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

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 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()`:

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**.

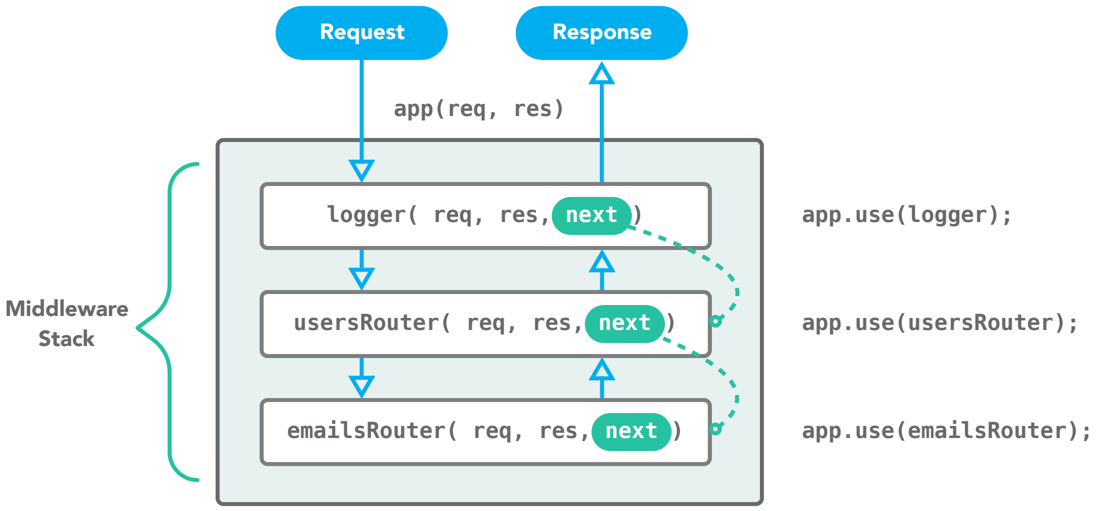
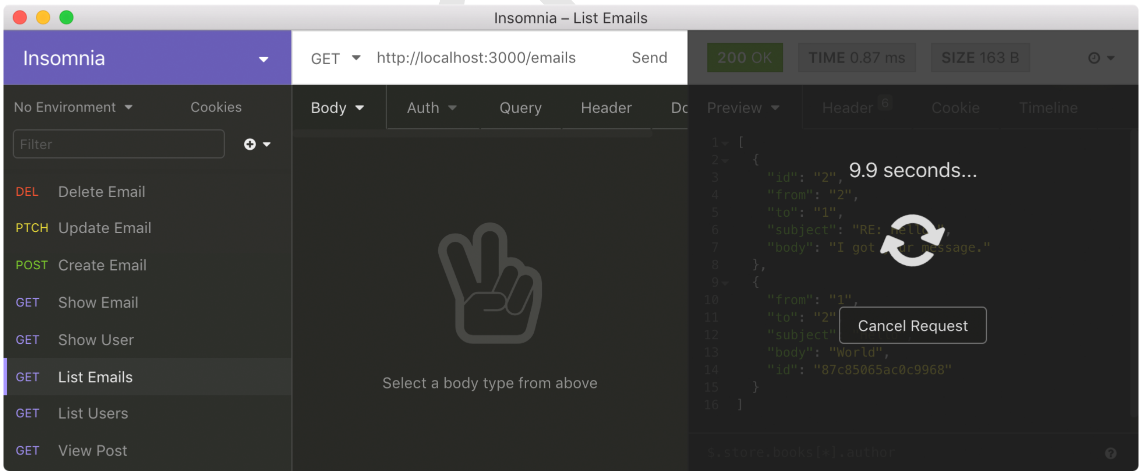


Figure 5.1: *Each middleware function in the stack gets to run before those below it.*

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:



```
[
  {
    "id": "2",
    "from": "2",
    "to": "1",
    "subject": "RE: Hello",
    "body": "I got your message."
  },
  {
    "from": "1",
    "to": "2",
    "subject": "Hello",
    "body": "World",
    "id": "87c85065ac0c9968"
  }
]
```

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()`:

Part III

Authentication & Authorization

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:

Index