## **TESTING ROUTES AND CONTROLLERS**

Unit Testing is important for many reasons that we probably don’t need to cover right now. If you’ve already taken our basic JavaScript course you’ve already encountered Unit Testing, and the point of *this* lesson is not to teach you the philosophy or mechanics of writing tests, but how they apply to our Express applications and APIs.

If you haven’t finished our [front-end javascript course](https://www.theodinproject.com/courses/javascript), go back and take a look at those lessons before progressing.

### **Learning Outcomes**

By the end of this lesson, you should be able to do or answer the following:

* Use the supertest module to test Express routes/controllers.
* Describe how supertest handles our express application.
* Explain the functionality superagent provides to supertest.
* What is the purpose of done? What convenience does supertest provide concerning it?
* Explain and have a firm understanding of .expect() method’s functionality.
* Have familiarity with supertest’s documentation and methods.

The most important, basic requirement for testing something in your code is that it be in an exported module. This is true for both custom middleware and your routes/controllers, so the very first thing you need to do is separate those things into their own modules, if they aren’t already.

In the case of routes, you already know how to do this using Express.Router. Below is a very simple example.

//// app.js const express = require("express"); const app = express(); app.use(express.urlencoded({ extended: false })); const indexRouter = require("./index"); app.use("/", indexRouter); app.listen(3000, () => console.log("running"));

//// index.js const express = require("express"); const index = express.Router(); const array = []; index.get("/", (req, res) => { res.json({ name: "frodo" }); }); index.get("/test", (req, res) => res.json({ array })); index.post("/test", (req, res) => { array.push(req.body.item); res.send('success!'); }); module.exports = index;

These two files, app.js and index.js simply define a couple of routes and then set up and start our express app. For the moment we do *not* need to test app.js because it only contains code that starts and runs an express app! It doesn’t include any of our own logic so we don’t need to test it. index.js however *does* include some things that we want to test.

To facilitate actually testing these routes we’re going to use a library called [Supertest](https://github.com/visionmedia/supertest), so go ahead and npm install supertest and while it’s installing take a few minutes to look through the readme on their git repo (linked above).

In the examples below we’re going to use Supertest inside of a Jest style describe/test block, but the syntax and use of these are common among most testing libraries, so the concepts should be easily replicated in Mocha with Chai or Jasmine or Tape or whatever testing library you prefer.

Here’s our test file:

const index = require("../index"); const request = require("supertest"); const express = require("express"); const app = express(); app.use(express.urlencoded({ extended: false })); app.use("/", index); test("index route works", done => { request(app) .get("/") .expect("Content-Type", /json/) .expect({ name: "frodo" }) .expect(200, done); }); test("testing route works", done => { request(app) .post("/test") .type("form") .send({ item: "hey" }) .then(() => { request(app) .get("/test") .expect({ array: ["hey"] }, done); }); });

Let’s step through it piece by piece.

To begin, we have to import the module we’re testing, in this case it’s the file index.js from above.

const index = require("../index");

Next, we include both supertest and express. We’re setting up a new express app and then using the index router that we imported previously. The reason we have to do this setup here is because we are not actually touching our original app.js file. The main reason that we’re doing it this way is so that we can avoid calling the app.listen command and starting our server, but it’s also useful because, in larger apps, we can skip some of the optional configuration steps and only include the bits that we need for our testing purposes. In a larger test suite, it would probably be useful to abstract this part out to its own file that gets imported into each test file.

const request = require("supertest"); const express = require("express"); const app = express(); app.use(express.urlencoded({ extended: false })); app.use("/", index);

The tests themselves are relatively simple thanks to the Supertest library! Remember that we imported supertest as the function request which we use as seen below. We call it on our freshly created express app, pass it our route, and then use it to make sure that the responses match the types and content that we expect.

Notice the parameter done that is passed into the test callback. Most testing libraries use this to signal that the test is complete in the case of asynchronous operations. In this case, supertest allows us to pass it into the last .expect and calls it for us. Thanks, supertest!

test("index route works", done => { request(app) .get("/") .expect("Content-Type", /json/) .expect({ name: "frodo" }) .expect(200, done); });

Our second test is very similar to the first one, but tests the post method. You can (and should) read about all the possible functions on the supertest readme, so I won’t go into the details of every step here. The last bit however is important to us. By this point in your JavaScript career, you should be familiar with Promises, so the .then() syntax should be familiar. In this case we wait for the POST request to finish and then we call the GET request when that promise resolves to check if that item has been pushed into the array.

test("testing route works", done => { request(app) .post("/test") .type("form") .send({ item: "hey" }) .then(() => { request(app) .get("/test") .expect({ array: ["hey"] }, done); }); });

If we were using a real database here, then we would want to do something similar using either a test or a mock database. We’ll talk about setting something like that up in a separate lesson. Suffice it to say for now that you do not want to run test code on your production database!

### **Assignment**

1. Make sure that you read through the [Supertest docs](https://github.com/visionmedia/supertest)
2. Supertest actually pulls from another related project called Superagent. Any method that you can call in Superagent you can also call from supertest, so you’ll need to take a look through the [SuperAgent docs](http://visionmedia.github.io/superagent/) as well.