Endpoint testing with Jest and Supertest

I played around with testing lately. One thing I tried to do was to test the endpoints of my Express application.

Setting up the test was the hard part. People who write about tests don't actually teach you how they set it up. I could not find any useful information about this, and I had to try and figure it out.

So today, I want to share the setup I created for myself. Hopefully, this can help you when you create your own tests.

First, let's talk about the stack.

The Stack

- I created my app with Express.
- I used Mongoose to connect to MongoDB
- I used Jest as my test framework.

You might have expected Express and Mongoose because everyone else seems to use those two frameworks. I used them too.

But why Jest and not other test frameworks?

Why Jest

I don't like Facebook, so I didn't want to try anything that was created by Facebook's team. I know it sounds silly, but that was the truth.

Before Jest, I tried out all sorts of test frameworks. I tried Tap, Tape, Mocha, Jasmine, and AVA. Each test framework has its own pros and cons. I almost ended up with AVA, but I didn't go with AVA because I found it hard to set up. Eventually, I tried Jest out because Kent C. Dodds recommended it.

I fell in love with Jest after trying it out. I love it because:

1. It's easy to setup
2. The watch-mode is amazing
3. When you `console.log` something, it actually shows up without

any difficulty (this was a bitch with AVA).

Setting up Jest

First, you need to install Jest.

```
npm install jest --save-dev
```

Next, you want to add tests scripts to your `package.json` file. It helps to add the `test` and `test:watch` scripts (for one-off testing and watch-mode respectively).

```
"scripts": {
  "test": "jest",
  "test:watch": "jest --watch"
},
```

You can choose to write your test files in one of the following formats. Jest picks them up for you automatically.

1. `js` files in the `__tests__` folder
2. files named with `test.js` (like `user.test.js`)
3. files named with `spec.js` (like `user.spec.js`)

You can place your files however you like. When I tested endpoints, I put the test files together with my endpoints. I found this easier to manage.

```
- routes
  |- users/
    |- index.js
    |- users.test.js
```

Writing your first test

Jest includes `describe`, `it` and `expect` for you in every test file. You don't have to `require` them.

- `describe` lets you wrap many tests together under one umbrella. (It is used for organizing your tests).
- `it` lets you run a test.
- `expect` lets you perform assertions. The test passes if all assertions passes.

Here's an example of a test that fails. In this example, I `expect` that `1` should be strictly equal to `2`. Since `1 !== 2`, the test fails.

```
// This test fails because 1 !== 2
it('Testing to see if Jest works', () => {
  expect(1).toBe(2)
})
```

You'll see a failing message from Jest if you run Jest.

```
npm run test:watch
```

```
FAIL routes/litmus.test.js
  ✕ Testing to see if Jest works (1ms)

  ● Testing to see if Jest works

    expect(received).toBe(expected) // Object.is equality

    Expected: 2
    Received: 1

       5 |
       6 | it('Testing to see if Jest works', () => {
    >  7 |   expect(1).toBe(2)
         |             ^
       8 | })
       9 |
      10 | // it('gets the test endpoint', async done => {

      at Object.toBe (routes/litmus.test.js:7:13)
```

You can make the test pass by expecting `1 === 1`.

```
// This passes because 1 === 1
it('Testing to see if Jest works', () => {
  expect(1).toBe(1)
})
```

```
PASS routes/litmus.test.js
  ✓ Testing to see if Jest works (2ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.329s, estimated 1s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

This is the most basic of tests. It's not useful at all because we haven't testing anything real yet.

Asynchronous tests

You need to send a request to test an endpoint. Requests are asynchronous, which means you must be able to conduct asynchronous

tests.

This is easy with Jest. There are two steps:

1. Add the `async` keyword
2. Call `done` when you're done with your tests

Here's what it can look like:

```
it('Async test', async done => {
  // Do your async tests here

  done()
})
```

Note: Here's an article on Async/await in JavaScript if you don't know how to use it.

Testing Endpoints

You can use Supertest to test endpoints. First, you need to install Supertest.

```
npm install supertest --save-dev
```

Before you can test endpoints, you need to setup the server so Supertest can use it in your tests.

Most tutorials teach you to `listen` to the Express app in the server file, like this:

```
const express = require('express')
const app = express()

// Middlewares...
// Routes...

app.listen(3000)
```

This doesn't work because it starts listening to one port. If you try to write many test files, you'll get an error that says "port in use".

You want to allow each test file to start a server on their own. To do this, you need to export `app` without listening to it.

```
// server.js
const express = require('express')
const app = express()

// Middlewares...
// Routes...

module.exports = app
```

For development or production purposes, you can listen to your `app` like normal in a different file like `start.js`.

```
// start.js
const app = require('./server.js')
app.listen(3000)
```

Using Supertest

To use Supertest, you require your app and supertest in the test file.

```
const app = require('./server') // Link to your server file
const supertest = require('supertest')
const request = supertest(app)
```

Once you do this, you get the ability to send GET, POST, PUT, PATCH and DELETE requests. Before we send a request, we need to have an endpoint. Let's say we have a `/test` endpoint.

```
app.get('/test', async (req, res) => {
  res.json({message: 'pass!'})
})
```

To send a GET request to `/test`, you use the `.get` method from Supertest.

```
it('Gets the test endpoint', async done => {
  // Sends GET Request to /test endpoint
  const res = await request.get('/test')

  // ...
  done()
})
```

Supertest gives you a response from the endpoint. You can test both HTTP status and the body (whatever you send through `res.json`) like this:

```
it('gets the test endpoint', async done => {
  const response = await request.get('/test')

  expect(response.status).toBe(200)
  expect(response.body.message).toBe('pass!')
  done()
})
```

```
 PASS  routes/litmus.test.js
  ✓ Testing to see if Jest works
  ✓ gets the test endpoint (4ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.218s, estimated 1s
Ran all test suites related to changed files.

Watch Usage: Press w to show more.
```

If you want to find out more about Supertest, you can read its documentation here.

In the next article, I'll show you how to to do a POST request and how to connect to Mongoose in your test file.