy(req)) {
-     next();
-   } else {
-     res.sendStatus(403);
-   }
+   req.authorize = (resource) => {
+     if (!policy(req.user, resource)) {
+       res.sendStatus(403);
+     }
+   };
+
+   next();
};

[···]
```

Instead of running the policy immediately, `enforce()` adds a method to `req` to defer that logic until the route decides to invoke it. By then, the route can look up the email and pass it to the policy as an argument.

Test out `PATCH /emails` and `DELETE /emails` with Insomnia. They should all work as before!

Enforcing Policies with Exceptions

Well, everything works with one exception: even though unauthorized users get a `403 Forbidden` response, the edit or deletion still happens! You can confirm this bug with a `GET /emails` request.

Even though the request seems to be denied, the route continued executing. There is even a stack trace in the terminal from the route trying to respond after `enforce()` already closed the connection. That's because `req.authorize()` responds to the request, but it doesn't prevent the rest of the route function from executing.

How could `req.authorize()` force the route to exit without a bunch of `if...else` statements? This is exactly what exceptions are for! Let's make a custom `Error` type in `lib/enforce.js`:

lib/enforce.js

```
+ class UserNotAuthorized extends Error {
+   constructor(message) {
+     super(message);
+     this.name = 'UserNotAuthorized';
+   }
+ }

  let enforce = (policy) => (req, res, next) => {
    req.authorize = (resource) => {
      if (!policy(req.user, resource)) {
        res.sendStatus(403);
+       throw new UserNotAuthorized();
      }
    };

    next();
  };

  [···]
```

Errors force the function that called `req.authorize()` to exit prematurely. It may seem strange to throw errors on purpose, but exceptions are simply a different kind of return value — one that bubbles up through functions.

Try `PATCH /emails` and `DELETE /emails` again — this time, the changes should never happen.

Sustainable Security

Our backend has come a long way over the last few chapters. Authentication and authorization are easily some of the most complex features every backend will tackle, so it's crucial to delay that complexity with opinionated design.

A completely secure backend isn't technically possible, but we can ward off many attacks by eliminating the primary source of vulnerabilities: regressions in complex code. A secure codebase is useless if new developers can't easily imitate or modify it for new functionality.

The only useful security is *sustainable* security.

Go Further

Enforce All the Things

There are many more routes that need authorization logic. Start by adding a policy for `getEmailRoute()`, then add a policy for `createEmailRoute()`. This one is trickier: a user shouldn't be able to create an email that is not from them.

Private Attachments

Now that `getEmailRoute()` is secured, viewing an attachment should follow the same access rules. Add a policy to the `/uploads` routes that only allows the recipient or author who uploaded the attachment to view it.

To `enforce()` the policy, you can insert middleware before `serveStatic()`:

```
- app.use('/uploads', serveStatic(...));
+ app.use('/uploads', enforce(emailAttachmentPolicy), serveStatic(...));
```

Index