Test a Node RESTful API with Mocha and Chai

Samuele Zaza (/@samuxyz)

July 12, 2016

#node.js (/tutorials?hFR%5Bcategory%5D%5B0%5D=Tutorials&dFR%5B_tags%5D%5B0%5D=node-js) #API (/tutorials?hFR%5Bcategory%5D%5B0%5D=Tutorials&dFR%5B_tags%5D%5B0%5D=api) #testing (/tutorials?hFR%5Bcategory%5D%5B0%5D=Tutorials&dFR%5B_tags%5D%5B0%5D=testing)

211,163 views 61 comments

Samuele Zaza(/@samuxyz)

(/@samuxyz)
Samuele Zaza (/@samuxyz)

July 12, 2016

#node.js (/tutorials?hFR%5Bcategory%5D%5B0%5D%5 js) #API (/tutorials?hFR%5Bcategory%5D%5B0%5D%5 #testing (/tutorials?hFR%5Bcategory%5D%5B0%5

211,163 views

61 comments

#Introduction

I still remember the satisfaction of being finally able to write the backend part of a bigger app in node and I am sure many of you do it too.

And then? We need to make sure our app behaves the way we expect and one of the strongly suggested methodologies is software testing. Software testing is crazily useful whenever a new feature is added to the system: Having the test environment already set up which can be run with a single command helps to figure out whether a new feature introduces new bugs.

Related Course: Getting Started with JavaScript for Web Development (https://bit.ly/2rVqDcs)

In the past, we've worked on building a RESTful Node API (/tutorials/build-a-restful-api-using-node-and-express-4) and authenticating a Node API (/tutorials/authenticate-a-node-js-api-with-json-web-tokens).

In this tutorial we are going to write a simple RESTful API with Node.js and use Mocha (https://mochajs.org/) and Chai (http://chaijs.com/) to write tests against it. We will test CRUD operations on a bookstore.

As usual you can build the app step-by-step throughout the tutorial or directly get it on github (https://github.com/samuxyz/bookstore).

#Mocha: Testing Environment

Mocha is a javascript framework for Node.js which allows Asynchronous testing. Let's say it provides the environment in which we can use our favorite assertion libraries to test the code.

mocha-homepage

.

Mocha comes with tons of great features, the website shows a long list but here are the ones I like the most:

- simple async support, including promises.
- async test timeout support.
- before, after, before each, after each hooks (very useful to clean the environment where each test!).

- use any assertion library you want, Chai in our tutorial.

#Chai: Assertion Library

So with Mocha we actually have the environment for making our tests but how do we do test HTTP calls for example? Moreover, How do we test whether a GET request is actually returning the JSON file we are expective, given a defined input? We need an assertion library, that's why mocha is not enough.

So here it is *Chai*, the assertion library for the current tutorial:

chai-homepage

Chai shines on the freedom of choosing the interface we prefer: "should", "expect", "assert" they are all available. I personally use should but you are free to check it out the API (http://chaijs.com/api/) and switch to the others two. Lastly Chai HTTP (https://github.com/chaijs/chai-http) addon allows Chai library to easily use assertions on HTTP requests which suits our needs.

Prerequisites

- *Node.js*: a basic understanding of node.js and is recommended as i wont go too much into detail on building a RESTful API.
- *POSTMAN for making fast HTTP requests to the API.*
- *ES6 syntax*: I decided to use the latest version of Node (6.*.*) which has the highest integration of ES6 features for better code readibility. If you are not familiar with ES6 you can take a look at the great scotch articles (Pt.1 (https://scotch.io/tutorials/better-node-with-es6-pt-i) , Pt.2 (https://scotch.io/tutorials/better-javascript-with-es6-pt-ii-a-deep-dive-into-classes) and Pt.3 (https://scotch.io/tutorials/better-javascript-with-es6-pt-iii-cool-collections-slicker-strings)) about it but do not worry I am going to spend a few words whenever we encount some "exotic" syntax or declaration.

Time to setup our Bookstore!

#Project setup

Directory Structure

Here is the project directory for our API, something you must have seen before:

BASH

```
-- controllers
---- models
------ book.js
---- routes
------ book.js
-- config
---- default.json
---- dev.json
---- test.json
-- test
---- book.js
package.json
server.json
```

Notice the `/config` folder containing 3 JSON files: As the name suggests, they contain particular configurations for a specific purpose.

In this tutorial we are going to switch between two databases, one for development and one for testing purposes, thus the files contain the mongodb URI in JSON format:

`dev.json` and `default.json`:

JAVASCRIPT

```
{
    "DBHost": "YOUR_DB_URI"
}
```

`test.json`:

JAVASCRIPT

```
{
    "DBHost": "YOUR_TEST_DB_URI"
}
```

NB: `default.json` is optional however let me highlight that files in the config directory are loaded starting from it. For more information about the configuration files (config directory, file order, file format etc.) check out this link (https://github.com/lorenwest/node-config/wiki/Configuration-Files).

Finally, notice `/test/book.js`, that's where we are going to write our tests!

Package.json

Create the `package.json` file and paste the following code:

JAVASCRIPT

```
{
  "name": "bookstore",
  "version": "1.0.0",
  "description": "A bookstore API",
  "main": "server.js",
  "author": "Sam",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.15.1",
    "config": "^1.20.1",
    "express": "^4.13.4",
    "mongoose": "^4.4.15",
    "morgan": "^1.7.0"
  },
  "devDependencies": {
    "chai": "^3.5.0",
    "chai-http": "^2.0.1",
    "mocha": "^2.4.5"
  },
  "scripts": {
    "start": "SET NODE_ENV=dev && node server.js",
    "test": "mocha --timeout 10000"
  }
}
```

1 61

Again the configuration should not surprise anyone who wrote more than a server with node.js, the test-related packages *mocha*, *chai*, *chai-http* are saved in the dev-dependencies (flag `--save-dev` from command line) while the scripts property allows for two different ways of running the server.

To run mocha I added the flag `--timeout 10000` because I fetch data from a database hosted on mongolab so the default 2 seconds may not be enough.

Congrats! You made it through the boring part of the tutorial, now it is time to write the server and test it.

#The server

Main

Let's create the file `server.js` in the root of the project and paste the following code:

JAVASCRIPT

```
let express = require('express');
let app = express();
let mongoose = require('mongoose');
let morgan = require('morgan');
let bodyParser = require('body-parser');
let port = 8080;
let book = require('./app/routes/book');
let config = require('config'); //we load the db location from the JSON files
//db options
let options = {
                server: { socketOptions: { keepAlive: 1, connectTimeoutMS: 30000 }
                replset: { socketOptions: { keepAlive: 1, connectTimeoutMS : 30000
              };

//db connection
mongoose.connect(config.DBHost, options);
let db = mongoose.connection;
db.on('error', console.error.bind(console, 'connection error:'));

//don't show the log when it is test
if(config.util.getEnv('NODE_ENV') !== 'test') {
    //use morgan to log at command line
    app.use(morgan('combined')); //'combined' outputs the Apache style LOGs
}

//parse application/json and look for raw text
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));
app.use(bodyParser.text());
app.use(bodyParser.json({ type: 'application/json'}));

app.get("/", (req, res) => res.json({message: "Welcome to our Bookstore!"}));

app.route("/book")
    .get(book.getBooks)
    .post(book.postBook);
app.route("/book/:id")
    .get(book.getBook)
    .delete(book.deleteBook)
    .put(book.updateBook);

app.listen(port);
console.log("Listening on port " + port);

module.exports = app; // for testing
```

Here are the key concepts:

- We require the module config to access the configuration file named as the *NODE_ENV* content to get the mongo db URI parameter for the db connection. This helps us to keep the "real" database clean by testing on another database hidden to our app future users.

- The enviroment variable *NODE_ENV* is test against *test* to disable morgan (https://www.npmjs.com/package/morgan) log in the command line or it would interfere with the test output.
- The last line of code exports the server for testing purposes.
- Notice the variables definition using let which makes the variable enclosed to the nearest enclosing block or global if outside any block.

The remaining lines of codes are nothing new, we simply go through requiring all the necessary modules, define the header options for the communication with the server, craete the specific roots and eventually let the server listen on a defined port.

Model and Routes

Time for our book model! Create a file in `/app/model/` called `book.js` and paste the following code:

JAVASCRIPT

```javascript
let mongoose = require('mongoose');
let Schema = mongoose.Schema;

//book schema definition
let BookSchema = new Schema(
  {
    title: { type: String, required: true },
    author: { type: String, required: true },
    year: { type: Number, required: true },
    pages: { type: Number, required: true, min: 1 },
    createdAt: { type: Date, default: Date.now },
  },
  {
    versionKey: false
  }
);

// Sets the createdAt parameter equal to the current time
BookSchema.pre('save', next => {
  now = new Date();
  if(!this.createdAt) {
    this.createdAt = now;
  }
  next();
});

//Exports the BookSchema for use elsewhere.
module.exports = mongoose.model('book', BookSchema);
```

Our book schema has a title, author, the number of pages, the publication year and the date of creation in the db. I set the versionKey to false since it's useless for the purpose of the tutorial.

NB: the exotic callback syntax in the `.pre()` function is an arrow function, a function who has a shorter syntax which, according to the definiton on MDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) , *"lexically binds the this value (does not bind its own this, arguments, super, or new.target). Arrow functions are always anonymous"*.

Well, pretty much all we need to know about the model so let's move to the routes.

in `/app/routes/` create a file called book.js and paste the following code:

JAVASCRIPT

```
let mongoose = require('mongoose');
let Book = require('../models/book');

/*
 * GET /book route to retrieve all the books.
 */
function getBooks(req, res) {
    //Query the DB and if no errors, send all the books
    let query = Book.find({});
    query.exec((err, books) => {
        if(err) res.send(err);
        //If no errors, send them back to the client
        res.json(books);
    });
}

/*
 * POST /book to save a new book.
 */
function postBook(req, res) {
    //Creates a new book
    var newBook = new Book(req.body);
    //Save it into the DB.
    newBook.save((err,book) => {
        if(err) {
            res.send(err);
        }
        else { //If no errors, send it back to the client
            res.json({message: "Book successfully added!", book });
        }
    });
}

/*
 * GET /book/:id route to retrieve a book given its id.
 */
function getBook(req, res) {
    Book.findById(req.params.id, (err, book) => {
        if(err) res.send(err);
        //If no errors, send it back to the client
        res.json(book);
    });
}

/*
 * DELETE /book/:id to delete a book given its id.
 */
function deleteBook(req, res) {
    Book.remove({_id : req.params.id}, (err, result) => {
        res.json({ message: "Book successfully deleted!", result });
    });
}

/*
 * PUT /book/:id to updatea a book given its id
 */
```

```
function updateBook(req, res) {
    Book.findById({_id: req.params.id}, (err, book) => {
        if(err) res.send(err);
        Object.assign(book, req.body).save((err, book) => {
            if(err) res.send(err);
            res.json({ message: 'Book updated!', book });
        });
    });
}

//export all the functions
module.exports = { getBooks, postBook, getBook, deleteBook, updateBook };
```

Here the key concepts:

- The routes are no more than standard routes, GET, POST, DELETE, PUT to perform CRUD operations on our data.
- In the function `updatedBook()` we use `Object.assign` , a new function introduced in ES6 which, in this case, overrides the common properties of book with `req.body` while leaving untouched the others.
- At the end we export the object using a faster syntax which pairs key and value to avoid useless repetitions.

We finished this section and actually we have a working app!

#A Naive Test

Now let's run the app and open POSTMAN to send HTTP request to the server and check if everything is working as expected.

in the command line run

BASH

```bash
npm start
```

GET /book

in POSTMAN run the GET request and, assuming the database contains books, here is the result:

get-book-list

The server correctly returned the book list in my database.

POST /book

Let's add a book and POST to the server:

post-book

It seems the book was perfectly added. The server returned the book and a message confirming it was added in our bookstore. Is it true? Let's send another GET request and here is the result:

get-book-list

Awesome it works!

PUT /book/:id

Let's update a book by changing the page and check the result:

Great! PUT also seems to be working so let's send another GET request to check all the list:

All is running smoothly...

GET /book/:id

Now let's get a single book by sending the id in the GET request and then delete it:

As it returns the correct book let's try now to delete it:

DELETE /book/:id

Here is the result of the DELETE request to the server:

delete-book

Even the last request works smoothly and we do not need to doublecheck with another GET request as we are sending the client some info from mongo (result property) which states the book was actually deleted.

By doing some test with POSTMAN the app happened to behave as expected right? So, would you shoot it to your clients?

Let me reply for you: **NO!!**

Ours is what I called a naive test because we simply tried few operations without testing strange situations that may happen: A post request without some expected data, a DELETE with a wrong id as parameter or even without id to name few.

This is obviously a simple app and if we were lucky enough, we coded it without introducing bugs of any sort, but what about a real-world app? Moreover, we spent time to run with POSTMAN some test HTTP requests so what would happen if one day we had to change the code of one of those? Test them all again with POSTMAN? Have you started to realize this is not an agile approach?

This is nothing but few situations you may encounter and you already encountered in your journey as a developer, luckily we have tools to create tests which are always available and can be launched with a single comman line.

Let's do something better to test our app!

#A Better Test

First, let's create a file in `/test` called `book.js` and paste the following code:

JAVASCRIPT

```
//During the test the env variable is set to test
process.env.NODE_ENV = 'test';

let mongoose = require("mongoose");
let Book = require('../app/models/book');

//Require the dev-dependencies
let chai = require('chai');
let chaiHttp = require('chai-http');
let server = require('../server');
let should = chai.should();

chai.use(chaiHttp);
//Our parent block
describe('Books', () => {
    beforeEach((done) => { //Before each test we empty the database
        Book.remove({}, (err) => {
           done();
        });
    });
/*
  * Test the /GET route
  */
  describe('/GET book', () => {
      it('it should GET all the books', (done) => {
        chai.request(server)
            .get('/book')
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('array');
                res.body.length.should.be.eql(0);
              done();
            });
      });
  });

});
```

Wow that's a lot of new things, let's dig into it:

1. You must have noticed the we set the *NODE_ENV* variable to test, by doing so we change the configuration file to be loaded so the server will connect to the test database and avoid morgan logs in the cmd.
2. We required the dev-dependencies modules and server itself (Do you remember we exported it by `module.exports` ?).
3.
4. We defined `should` by running `chai.should()` to style our tests on the HTTP requests result, then we told chai to use chai HTTP.

So it starts with "describe" blocks of code for better organizing your assertions and this organization will reflect in the output at command line as we will see later.

`beforeEach` is a block of code that is going to run before each the describe blocks on the same level. Why we did that? Well we are going to remove any book from the database to start with an empty bookstore whenever a test is run.

Test the /GET route

And here it comes the first test, chai is going to perform a GET request to the server and the assertions on the res variable will satisfy or reject the first parameter of the the `it` block *it should GET all the books*. Precisely, given the empty bookstore the result of the request should be:

1. Status 200.
2. The result should be an array.
3. Since the bookstore is empty, we presumed the length is equal to 0.

Notice that the syntax of *should* assertions is very intituitive as it is similar as a natural language statement.

Now, in the command line run:

JAVASCRIPT

```
npm test
```

and here it is the output:

The test passed and the output reflects the way we organized our code with blocks of `describe`.

Test the /POST route

Now let's check our robust is our API, suppose we are trying to add a book with missing pages field passed to the server: The server should not respond with a proper error message.

Copy and paste the following code in the test file:

JAVASCRIPT

```
process.env.NODE_ENV = 'test';

let mongoose = require("mongoose");
let Book = require('../app/models/book');

let chai = require('chai');
let chaiHttp = require('chai-http');
let server = require('../server');
let should = chai.should();

chai.use(chaiHttp);

describe('Books', () => {
    beforeEach((done) => {
        Book.remove({}, (err) => {
           done();
        });
    });
  describe('/GET book', () => {
      it('it should GET all the books', (done) => {
        chai.request(server)
            .get('/book')
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('array');
                res.body.length.should.be.eql(0);
              done();
            });
      });
  });
  /*
  * Test the /POST route
  */
  describe('/POST book', () => {
      it('it should not POST a book without pages field', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954
        }
        chai.request(server)
            .post('/book')
            .send(book)
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('object');
                res.body.should.have.property('errors');
                res.body.errors.should.have.property('pages');
                res.body.errors.pages.should.have.property('kind').eql('required');
              done();
            });
      });

  });
});
```

Here we added the test on an incomplete /POST request, let's analyze the assertions:

1. Status should be 200.
2. The response body should be an object.
3. One of the body properties should be `errors`.
4. `Errors` should have have the missing field pages as property.
5. Finally `pages` should have the property `kind` equal to *required* in order to highlight the reason why we got a negative answer from the server.

NB notice that we send the book along with the POST request by the `.send()` function.

Let's run the same command again and here is the output:

Oh Yeah our test test is correct!

Before writing a new test let me precise two things:

1. First of all, why the server response structured that way? If you read the callback function for the /POST route, you will notice that in case of missing required fields, the server sends back the error message from mongoose. Try with POSTMAN and check the response.
2. In case of missing fields we still return a status of 200, this is for simplicity as we are just learning to test our routes. However I suggest to return a status of *206 Partial Content instead*.

Let's send a book with all the required fields this time. Copy and paste the following code in the test file:

JAVASCRIPT

```
process.env.NODE_ENV = 'test';

let mongoose = require("mongoose");
let Book = require('../app/models/book');

let chai = require('chai');
let chaiHttp = require('chai-http');
let server = require('../server');
let should = chai.should();

chai.use(chaiHttp);

describe('Books', () => {
    beforeEach((done) => {
        Book.remove({}, (err) => {
           done();
        });
    });
  describe('/GET book', () => {
      it('it should GET all the books', (done) => {
        chai.request(server)
            .get('/book')
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('array');
                res.body.length.should.be.eql(0);
              done();
            });
      });
  });
  /*
  * Test the /POST route
  */
  describe('/POST book', () => {
      it('it should not POST a book without pages field', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954
        }
        chai.request(server)
            .post('/book')
            .send(book)
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('object');
                res.body.should.have.property('errors');
                res.body.errors.should.have.property('pages');
                res.body.errors.pages.should.have.property('kind').eql('required');
              done();
            });
      });
      it('it should POST a book ', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954,
            pages: 1170
        }
        chai.request(server)
            .post('/book')
            .send(book)
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('object');
                res.body.should.have.property('message').eql('Book successfully add
                res.body.book.should.have.property('title');
                res.body.book.should.have.property('author');
                res.body.book.should.have.property('pages');
                res.body.book.should.have.property('year');
              done();
            });
      });
  });
});
```

This time we expect a returning object with a message saying we succesfully added the book and the book itself (remember with POSTMAN?). You should be now quite familiar with the assertions I made so there is no need for going into detail. Instead, run the command again and here is the output:

Smooth~

Test /GET/:id Route

Now let's create a book, save it into the database and use the id to send a GET request to the server. Copy and paste the following code in the test file:

JAVASCRIPT

```
process.env.NODE_ENV = 'test';

let mongoose = require("mongoose");
let Book = require('../app/models/book');

let chai = require('chai');
let chaiHttp = require('chai-http');
let server = require('../server');
let should = chai.should();

chai.use(chaiHttp);

describe('Books', () => {
    beforeEach((done) => {
        Book.remove({}, (err) => {
           done();
        });
    });
  describe('/GET book', () => {
      it('it should GET all the books', (done) => {
            chai.request(server)
            .get('/book')
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('array');
                res.body.length.should.be.eql(0);
              done();
            });
      });
  });
  describe('/POST book', () => {
      it('it should not POST a book without pages field', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954
        }
            chai.request(server)
            .post('/book')
            .send(book)
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('object');
                res.body.should.have.property('errors');
                res.body.errors.should.have.property('pages');
                res.body.errors.pages.should.have.property('kind').eql('required');
              done();
            });
      });
      it('it should POST a book ', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954,
            pages: 1170
        }
```

```
                chai.request(server)
                .post('/book')
                .send(book)
                .end((err, res) => {
                    res.should.have.status(200);
                    res.body.should.be.a('object');
                    res.body.should.have.property('message').eql('Book successfully add
                    res.body.book.should.have.property('title');
                    res.body.book.should.have.property('author');
                    res.body.book.should.have.property('pages');
                    res.body.book.should.have.property('year');
                  done();
                });
          });
      });
     /*
      * Test the /GET/:id route
      */
      describe('/GET/:id book', () => {
          it('it should GET a book by the given id', (done) => {
            let book = new Book({ title: "The Lord of the Rings", author: "J.R.R. Tolki
            book.save((err, book) => {
                chai.request(server)
                .get('/book/' + book.id)
                .send(book)
                .end((err, res) => {
                    res.should.have.status(200);
                    res.body.should.be.a('object');
                    res.body.should.have.property('title');
                    res.body.should.have.property('author');
                    res.body.should.have.property('pages');
                    res.body.should.have.property('year');
                    res.body.should.have.property('_id').eql(book.id);
                  done();
                });
            });

          });
      });
    });
```

Through the assertions we made sure the server returned all the fields and the right book testing the two *ids* together. Here is the output:

Have you noticed that by testing single routes within independent blocks we provide a very clear output? Also, isn't it so efficient? We wrote several tests that can be repeated with a single command line, once and for all.

Test the /PUT/:id Route

Time for testing an update on one of our books, we first save the book and then update the year it was published. So, copy and paste the following code:

JAVASCRIPT

```
process.env.NODE_ENV = 'test';

let mongoose = require("mongoose");
let Book = require('../app/models/book');

let chai = require('chai');
let chaiHttp = require('chai-http');
let server = require('../server');
let should = chai.should();

chai.use(chaiHttp);

describe('Books', () => {
    beforeEach((done) => {
        Book.remove({}, (err) => {
           done();
        });
    });
  describe('/GET book', () => {
      it('it should GET all the books', (done) => {
            chai.request(server)
            .get('/book')
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('array');
                res.body.length.should.be.eql(0);
              done();
            });
      });
  });
  describe('/POST book', () => {
      it('it should not POST a book without pages field', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954
        }
            chai.request(server)
            .post('/book')
            .send(book)
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('object');
                res.body.should.have.property('errors');
                res.body.errors.should.have.property('pages');
                res.body.errors.pages.should.have.property('kind').eql('required');
              done();
            });
      });
      it('it should POST a book ', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954,
            pages: 1170
        }
```

```
                chai.request(server)
                .post('/book')
                .send(book)
                .end((err, res) => {
                    res.should.have.status(200);
                    res.body.should.be.a('object');
                    res.body.should.have.property('message').eql('Book successfully add
                    res.body.book.should.have.property('title');
                    res.body.book.should.have.property('author');
                    res.body.book.should.have.property('pages');
                    res.body.book.should.have.property('year');
                  done();
                });
          });
      });
      describe('/GET/:id book', () => {
          it('it should GET a book by the given id', (done) => {
            let book = new Book({ title: "The Lord of the Rings", author: "J.R.R. Tolki
            book.save((err, book) => {
                chai.request(server)
                .get('/book/' + book.id)
                .send(book)
                .end((err, res) => {
                    res.should.have.status(200);
                    res.body.should.be.a('object');
                    res.body.should.have.property('title');
                    res.body.should.have.property('author');
                    res.body.should.have.property('pages');
                    res.body.should.have.property('year');
                    res.body.should.have.property('_id').eql(book.id);
                  done();
                });
            });

          });
      });
     /*
      * Test the /PUT/:id route
      */
      describe('/PUT/:id book', () => {
          it('it should UPDATE a book given the id', (done) => {
            let book = new Book({title: "The Chronicles of Narnia", author: "C.S. Lewis
            book.save((err, book) => {
                    chai.request(server)
                    .put('/book/' + book.id)
                    .send({title: "The Chronicles of Narnia", author: "C.S. Lewis", yea
                    .end((err, res) => {
                        res.should.have.status(200);
                        res.body.should.be.a('object');
                        res.body.should.have.property('message').eql('Book updated!');
                        res.body.book.should.have.property('year').eql(1950);
                      done();
                    });
              });
          });
      });
    });
```

We wanna make sure the message is *the correct Book updated!* one and that the `year` field was actually updated. Here is the output:

Good, we are close to the end, we still gotta test the DELETE route.

Test the /DELETE/:id Route

The pattern is similar to the previous tests, we first store a book, delete it and test against the response. Copy and paste the following code:

JAVASCRIPT

```javascript
process.env.NODE_ENV = 'test';

let mongoose = require("mongoose");
let Book = require('../app/models/book');

let chai = require('chai');
let chaiHttp = require('chai-http');
let server = require('../server');
let should = chai.should();

chai.use(chaiHttp);

describe('Books', () => {
    beforeEach((done) => {
        Book.remove({}, (err) => {
           done();
        });
    });
  describe('/GET book', () => {
      it('it should GET all the books', (done) => {
            chai.request(server)
            .get('/book')
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('array');
                res.body.length.should.be.eql(0);
              done();
            });
      });
  });
  describe('/POST book', () => {
      it('it should not POST a book without pages field', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954
        }
            chai.request(server)
            .post('/book')
            .send(book)
            .end((err, res) => {
                res.should.have.status(200);
                res.body.should.be.a('object');
                res.body.should.have.property('errors');
                res.body.errors.should.have.property('pages');
                res.body.errors.pages.should.have.property('kind').eql('required');
              done();
            });
      });
      it('it should POST a book ', (done) => {
        let book = {
            title: "The Lord of the Rings",
            author: "J.R.R. Tolkien",
            year: 1954,
            pages: 1170
        }
```

```
                chai.request(server)
                .post('/book')
                .send(book)
                .end((err, res) => {
                    res.should.have.status(200);
                    res.body.should.be.a('object');
                    res.body.should.have.property('message').eql('Book successfully add
                    res.body.book.should.have.property('title');
                    res.body.book.should.have.property('author');
                    res.body.book.should.have.property('pages');
                    res.body.book.should.have.property('year');
                  done();
                });
          });
      });
      describe('/GET/:id book', () => {
          it('it should GET a book by the given id', (done) => {
            let book = new Book({ title: "The Lord of the Rings", author: "J.R.R. Tolki
            book.save((err, book) => {
                chai.request(server)
                .get('/book/' + book.id)
                .send(book)
                .end((err, res) => {
                    res.should.have.status(200);
                    res.body.should.be.a('object');
                    res.body.should.have.property('title');
                    res.body.should.have.property('author');
                    res.body.should.have.property('pages');
                    res.body.should.have.property('year');
                    res.body.should.have.property('_id').eql(book.id);
                  done();
                });
            });

          });
      });
      describe('/PUT/:id book', () => {
          it('it should UPDATE a book given the id', (done) => {
            let book = new Book({title: "The Chronicles of Narnia", author: "C.S. Lewis"
            book.save((err, book) => {
                    chai.request(server)
                    .put('/book/' + book.id)
                    .send({title: "The Chronicles of Narnia", author: "C.S. Lewis", yea
                    .end((err, res) => {
                        res.should.have.status(200);
                        res.body.should.be.a('object');
                        res.body.should.have.property('message').eql('Book updated!');
                        res.body.book.should.have.property('year').eql(1950);
                      done();
                    });
              });
          });
      });
     /*
      * Test the /DELETE/:id route
      */
      describe('/DELETE/:id book', () => {
          it('it should DELETE a book given the id', (done) => {
            let book = new Book({title: "The Chronicles of Narnia", author: "C.S. Lewis'
            book.save((err, book) => {
                    chai.request(server)
                    .delete('/book/' + book.id)
                    .end((err, res) => {
                        res.should.have.status(200);
                        res.body.should.be.a('object');
                        res.body.should.have.property('message').eql('Book successfully
                        res.body.result.should.have.property('ok').eql(1);
                        res.body.result.should.have.property('n').eql(1);
                      done();
                    });
              });
          });
      });
  });
```

Again the server returns a message and properties from mongoose that we assert so let's check the output:

Great, our tests are all positive and we have a good basis to continue testing our routes with more sophisticated assertions.

Congratulation for completing the tutorial!

#Conclusion

In this tutorial we faced the problem of testing our routes to provide our users a stable experience.

We went through all the steps of creating a RESTful API, doing a naive test with POSTMAN and then propose a better way to test, in fact the main topic of the tutorial.

It is good habit to always spend some time making tests to assure a server as reliable as possible but unfortunately it is often underestimated.

During the tutorial we also discuss a few benefits of code testing and this will open doors to more advanced topics such as Test Driven Development (TDD).

Good job!

#Bonus Mockgoose

One may argue that using two different databases is not the best situation and a second one is often not available. So what to do? Well, there is an alternative: Mockgoose.

Basically Mockgoose (https://github.com/mccormicka/Mockgoose) wraps your mongoose driver by intercepting the connection so your database will not be touched but instead use in memory store. Moreover it integrates well with *Mocha*.

NB: Apparently it requires mongodb to be installed on the running machine.

Latest Posts

CONTENTS

A Zero Configuration Module Bundler -- Meet Parcel(/tutorials/a-zero-configuration-module-bundler-meet-parcel)

End of Year Giveaway 2017(/bar-talk/end-of-year-giveaway-2017)

Building a Slack Bot with Modern Node.js Workflows(/tutorials/building-a-slack-bot-with-modern-nodejs-workflows)

Understanding JavaScript Closures: A Practical Approach(/tutorials/understanding-javascript-closures-a-practical-approach)

Build a Secure To-Do App with Vue, ASP.NET Core, and Okta(/tutorials/build-a-secure-to-do-app-with-vuejs-aspnet-core-and-okta)

More Posts(/tutorials?hFR%5Bcategory%5D%5B0%5D=Tutorials)

(/@samuxyz)

SAMUELE ZAZA (/@SAMUXYZ)

I am a full-stack web developer working for Taroko Software as front-end web developer and Filestack Tech Evangelist. When not coding I may be spotted in a gym lifting or planning to conquer the world LOL.

VIEW MY 7 POSTS (/@SAMUXYZ)

(/@samuxyz)

Samuele Zaza (/@samuxyz)

7 posts (/@samuxyz)

I am a full-stack web developer working for Taroko Software as front-end web developer and Filestack Tech Evangelist. When not coding I may be spotted in a gym lifting or planning to conquer the world LOL.

(/@samuxyz) (https://facebook.com/samuelezaza) (https://github.com/samuxyz) (https://www.linkedin.com/in/samuele-zaza-46523681?trk=nav_responsive_tab_profile)

CONTENTS

(https://twitter.com/scotch_io)
(https://www.facebook.com/scotchdevelopment)
(https://github.com/scotch-io)

CONTENTS

FAQ (/faq) Privacy (/privacy) Terms (/terms) Rules (/rules) Hosted by Digital Ocean (https://m.do.co/c/7a59e9361ab7)