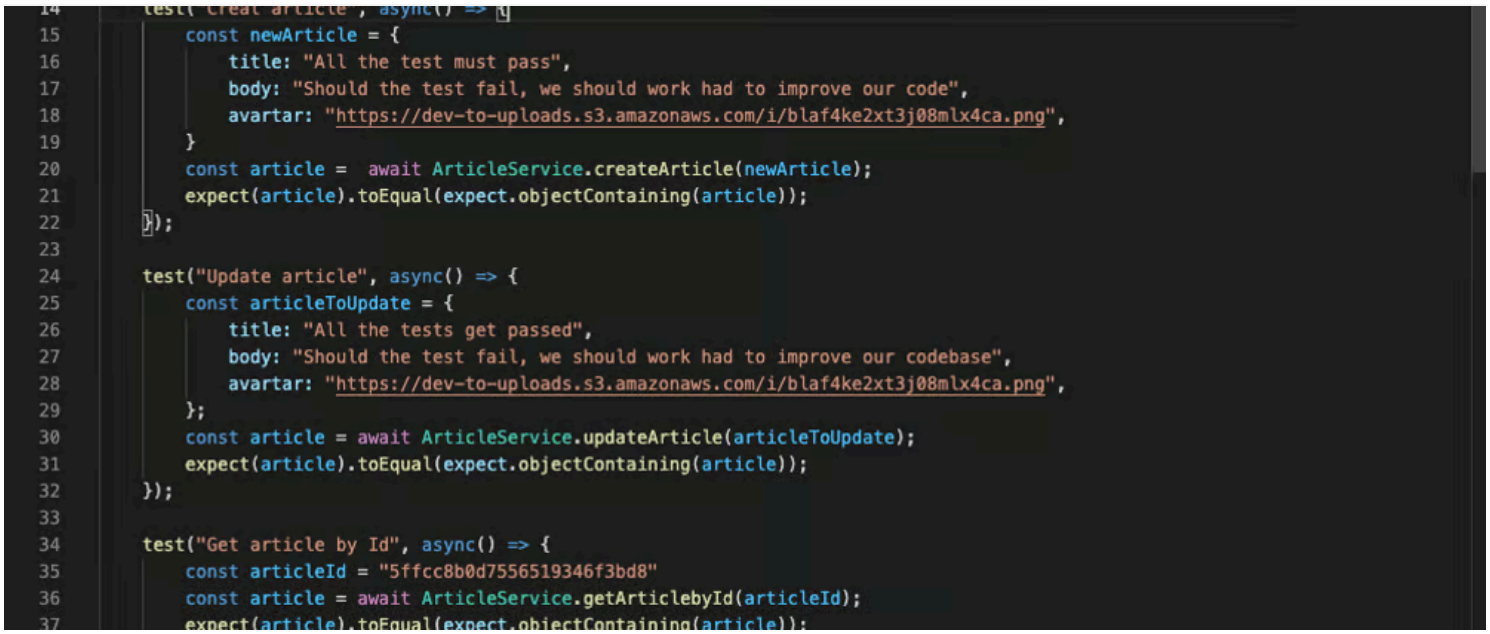

```
    test("Creat article", async() => {
        const newArticle = {
            title: "All the test must pass",
            body: "Should the test fail, we should work had to improve our code",
            avartar: "https://dev-to-uploads.s3.amazonaws.com/i/blaf4ke2xt3j08mlx4ca.png",
        }
        const article =  await ArticleService.createArticle(newArticle);
        expect(article).toEqual(expect.objectContaining(article));
    });

    test("Update article", async() => {
        const articleToUpdate = {
            title: "All the tests get passed",
            body: "Should the test fail, we should work had to improve our codebase",
            avartar: "https://dev-to-uploads.s3.amazonaws.com/i/blaf4ke2xt3j08mlx4ca.png",
        };
        const article = await ArticleService.updateArticle(articleToUpdate);
        expect(article).toEqual(expect.objectContaining(article));
    });

    test("Get article by Id", async() => {
        const articleId = "5ffcc8b0d7556519346f3bd8"
        const article = await ArticleService.getArticlebyId(articleId);
        expect(article).toEqual(expect.objectContaining(article));
```

Emmanuel Etukudo
Posted on Jan 13, 2021

11

Server-Side Testing with Jest

#node #javascript #testing #mongodb

This is the last tutorial for the Test-driven Development with Nodejs, Express, Mongoose & Jest series, in this tutorial we will focus on writing unit tests for the endpoints we built in the previous tutorial; Understanding MVC pattern in Nodejs.

Recall that we had covered **installing** of the **Jest** package via **npm**, and writing our first test in **Jest**. If you are reading this series for the first time, please follow the first tutorial here to get up and running.

Before we proceed, let's look at the topics covered in this tutorial.

- Unit Testing
- Mocking Technique
- Parameterized testing
- Configuring Jest to work with Nodejs

Unit Testing

Unit testing is a software testing technique where individual units (components) of software is tested. The purpose of unit testing is to validate that each unit of the software performs individual tasks as

designed. A unit is the smallest testable part of any software.

Mocking Technique

Mocking is a technique where dummy values are referenced during testing to emulate an actual scenario or real code. Mocking helps achieve isolation of tests. Mocking is applicable to unit testing.

Parameterized testing

Parameterized tests allow us to run the same test multiple times using different values. This approach will help our code to test for different cases and seniors. In jest the must popular function used in **Parameterized Testing** is the **each()** global function.

Configuring `Jest` to work with `Nodejs`

Because Jest is primarily designed for testing `React` application so using `Jest` to test server-side applications (e.g: `Nodejs`) reacquires some configurations. Jest uses the `jsdom` test environment by default, it attempts to create a browser-like test environment in Node.js. Mongoose does not support `jsdom` in general and is not expected to function correctly in the `jsdom` test environment.

To change your `testEnvironment` to Node.js, create a new file name `jest.config.js` within the root directory of your `tdd-with-nodejs` project, and copy-paste the code below to add `testEnvironment` to your `jest.config.js` file:

```
module.exports = {
  testEnvironment: 'node'
};
```

Here we have explored a very basic config, you can read more about testing MongoDB with Jest here.

Testing the DB Connection

Now that you are familiar with our todo-list, let's begin the business of the day. First, open your `"tdd-with-nodejs"` project in your favorite code editor, navigate into the `test` directory, Delete the `sum.test.js`, and create a new file named `db-connection.test.js`.

Copy-paste the code below into your `db-coonection.test.js` file.

```
require("dotenv").config();
const mongoose = require("mongoose");
const ArticleService = require("../services/ArticleService");

describe("Connection", () => {
  beforeAll(async () => {
    await mongoose.connect(process.env.mongoURI, {
        useNewUrlParser: true,
        useCreateIndex: true,
```

```
            useUnifiedTopology: true,
        })
    });

    test("Retrieve article by Id", async () => {
        const id = "5ff2454f94eeee0a7acb5c30";
        const article =  await ArticleService.getArticlebyId(id);
        expect(article.title).toBe("This is another post example");
    });

    afterAll(async done => {
        mongoose.disconnect();
        done();
});

});
```

To test our `DB connection`, we have to initiate a connection to our `MongoDB` database then subsequently testing if the connection was successful by attempting to retrieve data from our `"articles"` collection. We are using the **Mocking Technique** to test if the article with the specified id is in our database. Since the `beforeAll()` is the block of code that runs before the rest of our code, it is the right place to actually perform the `DB` connection. This line of code; `expect(article.title).toBe("This is another post example");` checks if the article returned from our `DB` has the title "This is another post example"; Similarly the `afterAll()` function executes a code block after all the test have passed.

Testing the `apiGetAllArticles` endpoint

Create a new file called `get-all-articles.test.js` in the test directory, and copy-paste the code below.

```
require("dotenv").config();
const mongoose = require("mongoose");
const ArticleService = require("../services/ArticleService");

describe("Get all Articles", () => {
    beforeAll(async () => {
        await mongoose.connect(process.env.mongoURI, {
            useNewUrlParser: true,
            useCreateIndex: true,
            useUnifiedTopology: true,
        })
    });

    test("Get all Articles", async() => {
        const articles = await ArticleService.getAllArticles();
        expect(articles).toEqual(expect.arrayContaining(articles));
    });
```

```
    afterAll(async done => {
        mongoose.disconnect();
        done();
    });
})
```

To validate if the output of our getAllArticles() endpoint returns an array, we make use of the `expect(articles)`, `toEqual()`, and `expect.arrayContaining(Array)` function in `Jest`. Even as these functions come in handy, there's a great benefit to understand the logic behind their combination. Here, we are checking to see if the articles are returned from the database grouped in `Array`, what if there are no articles returned? The result will be an empty array `[]`. Open your terminal, `cd` into your `tdd-with-nodejs` directory, copy-paste the code below to run the test.

```
$ npm test
```

You should get a response similar to the screenshot below

Testing for CRUD operation

```
require("dotenv").config();
const mongoose = require("mongoose");
const ArticleService = require("../services/ArticleService");

describe("Should perform CRUD on article Service", () => {
    beforeAll(async() => {
        await mongoose.connect(process.env.mongoURI, {
            useNewUrlParser: true,
            useCreateIndex: true,
            useUnifiedTopology: true,
        })
    });

    test("Creat article", async() => {
        const newArticle = {
            title: "All the test must pass",
            body: "Should the test fail, we should work had to improve our code",
            avartar: "https://dev-to-uploads.s3.amazonaws.com/i/blaf4ke2xt3j08mlx4ca.png",
        }
        const article =  await ArticleService.createArticle(newArticle);
        expect(article).toEqual(expect.objectContaining(article));
    });
```

```
    test("Update article", async() => {
        const articleToUpdate = {
            title: "All the tests get passed",
            body: "Should the test fail, we should work had to improve our codebase",
            avartar: "https://dev-to-uploads.s3.amazonaws.com/i/blaf4ke2xt3j08mlx4ca.png",
        };
        const article = await ArticleService.updateArticle(articleToUpdate);
        expect(article).toEqual(expect.objectContaining(article));
    });

    test("Get article by Id", async() => {
        const articleId = "5ffcc8b0d7556519346f3bd8"
        const article = await ArticleService.getArticlebyId(articleId);
        expect(article).toEqual(expect.objectContaining(article));
    });

    test("Delete article", async() => {
        const articleId = "5ffcc8fcb6f631195c9a3529";
        const article =  await ArticleService.deleteArticle();
        expect(article).toEqual(expect.objectContaining(article));
    })

    afterAll(async (done) => {
        mongoose.disconnect();
        done()
    })
})
```

Here we have put together all the testing techniques we have explored so far to perform a complete test of the `article endpoint`. Type the following command on your terminal for mac users or command-prompt for windows users.

```
$ npm test
```

If you got everything set up correctly, you should have a response on your terminal similar to the one below:

Conclution

We have been able to perform `unit-testing` all thanks to our `MVC` and `Layered Structure` design pattern we explored in our previous tutorial. The benefits of using clean architecture are enormous, it

helps you to write easy readable, testable, and efficient code. Feel free to dive deeper into the Jest official Documentation, the developers at Facebook have done a lot of work there.

The source-code for this series can be accessed here

Thank you for reading, I will love to hear from you, please drop a comment.

Top comments (1)

Su Wai Phyo • Sep 3 '21

```
const articles = await ArticleService.getAllArticles();
expect(articles).toEqual(expect.arrayContaining(articles));
```

why articles toequal articles, this is no sense.

Code of Conduct • Report abuse

Emmanuel Etukudo

Software engineer with a passion for new technologies.

JOINED
Feb 14, 2020

More from Emmanuel Etukudo

Build automatic URL shortener with react and Emly
#react #javascript #webdev #reactnative

How to implement an API using Vuejs & Axios
#vue #axio #javascript #tutorial

Test-Driven Development with Nodejs, Express, Mongoose & Jest
#mongodb #node #javascript #testing