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

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.