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 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**.

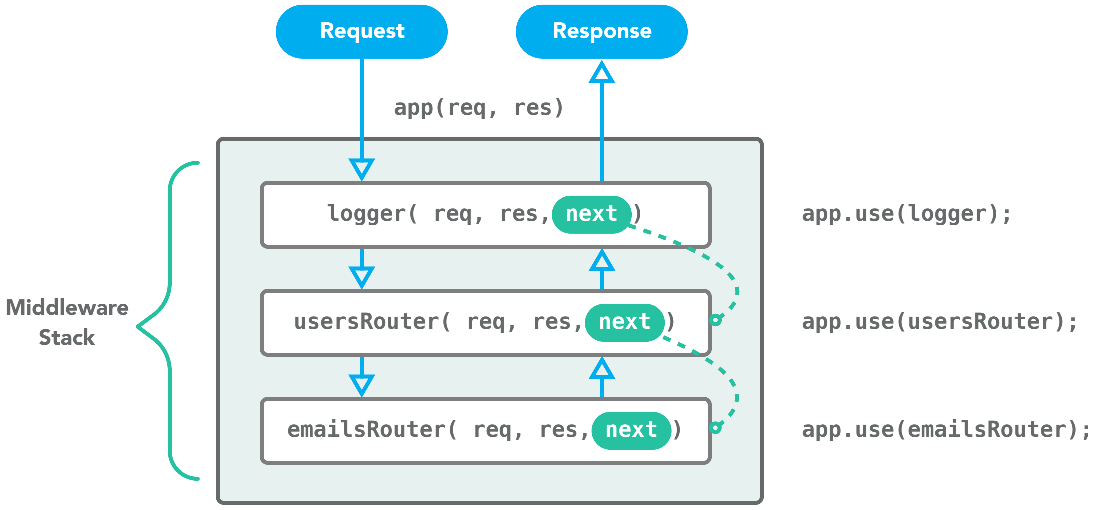
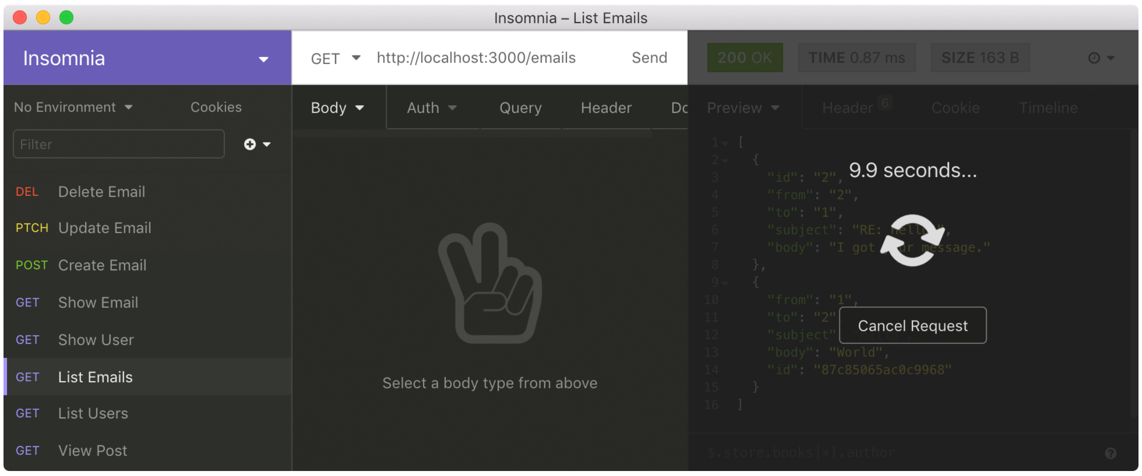


Figure 5.1: *Each middleware function in the stack gets to run before those below it.*

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()`:

```
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.

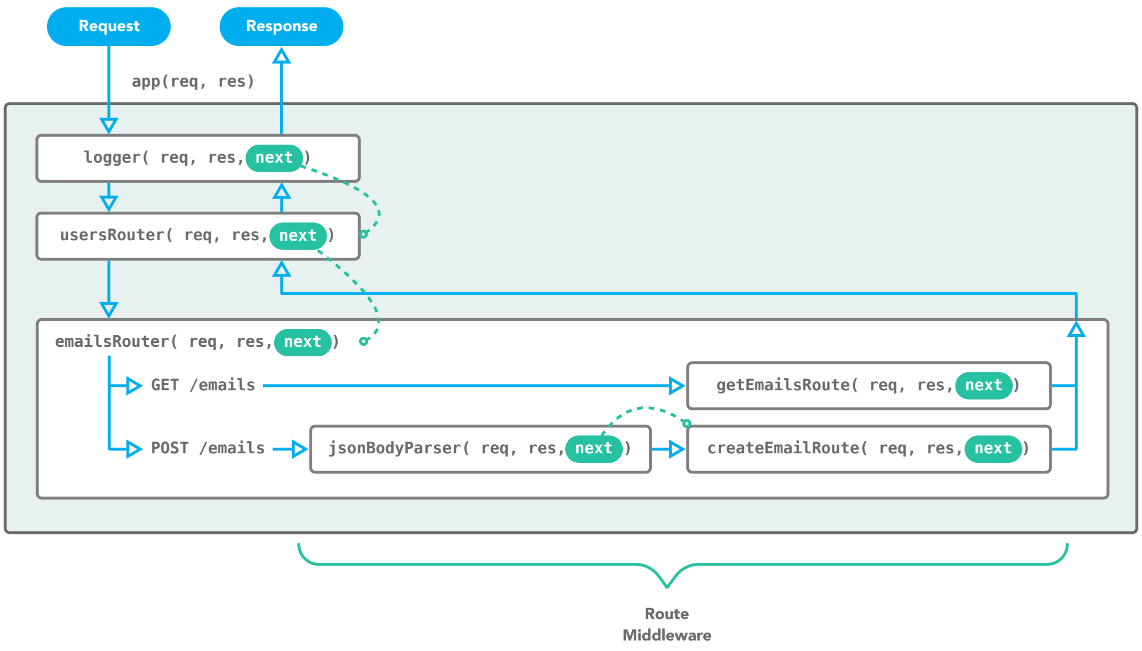

Request
Response
app(req, res)
logger(req, res, next)
usersRouter(req, res, next)
emailsRouter(req, res, next)
GET /emails
POST /emails
getEmailsRoute(req, res, next)
jsonBodyParser(req, res, next)
createEmailRoute(req, res, next)
Route Middleware

***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`: