

Unit Testing the eCommerce Applications - 2



Class Agenda

- We will start implementing unit test in our ecommerce application
- We will also do some setup and configuration to implement integration test.
- We will then start writing integration test.

Educator Introduction

Unit test for Controller

Install the required package

- Run the below command to install jest in our project
- `npm install -D cross-env jest`

What is Interceptors file and Why it's needed?

While testing a controller we need to make sure that the request object contains desired parameters and body object the test different method and moreover the response object container the desired body or message or status code we are anticipating in the test case.

So, to complete this requirement we used jest mock functions to create an object to request and response with required keys in it like body and params in request and status, send and json in response to pass as parameters to each controller in the test case and expect the desired result to properly verify and pass the test case.

- **How it is implemented:**
- Here we are importing **mockRequest** and **mockResponse** from **interceptor.js**.
- We are using **mockRequest** to mock the request for our **controllers** by using **jest.fn().mockReturnValue(req)** and store the mocked value inside request body and request params properties of the request object and then return the request object.
- The request is generally referred to as **req** and response are referred as **res**.
- When you observe carefully, you can see we are just returning an **empty request object** while mocking for body, params, and **empty response object** while mocking for send, status, json and also returning it as the result of **mockRequest()** and **mockResponse()**.
- This is done so that we can call the **expect()** with req and res as arguments and set it to the value that we want for testing the mock using **toHaveBeenCalledWith()** on **expect()**, now the req and res can be set as anything that we need so it gives us the leverage of setting it to error or actual payload as well. We will see its implementation in detail below.

`interceptor.js` contains two properties
/functions as below.

```
JS interceptor.js > [?] <unknown>
✓ module.exports = {
  ...
  ✓ mockRequest: () => {
    const req = {}
    req.body = jest.fn().mockReturnValue(req)
    req.params = jest.fn().mockReturnValue(req)
    return req
  },

  ✓ mockResponse: () => {
    const res = {}
    res.send = jest.fn().mockReturnValue(res)
    res.status = jest.fn().mockReturnValue(res)
    res.json = jest.fn().mockReturnValue(res)
    return res
  },
}
```


product.controller.js

Implementing test for product controller

Step 1: Imports and lifecycle method and test data

- models are where database queries, stored procedures are stored, and data is modified based on the operations requested/called on the model by the controller.
- So, we import the **product** model object from model and store it in a variable **productModel**.
- beforeEach** is a lifecycle method which we are using here to reset our **req** and **res** object before every test, which we have been doing anyways but by this way we reduce the repetition on use of extra variables here.
- And, **testPayload** will be the test data which we will be passing through **req** object and test it by expecting the response from controller methods.

```
const db = require('../models');
const ProductModel = db.product;
const ProductController = require('../controllers/product.controller');
const { mockRequest, mockResponse } = require('../interceptor');

let req, res;

beforeEach(() =>{
  req = mockRequest();
  res = mockResponse();
})

const testPayload =
{
  name: "Sony Bravia",
  description: "This is an amazing TV",
  cost: 10000,
  categoryId: 1
};
```

Step 2: Testing create method

- Here we have used the **describe()** function, which accepts string- '**Product controller create method**' and a **callback** function.
- **it()** function which accepts string- '**should return success message with product details**' and an **async() callback** function.
- **jest.spyOn()** method is used for spying/observing the **method/function** calls on an **object** without changing the method.
- Here we spy on the **create** method of the **productModel** object imported from the model's file **productModel**.
- We are calling **mockImplementation()** on the **create()**, which accepts a callback. As we are testing for a **Success** scenario, we are returning a Promise object and by resolving it.

```
    mockImplementation(  
      (testPayload) => new Promise(function (resolve, reject) {  
        resolve(testPayload);  
      });  
    ));
```

- Now we are storing the **mockRequest** and **mockResponse** from **interceptor.js** as **req** and **res**.
- We insert the **testPayload** in **req** object body like this:
req.body = testPayload;
- We call **create()** on **productController** object with **await** and wait for it to return either error or the result response.
- **productController.create(req, res)**
- (Refer **product.controller.js** for better understanding and explanation)
- Finally we expect the **spy** to be called using **toHaveBeenCalled()** on the **spy** object.
- **toHaveBeenCalled()** ensures our function is called/executed.
- Then we set the response **status** as **201** and response **send** as

```
{
  name: "Sony Bravia",
  description: "This is an amazing TV",
  cost: 10000,
  categoryId: 1
};
```
- by calling **toHaveBeenCalled()** and passing the status and send parameters to it.

```
describe('Product controller create method', () => {
  Run | Debug
  it('should return success message with product details',
  async () => {
    const spy = jest.spyOn(ProductModel, 'create').
    mockImplementation(
      (testPayload) => new Promise(function (resolve,
      reject) {
        resolve(testPayload);
      }));

    req.body = testPayload;

    await ProductController.create(req, res);

    expect(spy).toHaveBeenCalled();
    expect(res.status).toHaveBeenCalledWith(201);
    expect(res.send).toHaveBeenCalledWith(testPayload)
    ;
  });
});
```

- Here we have used the **describe()** function, which accepts string- '**Product controller create method**' and a **callback** function.
- **it()** function which accepts string- '**should return error**' and an **async()** **callback** function.
- **jest.spyOn()** method is used for spying/observing the **method/function** calls on an **object** without changing the method.
- Here we spy on the **create** method of the **productModel** object imported from the model's file **productModel**.
- We are calling **mockImplementation()** on the **create()**, which accepts a callback. As we are testing for an **Error** scenario, we are returning a Promise object and by rejecting it.

```
    mockImplementation(  
      () => Promise.reject(new Error("This is an error"))  
    );
```

- Now we are storing the **mockRequest** and **mockResponse** from **interceptor.js** as **req** and **res**.
- We call **create()** on **productController** object with **await** and wait for it to return either error or the result response.
- **productController.create(req, res)**
- (Refer product.controller.js for better understanding and explanation)
- Finally we expect the **spy** to be called using **toHaveBeenCalled()** on the **spy** object.
- **toHaveBeenCalled()** ensures our function is called/executed

```
describe('Product controller create method', () => {  
  Run | Debug  
  it('should return error', async () => {  
    const spy = jest.spyOn(ProductModel, 'create').  
      mockImplementation(  
        () => Promise.reject(new Error("This is an error"))  
      );  
  
    await ProductController.create(req, res);  
  
    expect(spy).toHaveBeenCalled();  
  });  
});
```

Step 3: Testing findOne method

- Here we spy on the **findByPk** method of the **productModel** object imported from the model's file **productModel**.
- We are calling **mockImplementation()** on the **create()**, which accepts a callback. As we are testing for a **Success** scenario, we are returning a Promise object and by resolving it.

```
mockImplementation(  
  (testPayload) => new Promise(function (resolve, reject) {  
    resolve(testPayload);  
  }  
));
```


- Now we are storing the **mockRequest** and **mockResponse** from **interceptor.js** as **req** and **res**.
- We insert the **product id** in **req** object body like this:
req.params.id = 1;
- We call **findOne()** on **productController** object with **await** and wait for it to return either error or the result response.
productController.findOne(req, res)
(Refer **product.controller.js** for better understanding and explanation)
- Finally we expect the **spy** to be called using **toHaveBeenCalled()** on the **spy** object.
- **toHaveBeenCalled()** ensures our function is called/executed.
- Then we set the response **status** as **200** and response **send** as

```
{
  name: "Sony Bravia",
  description: "This is an amazing TV",
  cost: 10000,
  categoryId: 1
};
```
- by calling **toHaveBeenCalled()** and passing the status and send parameters to it.

```
describe('Product controller findOne method', () => {
  Run | Debug
  it('should return product details', async () => {
    const spy = jest.spyOn(ProductModel, 'findByPk').
    mockImplementation(
      () => new Promise(function (resolve, reject) {
        resolve(testPayload);
      }));
    req.params.id = 1;
    await ProductController.findOne(req, res);

    expect(spy).toHaveBeenCalled();
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.send).toHaveBeenCalledWith(testPayload);
  });
});
```

```
describe('Product controller findOne method', () => {  
  Run | Debug  
  it('should return error', async () => {  
    const spy = jest.spyOn(ProductModel, 'findByPk').  
      mockImplementation(  
        () => new Promise(function (resolve, reject) {  
          reject(new Error('This is an error'));  
        }));  
    req.params.id = 1;  
    await ProductController.findOne(req, res);  
  
    expect(spy).toHaveBeenCalled();  
  });  
})
```

- Here we spy on the **findByPk** method of the **productModel** object imported from the model's file **productModel**.
- We are calling **mockImplementation()** on the **create()**, which accepts a callback. As we are testing for an **Error** scenario, we are returning a Promise object and by rejecting it.

```
mockImplementation(  
  () => Promise.reject(new Error("This is an error"))  
);
```

- Now we are storing the **mockRequest** and **mockResponse** from **interceptor.js** as **req** and **res**.
- We insert the **product id** in **req** object body like this:
- **req.params.id = 1;**
- We call **findOne()** on **productController** object with **await** and wait for it to return either error or the result response.
- **productController.findOne(req, res)**
- (Refer **product.controller.js** for better understanding and explanation)
- Finally we expect the **spy** to be called using **toHaveBeenCalled()** on the **spy** object.
- **toHaveBeenCalled()** ensures our function is called/executed.

Step 4: Testing update method

- Here we spy on the **findByPk** method of the **productModel** object imported from the model's file **productModel**.
- We are calling **mockImplementation()** on the **findByPk()** and **update()**, , which accepts a callback. As we are testing for a **Success** scenario, we are returning a Promise object and by resolving it.

```
      mockImplementation(  
        (testPayload) => new Promise(function (resolve, reject) {  
          resolve(testPayload);  
        })  
      ));  
    mockImplementation(  
      () => new Promise(function (resolve, reject) {  
        resolve(testPayload);  
      })  
    ));
```

- Refer to the `update()` method implementation of `product.controller.js`
- Now we are storing the **mockRequest** and **mockResponse** from **interceptor.js** as **req** and **res**.
- We insert the **product id** in **req** object body like this:
- **req.params.id = 1;**
- We call **update()** on **productController** object with `await` and wait for it to return either error or the result response.
- **productController.findOne(req, res)**
- (Refer `product.controller.js` for better understanding and explanation)
- Finally we expect the **spy1** and **spy2** to be called using **toHaveBeenCalled()** on the **spy1** and **spy2** object.
- **toHaveBeenCalled()** ensures our function is called/executed.

```
describe('Product controller update method', () => {
  Run | Debug
  it('should return product details', async () => {
    const spy1 = jest.spyOn(ProductModel, 'findByPk').
    mockImplementation(
      () => new Promise(function (resolve, reject) {
        resolve(testPayload);
      }));
    const spy2 = jest.spyOn(ProductModel, 'update').
    mockImplementation(
      (data) => new Promise(function (resolve, reject) {
        resolve("product updated");
      }));

    req.params.id = 1;
    await ProductController.update(req, res);

    expect(spy1).toHaveBeenCalled();
    expect(spy2).toHaveBeenCalled();
  });
});
```

- Here we spy on the **update** method of the **productModel** object imported from the model's file **productModel**.
- We are calling **mockImplementation()** on the **update ()**, which accepts a callback. As we are testing for an **Error** scenario, we are returning a Promise object and by rejecting it.

```
mockImplementation(
    () => Promise.reject(new Error("This is an error"))
);
```

- Now we are storing the **mockRequest** and **mockResponse** from **interceptor.js** as **req** and **res**.
- We call **update ()** on **productController** object with **await** and wait for it to return either error or the result response
- **productController.update(req, res)**
- (Refer **product.controller.js** for better understanding and explanation)
- Finally we expect the **spy** to be called using **toHaveBeenCalled()** on the **spy** object.
- **toHaveBeenCalled()** ensures our function is called/executed.

```
describe('Product controller update method', () => {
  Run | Debug
  it('should return error', async () => {
    const spy = jest.spyOn(ProductModel, 'update').
    mockImplementation(
      () => new Promise(async function (resolve,
        reject) {
          await reject(new Error('This is an error'));
        }));

    req.params.id = 1;
    await ProductController.update(req, res);

    expect(spy).toHaveBeenCalled();
  });
});
```

Step 5: Testing delete method

- Here we spy on the **destroy** method of the **productModel** object imported from the model's file **productModel**.
- We are calling **mockImplementation()** on the **destroy()**, which accepts a callback. As we are testing for a **Success** scenario, we are returning a Promise object and by resolving it.

```
mockImplementation(  
  (data) => new Promise(function (resolve, reject) {  
    resolve("product deleted");  
  });  
));
```

- Now we are storing the **mockRequest** and **mockResponse** from **interceptor.js** as **req** and **res**.
- We insert the **product id** in **req** object body like this:
req.params.id = 1;
- We call **delete()** on **productController** object with **await** and wait for it to return either error or the result response.

- `productController.delete(req, res)`
- Finally we expect the `spy` to be called using `toHaveBeenCalled()` on the `spy` object.
- `toHaveBeenCalled()` ensures our function is called/executed.
- Then we set the response `status` as `200` and response `send` as

```

    {
      message: "Successfully deleted the
product"
    }

```

- by calling `toHaveBeenCalledWith()` and passing the status and send parameters to it.

```

describe('Product controller delete method', () => {
  Run | Debug
  it('should return success message', async () => {
    const spy = jest.spyOn(ProductModel, 'destroy').
    mockImplementation(
      (data) => new Promise(function (resolve, reject) {
        resolve("product deleted");
      }));

    req.params.id = 1;
    await ProductController.delete(req, res);

    expect(spy).toHaveBeenCalled();
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.send).toHaveBeenCalledWith( {
      message: "Successfully deleted the product"
    });
  });
});

```


- Here we spy on the **destroy** method of the **productModel** object imported from the model's file **productModel**.
- We are calling **mockImplementation()** on the **destroy ()**, which accepts a callback. As we are testing for an **Error** scenario, we are returning a Promise object and by rejecting it.

```
mockImplementation(
  () => new Promise(function (resolve, reject) {
    reject(new Error('This is an error'));
  });
```

- Now we are storing the **mockRequest** and **mockResponse** from **interceptor.js** as **req** and **res**.
- We call **delete ()** on **productController** object with **await** and wait for it to return either error or the result response.
- **productController.update(req, res)**
- (Refer **product.controller.js** for better understanding and explanation)
- Finally we expect the **spy** to be called using **toHaveBeenCalled()** on the **spy** object.
- **toHaveBeenCalled()** ensures our function is called/executed.

```
Run | Debug
describe('Product controller delete method', () => {
  Run | Debug
  it('should return error', async () => {
    const spy = jest.spyOn(ProductModel, 'destroy').
    mockImplementation(
      () => new Promise(function (resolve, reject) {
        reject(new Error('This is an error'));
      });
    await ProductController.delete(req, res);
    expect(spy).toHaveBeenCalled();
  });
});
```

Test Result

```
PASS tests/controllers/product.controller.test.js
  Product controller create method
    ✓ should return success message with product details (18 ms)
    ✓ should return error (5 ms)
  Product controller findOne method
    ✓ should return product details (2 ms)
    ✓ should return error (1 ms)
  Product controller update method
    ✓ should return product details (1 ms)
    ✓ should return error
  Product controller delete method
    ✓ should return success message (1 ms)
    ✓ should return error (1 ms)
  Product controller getProductsUnderCategory method
    ✓ should return success message (2 ms)
    ✓ should return error (1 ms)

Test Suites: 1 passed, 1 total
Tests:       10 passed, 10 total
Snapshots:   0 total
Time:        2.222 s, estimated 3 s
Ran all test suites matching /d:\Mohit\Relevel\Backend\ eCommerce\tests\controllers\product.controller.test.js/i.
PS D:\Mohit\Relevel\Backend\ eCommerce>
```

Implementing Integration Test

Let us now start with first integration testing in our first

Database Setup

- For testing routes, we need to have our database connected but, we can afford to manipulate our production or development database to change data in our tables and also the test case might fail due to changes occurring inside development environment. So, we create a testing database here where we can have a full control and can truncate table before every test run without the risk of losing the data.
- For this we just need to connect to our mysql workbench or mssql shell as you prefer and run below command:
- **create database ecom_test_db;**

Testing products routes:

- Step 1: Install all the required packages:
-
- **npm install -D cross-env supertest**
- **npm install sequelize-cli**
- sequelize cli will help us to run commands from terminal, so we can add scripts in our package.json file and run it.
- cross-env package is used to access process object from cli like `NODE_ENV` property to set into to test to used test database while testing and
- supertest is yet another handy testing tool which is explained in detail below

Step 2: Configure testing script in package.json file:

- **Pretest:** The pretest is an npm script that is automatically invoked when the npm test command is invoked. We'll hook in the command to change the environment to test and refresh the database before each test runs.
- **migrate:reset:** This command will be responsible for refreshing the database before each test runs.
- **cross-env:** an operating system agnostic package for setting environment variables. We used it to set the NODE_ENV to test so that our test can use the test database.
- **--testTimeout flag:** This increases the default timeout of Jest which is 5000ms. This is important since the test runner needs to refresh the database before running the test.
- **sequelize-cli:** sequelize-cli package is used to run sequelize commands from command line interface i.e., db:migrate which is used in migrate command to reset the database.

Now edit the scripts for package.json as shown below:

```
"scripts": {  
  "start": "node server",  
  "migrate": "npx sequelize-cli db:migrate",  
  "migrate:reset": "npx sequelize-cli db:migrate:undo:all &  
    & npm run migrate",  
  "test": "cross-env NODE_ENV=test jest --testTimeout=10000  
    --coverage",  
  "pretest": "cross-env NODE_ENV=test npm run migrate:reset"  
},  
"author": "Vishnu Vignesh"
```

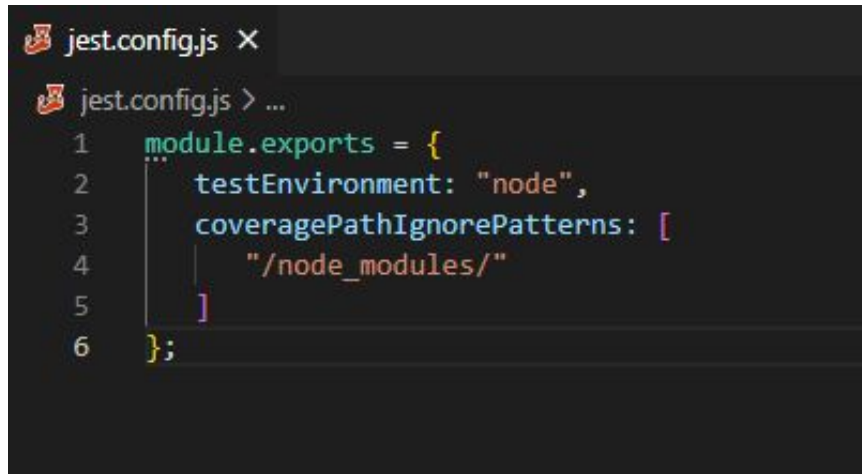
Step 3: Create config.json file inside config folder with all the DB connection details:

- When we run cli commands to reset our database using sequelize cli it checks for the connection config in below mentioned path:
- **config/config.json**
- Here, we have made a separate database for testing environment because while testing our api endpoints i.e., routes testing we don't want any test data to interfere with our development or production database and also to make sure the testing environment is always consistent according to test case, which when used with other environments might change expected result.

```
{ } config.json x
config > { } config.json > ...
1  {
2    "development": {
3      "username": "root",
4      "password": null,
5      "database": "database_development",
6      "host": "127.0.0.1",
7      "dialect": "mysql",
8      "pool": {
9        "max": 5,
10       "min": 0,
11       "acquire": 30000,
12       "idle": 10000
13     }
14   },
15   "test": {
16     "username": "root",
17     "password": "Mohit@19",
18     "database": "ecommerce_test_db",
19     "host": "127.0.0.1",
20     "dialect": "mysql",
21     "pool": {
22       "max": 5,
23       "min": 0,
24       "acquire": 30000,
25       "idle": 10000
26     }
27   },
28   "production": {
29     "username": "root",
30     "password": null,
31     "database": "database_production",
32     "host": "127.0.0.1",
33     "dialect": "mysql"
34   }
35 }
```


Step 4: Create jest.config.js file:

- When we run cli commands to reset our database using sequelize cli it checks for the connection config in below mentioned path:
- **config/config.json**
- Here, we have made a separate database for testing environment because while testing our api endpoints i.e., routes testing we don't want any test data to interfere with our development or production database and also to make sure the testing environment is always consistent according to test case, which when used with other environments might change expected result.

A screenshot of a code editor window titled 'jest.config.js X'. The editor shows the following JavaScript code:

```
1  module.exports = {  
2    testEnvironment: "node",  
3    coveragePathIgnorePatterns: [  
4      "/node_modules/"  
5    ]  
6  };
```

Step 5: Update models' index.js file to configure testing database configuration:

- While testing our api endpoints this is where our application will pick up the db connection details and by updating the config we will make sure the connection is made to testing database only while running test cases.

```
const env = process.env.NODE_ENV || 'development';
const config = require("../configs/db.config")[env];
const Sequelize = require("sequelize");

console.log("ENV: " + env);
```

Step 6: Update db.config.js file as well with test connections:

```
JS db.config.js X
configs > JS db.config.js > [0] <unknown>
1  module.exports = {
2      development: {
3          HOST: "localhost",
4          USER: "root",
5          PASSWORD: "Welcome1",
6          DB: "ecom_db",
7          dialect: "mysql",
8          pool: {
9              max: 5,
10             min: 0,
11             acquire: 30000, //max time in ms that a pool will
12                           try to get connection before throwing error
13             idle: 10000 // maximum time in ms that a
14                           connection can be idle before being released
15         },
16         test: {
17             HOST: "localhost",
18             USER: "root",
19             PASSWORD: "Mohit@19",
20             DB: "ecom_test_db",
21             dialect: "mysql",
22             pool: {
23                 max: 5,
24                 min: 0,
25                 acquire: 30000, //max time in ms that a pool will
26                               try to get connection before throwing error
27                 idle: 10000 // maximum time in ms that a
28                               connection can be idle before being released
29             }
30         },
31     },
32 }
```

Step 7: Create migrations folder in root directory

Make sure to create a “**migrations**” folder in the parent directory as one of the node_modules dependencies will need this.

Step 8: Separating server.js file

- Create a new file in root named app.js and move all the code from server.js to express.js
- Except below show from server.js file.
- Also add export line in app.js to use it in testing. Since supertest while testing make the application run on another port to test it keeping app.listen code will make app to run on different ports or maybe clash on the same one. So, we separated the code to avoid this issue.



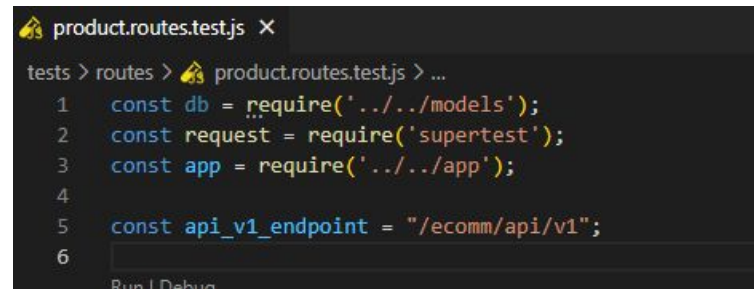
```
JS server.js X
JS server.js > ...
1  const app = require('./app');
2
3  //Starting the server
4  app.listen(serverConfig.PORT, () => {
5    console.log(`Application started on the port no : ${
      {serverConfig.PORT}}`);
6  });
```

Step 9: Create product.routes.test.js file and add below test codes

superset:

- It is a library that allows us to test the node.js HTTP servers
- It is used to test HTTP requests, request headers etc.

Here **app** is nothing but the **express()** function that we have imported from the **app.js** file, refer app.js for this.



```
product.routes.test.js X
tests > routes > product.routes.test.js > ...
1  const db = require('../models');
2  const request = require('supertest');
3  const app = require('../app');
4
5  const api_v1_endpoint = "/ecomm/api/v1";
6
```

```

describe('Post details of products endpoint', () => {
  Run | Debug
  it('should add new product', async () => {
    const res = await request(app)
      .post(api_v1_endpoint + '/categories')
      .send({
        name: "Electronics",
        description: "Category on eletronic items",
      })
    expect(res.statusCode).toEqual(201);
  })
  Run | Debug
  it('should add new product', async () => {
    const res = await request(app)
      .post(api_v1_endpoint + '/products')
      .send({
        name: "Sony Bravia",
        description: "This is an amazing TV",
        cost: 10000,
        categoryId: 1
      })
    expect(res.statusCode).toEqual(201);
  })
})

```

```
Run | Debug
describe('Get details of products endpoint', () => {
  Run | Debug
  it('should get details of product', async () => {
    const res = await request(app)
      .get(api_v1_endpoint + '/products')
    expect(res.statusCode).toEqual(200);
  })
})

Run | Debug
describe('Get details of one product endpoint', () => {
  Run | Debug
  it('should get details of one product', async () => {
    const res = await request(app)
      .get(api_v1_endpoint + '/products/2')
    expect(res.statusCode).toEqual(200);
  })
})
```


Test Result

```
de sequelize.log (node_modules/sequelize/src/sequelize.js:1100:15)

PASS tests/routes/product.routes.test.js
  Post details of products endpoint
    ✓ should add new product (441 ms)
    ✓ should add new product (40 ms)
  Get details of products endpoint
    ✓ should get details of product (32 ms)
  Get details of one product endpoint
    ✓ should get details of one product (58 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        2.526 s, estimated 4 s
Ran all test suites matching /d:\Mohit\Relevel\Backend\ eCommerce\tests\routes\product.routes.test.js/i.
```

Intro to code coverage:

What is Code Coverage?

Code coverage is a term used in software testing to describe how much of the source code is covered by a testing suite or list test cases. It is also known as test coverage.

Using code coverage development teams can provide reassurance that their programs have been broadly tested for bugs and should be relatively error-free.

Intro to code coverage:

Implementation

This is how we can check the coverage of our application code.

Run the below command in cmd

npm test --coverage or,

update our test script in package .json file to:

```
"test": "cross-env NODE_ENV=test jest  
--testTimeout=10000 --coverage",
```

This will execute the unit tests and give our application code coverage.

Code coverage is considered one of the important components and is kind of a quality check that demonstrates and ensures the code is production-ready for deployment. It's best to have code coverage of more than 75%.

db.config.js	100	100	100	100	
server.config.js	100	50	100	100	1
eCommerce/controllers	52.69	30.76	42.62	53.01	
auth.controller.js	69.69	50	60	71.87	22-30,36,53,62,74-77
cart.controller.js	21.62	0	0	21.62	18-27,37-65,85-98
category.controller.js	33.33	0	11.76	33.33	31-32,44-58,68-73,88-107,117-130
product.controller.js	73.77	31.25	78.26	73.77	51,58,67,75,89,133,194-219
eCommerce/middlewares	28.4	0	0	29.06	
authjwt.js	32	0	0	33.33	7-22,27-38
index.js	100	100	100	100	
requestValidator.js	18.18	0	0	18.18	6-12,21-48,55-77
verifySignUp.js	26.92	0	0	28	6-33,38-48
eCommerce/models	100	50	100	100	
cart.model.js	100	100	100	100	
category.model.js	100	100	100	100	
index.js	100	50	100	100	12
product.model.js	100	100	100	100	
role.model.js	100	100	100	100	
user.model.js	100	100	100	100	
eCommerce/routes	93.75	100	80	93.75	
auth.routes.js	75	100	50	75	7-11
cart.routes.js	100	100	100	100	
category.routes.js	100	100	100	100	
product.routes.js	100	100	100	100	
eCommerce/tests	100	100	100	100	
interceptor.js	100	100	100	100	

Test Suites: 3 passed, 3 total					
Tests: 16 passed, 16 total					
Snapshots: 0 total					
Time: 6.477 s					
Ran all test suites.					
PS D:\Mohit\Relevel\Backend\ eCommerce >					

MCQ

1. Which test package is used to test api endpoints of our application

- A. supertest
- B. jest
- C. mocha
- D. None of the above

2. toHaveBeenCalledWith() is used to:

- A. Accepts a function that will be used as an implementation of the mock for every call to the mocked function.
- B. Accepts a value that will be returned for one call to the mock function.
- C. Accepts a value that will be set as a mock value for request or response.
- D. Accepts a value that will be returned for every call to the mock function.

3. mockImplementation is used to.

- A. Accepts a function that will be used as an implementation of the mock for every call to the mocked function.
- B. Accepts a value that will be returned for one call to the mock function.
- C. Accepts a function that will be used as an implementation of the mock for one call to the mocked function.
- D. Accepts a value that will be returned for every call to the mock function.

4. Jest uses _____ and _____ matchers to test a value with exact quality.

- A. toBeTruthy, toBeFalsy
- B. toBeEqual , toMatch
- C. toBe , toEqual
- D. toBeUndefined, toBeDefined

5. Which lifecycle method gets called before the execution of a test case

- A. before
- B. beforeAll
- C. beforeEach
- D. None of the options

Practice Problems

- Write tests for `getProductsUnderCategory` method in product controller.
- Write unit tests for auth controller.

Upcoming Class

In next class we will get to know about deployments and try to deploy our application on Heroku.

Thank You!