Express API Validation Essentials

Simon Plenderleith

Table of Contents

Preface

What you're going to learn

In this book you'll learn:

- How Express middleware fits into a validation strategy.
- What the JSON Schema specification is and how you can apply it to validate request data.
- How the Problem Details for HTTP APIs specification can help you create awesome error responses.
- How to bring all these things together to build better Express APIs.

The code examples in this book will show you how to apply validation best practices in your Express APIs. The best part though, is the flexibility of the tools and techniques which I'm going to show you. By the end of the the book you'll be able to mix and match everything you've learnt to create your own API validation strategy.

Let's get started!

What you'll need to get the most from this book

You must have Node.js v14.13.0 or greater installed. I recommend using nvm to manage your Node.js installations.

When installing Express make sure you are using at least v4.16.0. This is the earliest 4.x version of Express which comes bundled with body parser middleware.

Code examples

All code examples in this book use the ECMAScript module syntax.

All code examples in this book are also bundled as standalone JavaScript files. They are contained in the `examples` directory of the zip file that you downloaded when you purchased this book. For usage instructions look at the README in that directory.

Get in touch

Feedback

If you have any feedback that you'd like to share about this book, drop me an email at simon@simonplend.co.uk.

Spotted a mistake?

If you've spotted a mistake in this book, please raise an issue on GitHub and I will review it: https://github.com/simonplend/express-api-validation-essentials-issues/

Reviews

As this book has been self-published, there's no official platform where you can leave a review for it. Please do tweet or blog your thoughts about it though!

Part 1: Doing things the Express way

In order to effectively validate requests to your API and send consistent error responses you'll need to work with the patterns and conventions which are baked in to Express. The most fundamental pattern it uses is "middleware".

> *Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.*

— *Source: Express guide: Using middleware*

In this part of the book we're going to dig into the middleware pattern. We'll also look at the different types of middleware and how they will shape our validation strategy.

The middleware pattern

In Express, middleware are a specific style of function which you configure your application to use. They can run any code you like, but they typically take care of processing incoming requests, sending responses and handling errors. They are the building blocks of every Express application.

When you define a route in Express, the route handler function which you specify for that route is a middleware function:

```
app.get("/user", function routeHandlerMiddleware(request, response, next) {
    // execute something
});
```

(Example 1.1)

Middleware is flexible. You can tell Express to run the same middleware function for different routes, enabling you to do things like making a common check across different API endpoints.

As well as writing your own middleware functions, you can also install third-party middleware to use in your application. The Express documentation lists some popular middleware modules. There are also a wide variety of Express middleware modules available on npm.

Middleware syntax

Here is the syntax for a middleware function:

```
/**
 * @param {Object} request - Express request object (commonly named `req`)
 * @param {Object} response - Express response object (commonly named `res`)
 * @param {Function} next - Express `next()` function
 */
function middlewareFunction(request, response, next) {
    // execute something
}
```

(Example 1.2)

> *Note: You might have noticed that I refer to `req` as `request` and `res` as `response`. You can name the parameters for your middleware functions whatever you like, but I prefer verbose*

> *variable names as I think that it makes it easier for other developers to understand what your code is doing, even if they're not familiar with the Express framework.*

When Express runs a middleware function, it is passed three arguments:

- An Express request object (commonly named `req`) - this is an extended instance of Node.js' built-in http.IncomingMessage class.
- An Express response object (commonly named `res`) - this is an extended instance of Node.js' built-in http.ServerResponse class.
- An Express `next()` function - Once the middleware function has completed its tasks, it must call the `next()` function to hand off control to the next middleware. If you pass an argument to it, Express assumes it to be an error. It will skip any remaining non-error handling middleware functions and start executing error handling middleware.

Middleware functions should not `return` a value. Any value returned by middleware will not be used by Express.

The two types of middleware

Plain middleware

Most middleware functions that you will work with in an Express application are what I call "plain" middleware (the Express documentation doesn't have a specific term for them). They look like the function defined in the middleware syntax example above (*Example 1.2*).

Here is an example of a plain middleware function:

```
function plainMiddlewareFunction(request, response, next) {
    console.log(`The request method is ${request.method}`);

    /**
     * Ensure the next middleware function is called.
     */
    next();
}
```

(Example 1.3)

Error handling middleware

The difference between error handling middleware and plain middleware is that error handler middleware functions specify four parameters instead of three i.e. `(error, request, response, next)`.

Here is an example of an error handling middleware function:

```
function errorHandlingMiddlewareFunction(error, request, response, next) {
    console.log(error.message);

    /**
     * Ensure the next error handling middleware is called.
     */
    next(error);
}
```

(Example 1.4)

This error handling middleware function will be executed when another middleware function calls the `next()` function with an error object e.g.

```
function anotherMiddlewareFunction(request, response, next) {
    const error = new Error("Something is wrong");

    /**
     * This will cause Express to start executing error
     * handling middleware.
     */
    next(error);
}
```

(Example 1.5)

Using middleware

The order in which middleware are configured is important. You can apply them at three different levels in your application:

- The route level
- The router level
- The application level

If you want a route (or routes) to have errors which they raise handled by an error handling middleware, you must add it after the route has been defined.

Let's look at what configuring middleware looks like at each level.

At the route level

This is the most specific level: any middleware you configure at the route level will only run for that specific route.

```
app.get("/", someMiddleware, routeHandlerMiddleware, errorHandlerMiddleware);
```

(Example 1.6)

At the router level

Express allows you to create Router objects. They allow you to scope middleware to a specific set of routes. If you want the same middleware to run for multiple routes, but not for all routes in your application, they can be very useful.

```
import express from "express";

const router = express.Router();

router.use(someMiddleware);

router.post("/user", createUserRouteHandler);
router.get("/user/:user_id", getUserRouteHandler);
router.put("/user/:user_id", updateUserRouteHandler);
router.delete("/user/:user_id", deleteUserRouteHandler);

router.use(errorHandlerMiddleware);
```

(Example 1.7)

At the application level

This is the least specific level. Any middleware configured at this level will be run for all routes.

```
app.use(someMiddleware);

// define routes

app.use(errorHandlerMiddleware);
```

(Example 1.8)

Technically you can define some routes, call `app.use(someMiddleware)` , then define some other routes which you want `someMiddleware` to be run for. I don't recommend this approach as it tends to result in a confusing and hard to debug application structure.

You should only configure middleware at the application level if absolutely necessary i.e. it really must be run for every single route in your application. Every middleware function, no matter how small, takes *some* time execute. The more middleware functions that need to be run for a route, the slower requests to that route will be. This really adds up as your application grows and is configured with lots of middleware. Try to scope middleware to the route or router levels when you can.

Middleware in a validation strategy

We will need to use middleware at three key points in the request validation process.

Request body parsing

Before a JSON request body can be validated, it must first be parsed.

For request body parsing we'll be using the `express.json()` middleware function. At the time of writing, this is one of only three middleware functions which are built-in to Express.

Request validation

We will be using the express-json-validator-middleware module to help us validate our requests.

If there are validation errors, the validator middleware will call the `next()` function with a `ValidationError` error object.

Sending an error response

We will create an error handler middleware to handle any `ValidationError` errors which have been created by the validator middleware.

Our error handler middleware will take care of formatting this error and sending it as a response back to the client.

Summary

In this part of the book, we've learnt about the middleware pattern in Express. We've also learnt about the different types of middleware and how they will fit into our validation strategy.

Now it's time to learn about the tools which can help us validate our API requests.