CO328 PROJECT REPORT
ONLINE GROCERY DELIVERY SERVICE

GROUP 14

E/15/330

E/15/366

E/15/373

Introduction

Nowadays the lifestyle of the people is different. People feel uncomfortable and time consuming going to crowded markets and especially after Covid 19 pandemic people have realized the importance of online delivery platforms. So, delivering the goods at their doorstep will save a lot of time for them as well as it will ensure their safety. This online grocery delivery service system primarily delivers grocery goods, home essentials and food supplies found in convenience stores, to its customers. A customer can order the goods from this online service from several stores and have to submit their address then the delivery will be delivered to the customer by the stores.

When the customer visits the ordering webpage, they are presented with an interactive and up-to-date options of groceries, complete with all available options and dynamically adjusting prices based on the selected options. After making a selection, the item is then added to their cart, which the customer can review the details of at any time before confirming the order. This provides instant visual confirmation of what was selected and ensures that items in the order are, in fact, what was intended.

This system also greatly lightens the load on the store/super market's end, as the entire process of taking orders is automated. Once an order is placed on the webpage, it is entered into the database and then retrieved, in pretty much real-time, by the admin page of the store which can be registered and maintained by the store employees. Within this admin page, all items in the order are displayed, along with their corresponding options and delivery details, in a concise and easy to read manner.

Background

This Case study looks at the problem of setting up this system. In existing system there are few problems:

• For placing any orders customers have to visit stores or supermarkets to buy product items and to pay for the goods they have to wait in a queue. In this method the customer's time is wasted.

• While placing an order over the phone, customers lack knowledge of available goods and their prices, lack of visual confirmation also needed in some products.

Hence, to solve this issue, this online grocery delivery service is designed. The main advantage of my system is that it greatly simplifies the ordering process for both the customer and the supermarkets/groceries.

Anticipated Benefits are:

1. This will minimize the number of employees at the back of the counter and the employees who assist the customers. These employees can be used to deliver the goods to customers.

2. Later on supermarkets don't need to display products for customers in a physical place can completely rely on this system and they can deliver them directly from the warehouse.

3. The system will be less probable to make mistakes, since it’s a machine.

4. This will avoid long queues at the counter due to the speed of execution and achieve maximum throughput.

Requirement analysis

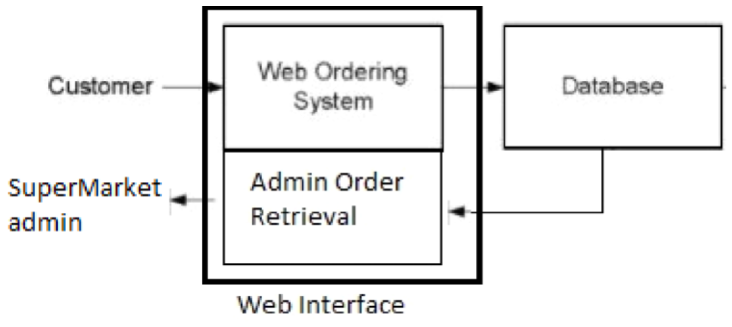
There are two types of users for this system

1.Customers who order groceries by this system.

2.Shop owners who display their goods by this system.

System Perspective:

System model



The structure of the system can be divided into 2 main logical components:

• Web Ordering System- provides the functionality for customers to place their order and supply necessary details.

• Store admin management module- Allows supermarket administrator to display products and keep track of all orders placed. This component takes care of order retrieving and displaying order information.

Web Ordering System Module

Web ordering module is a part of the whole web interface where there is a link for customer access.This module provides the functionality for customers to place their order and supply necessary details.

Users of the system, must be provided the following functionality:

• Create an account.

• Manage their account.

• Log in to the system.

• Select an item from the display.

• Add an item to their current order.

• Review their current order.

• Remove an item/remove all items from their current order.

• Provide payment/delivery details.

• Place an order.

• Receive confirmation.

• View order placed.

All users of the system, are provided with below options:

Register, Login, Home, Product display, Sorting products by price/recently added, Search for super market/store, Select product, Cart, Add quantity, Add delivery details, Add payment methods, Place order

Store admin management module

It is designed to be used only by grocery administrators of each groceries,this module is the another part of web interface and provides the following functions:

• Create an account for their business.

• Manage their account.

• Log in to the system.

• Add/update/delete product category to/from the display.

• Add /update/delete item to/from the display.

• Update price for a given item.

• Update additional information (description, photo, etc.) for a given item.

• Retrieve new orders from the database.

• Display the orders in an easily readable, graphical way.

All administrator of the system, are provided with below options:

Register, Log in, Home, Orders display, Customer delivery details, Add/update/delete products and details.

The web interface can be accessed using IE 10.0 and above, Firefox 31 and above safari and Google Chrome

Non-functional Requirements

The server (Node js & Express) frontend application (React) and database (mongodb) can handle the connections when hosted on the LAN using a personal computer but when the requests are increasing the server needs more cpu power. This website loads faster because of the React framework used in front end the virtual DOM on this framework make this happen. This system can be deployed in IAAS platforms like AWS to achieve availability, maintainability and scalability.

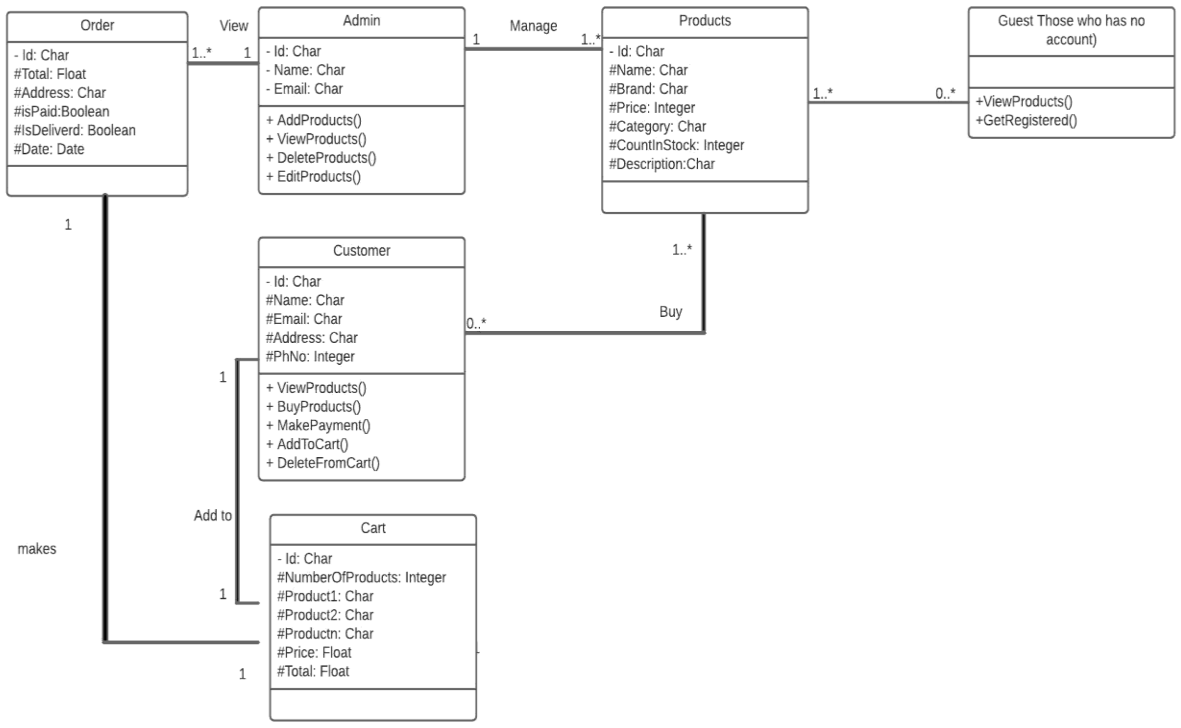
The customer must create a user account and business owners also have to create an account for their account and only the authenticated person can access their accounts. The website uses cookies to remember stateful information like details on cart and for sending and receiving authentication information it uses JWT.

Design of the project

Class Diagram

The purpose of this diagram is to introduce some common terms, "dictionary" for online shopping - Customer, Admin, Products, Cart, Order, Guest and relationships between them. It could be used as a common ground between business analysts and software developers.

Each customer has a unique id. A Customer can makes unique payment and buy many products. Every time a customer buys a product it is added to the cart and all the items on cart will be ordered. Guest can view products, but without login he can’t access the shopping cart. Admin can Manage many products and admin can view list of customer’s orders.



Activity Diagram

An activity diagram consists of activities and transitions, showing the flow of control from activity to activity as shown in Figure 1 for the functionality ‘Customer does Online Shopping’. Customer should login to the system before proceed to checkout. Customer can browse or search products, view products, add it to shopping cart, cancel the order or confirm order and edit profile, change password. Each action is represented using an activity construct and the flow of control from one activity to another, represented using arrows (edges). Activity diagrams can be used to model both sequential and concurrent activities.

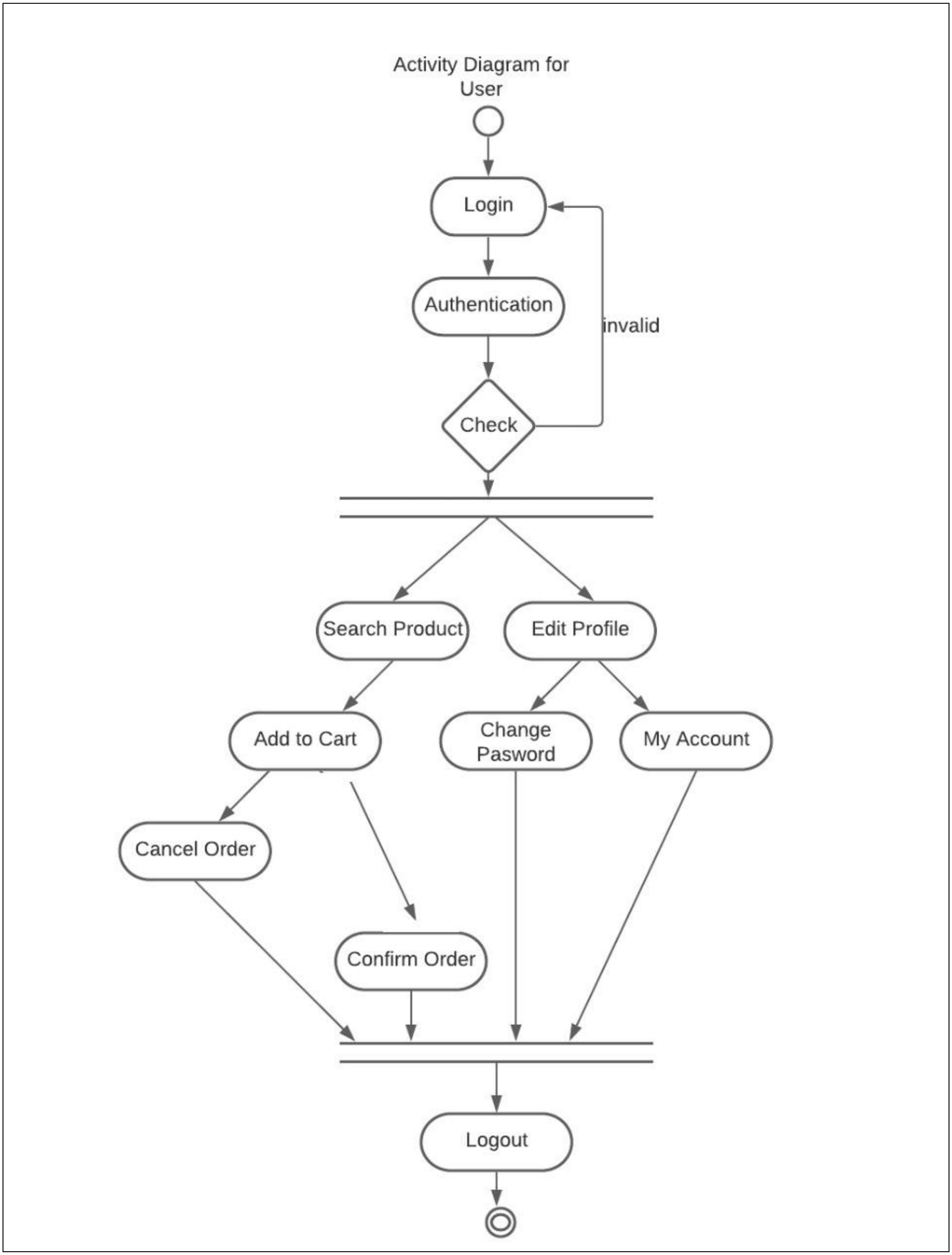

Activity Diagram for User
Login
Authentication
invalid
Check
Search Product
Edit Profile
Add to Cart
Change Pasword
My Account
Cancel Order
Confirm Order
Logout

FIGURE 1

Figure 2 explains about the activity diagram of Shop. Shop should login to the system before proceed to checkout. Shop can add products, view products, edit/ delete products and can view the customer orders.

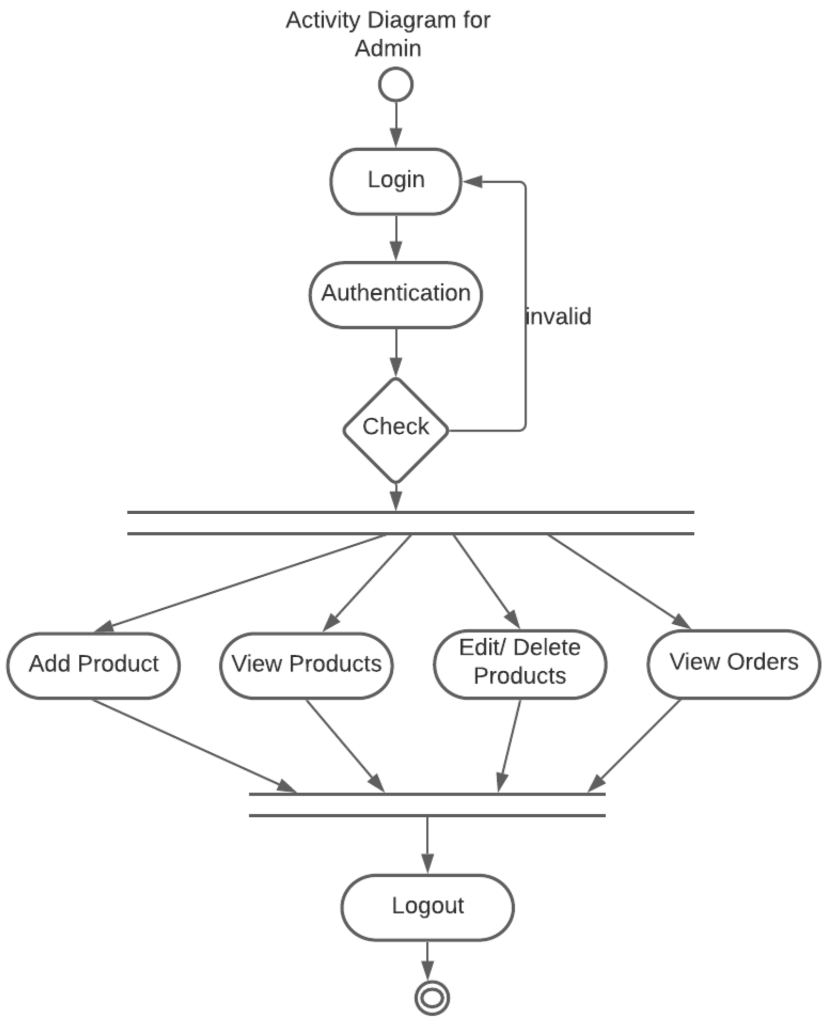


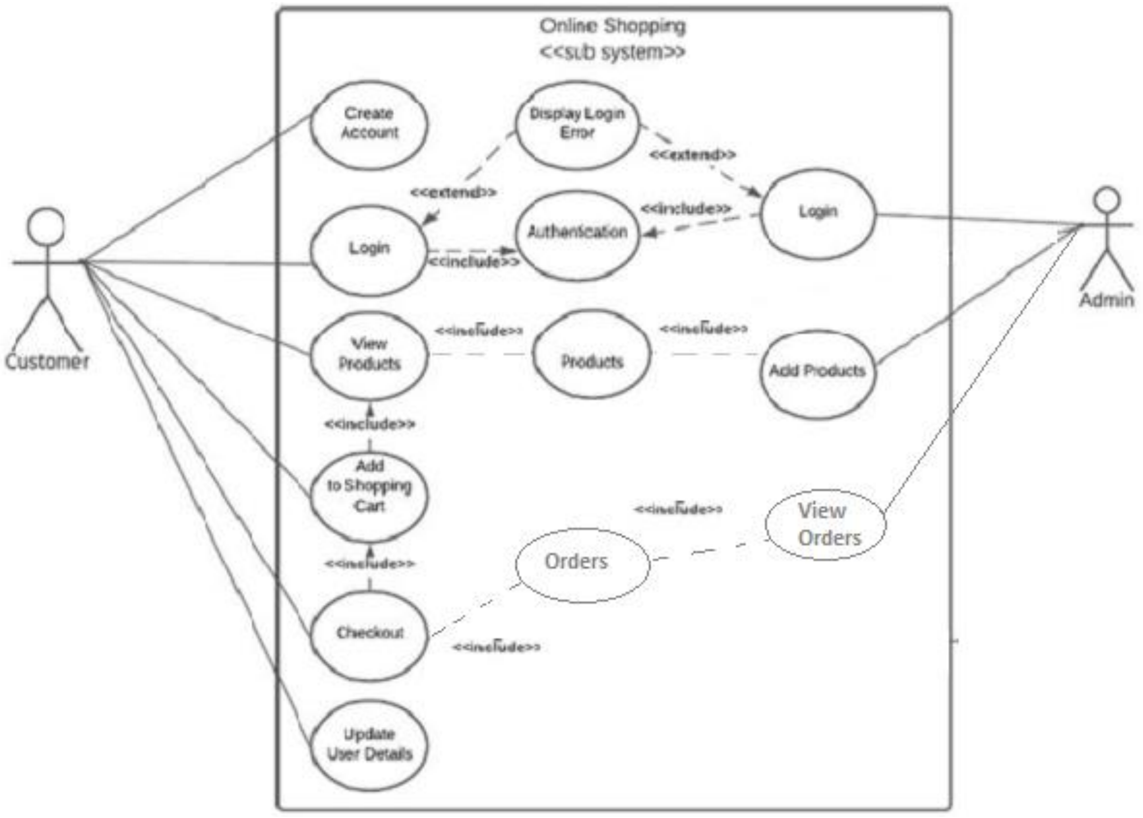
FIGURE 2

Use Case Diagram

Customer actor uses some website to make purchases online. Top level use cases are Create Account, Login, View Products, Add to Shopping Cart , Check Out and Update User Details. Create Account use case allows customers to register on the web site. Authentication use case is included in the Login, because it required the customer to be authenticated. View Products use case could be used by customers , if the customer only wants to find and see some products. This use case could also be used as a part of the Add to Shopping Cart use case. Note, that Checkout use case is included use case not available by itself - checkout is part of Add to Shopping Cart use case.

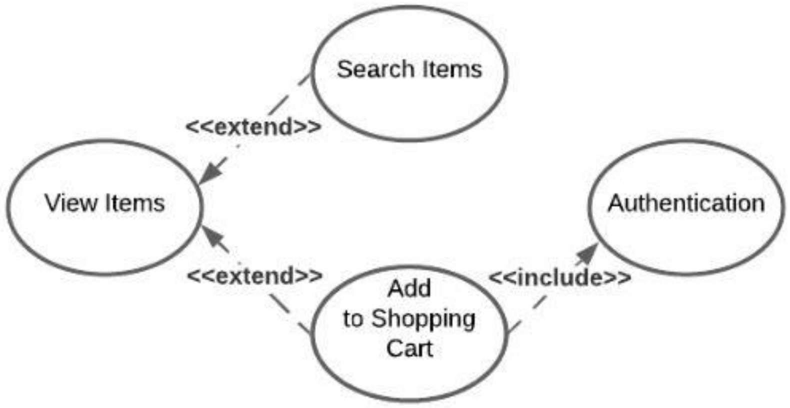
Admin actor is used to monetize the website. Login, View Products and Add Products use cases are the top level use cases for Admin. Add Products use case is used to add products. Authentication use case is included in the Add Products, because it required the admin to be authenticated and to add products.

Credit card/cash actor is used for payment. Checkout is included in the Payment use case, because product list is required for payment.

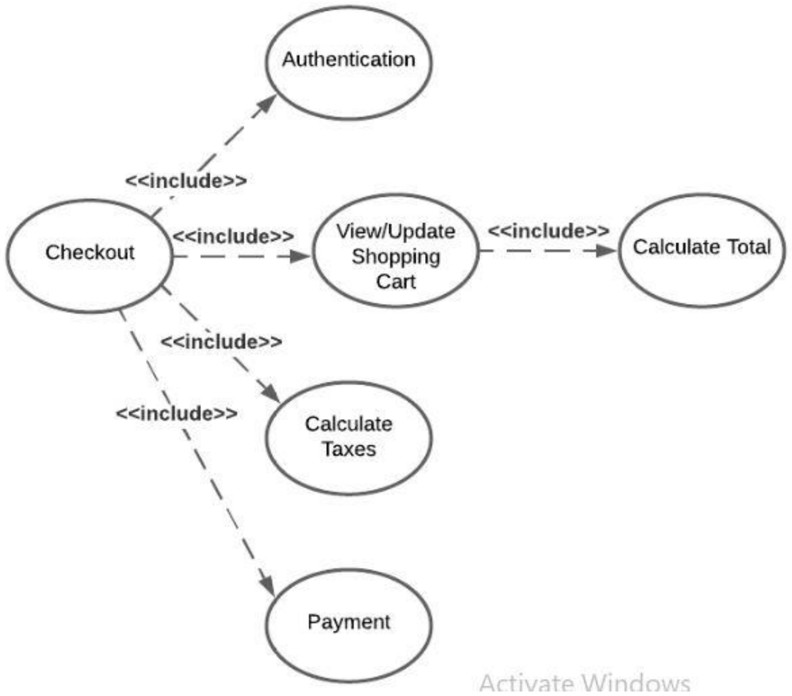


View Items use case is extended by several optional use cases - customers may search for products and add products to Shopping cart. All these use cases are extending use cases because they provide some optional functions allowing customers to find products.

Authentication use case is included in the Add to Shopping Cart, because it requires the customer to be authenticated.



Checkout use case includes several required use cases. Customer should be authenticated. It could be done through user login page. Checkout use case also includes payment use case which could be done by Credit card/ cash. Checkout use case also includes View/Update Shopping Cart Calculate taxes and Calculate Total use cases.



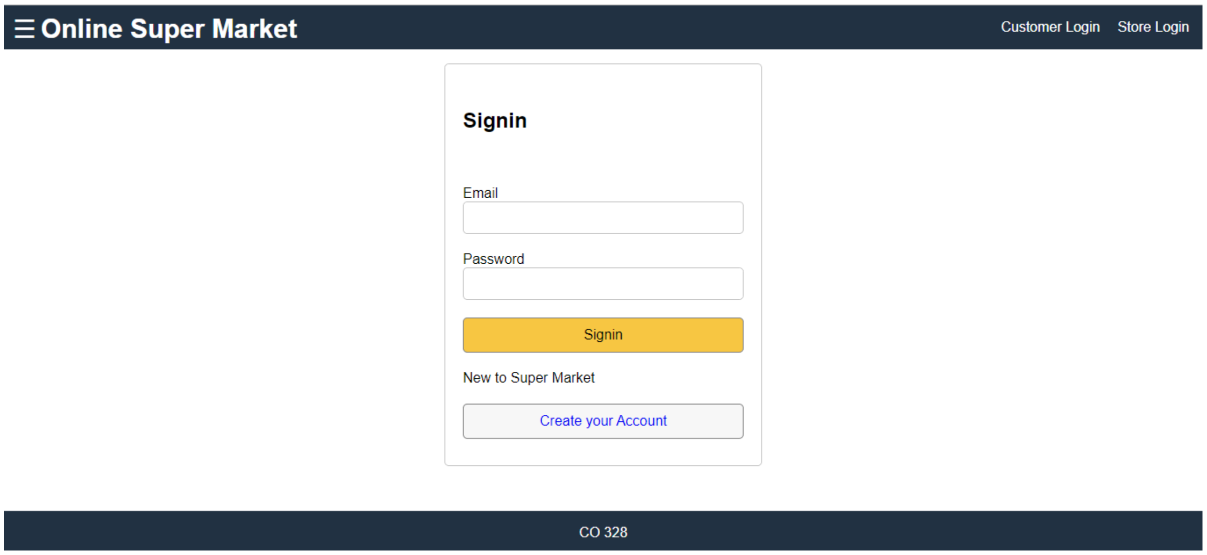
Implementation

The following technologies are used to developed the website.:

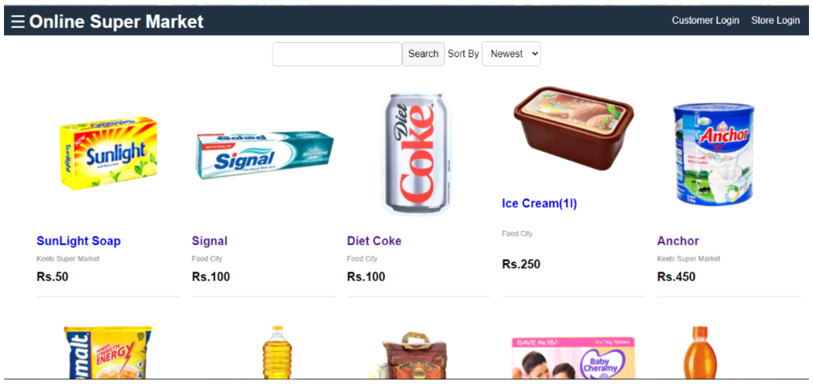
- HTML5 and CSS3: Semantic Elements, CSS Grid, Flexbox
- React: Components, Props, Events, Hooks, Router, Axios
- Redux: Store, Reducers, Actions
- Node & Express: Web API, Body Parser, File Upload, JWT
- MongoDB: Mongoose, Aggregation
- Development: ESLint, Babel, Git, Github,

React library in JavaScript is used to create the frontend of this website.

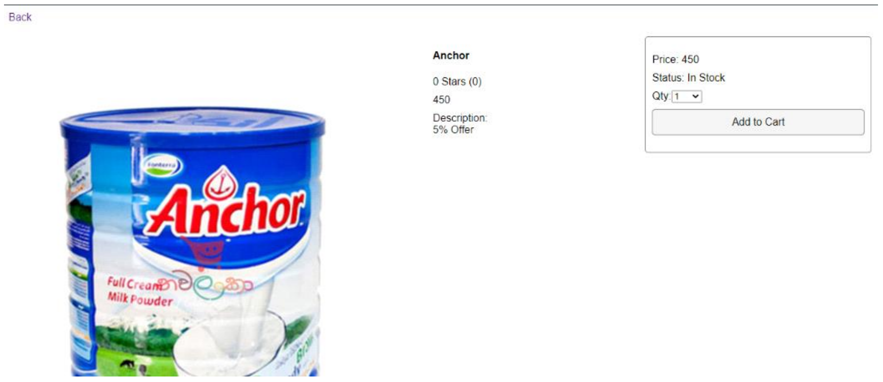
A user or store administrator must register into the system and using the registration details they can access the system with their allowed functionalities. Below is the user interface of login form of this website user and super market administrators will have separate registration as well as separate login forms.



This is the home page of online grocery delivery. It shows a list of products added by several supermarket/grocery administrators.



When the user clicks on a product, the details of the product will be shown like here..


Back

Anchor

0 Stars (0)

450

Description:
5% Offer

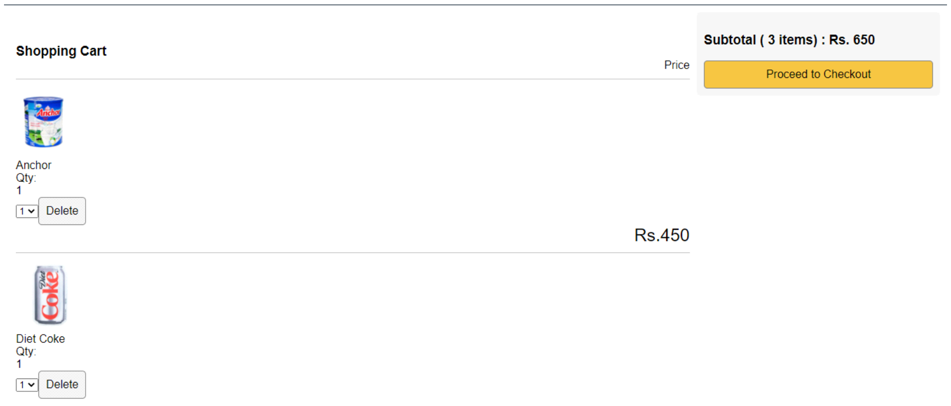
Price: 450

Status: In Stock

Qty: 1

Node and Express are used to develop the backend server of this application . A MongoDB database is created to send and retrieve the data. React Hooks is used to fetch data from the server. The axios library is used to do this job in a modern async/await style. When it comes to handling multiple forms redux is used to have the state management.

A user friendly shopping cart component is designed using React and Redux like shown here.


Shopping Cart

Price

Anchor
Qty:
1

Rs.450

Diet Coke
Qty:
1

Subtotal (3 items) : Rs. 650

Grocery administrators should be able to define products and update the count in stock whenever they like. They can also view the orders when they login to their accounts and they will respond to the orders by delivering them to the address provided by customers.

Test cases and results of main functionalities

Prerequisites for Javascript unit testing

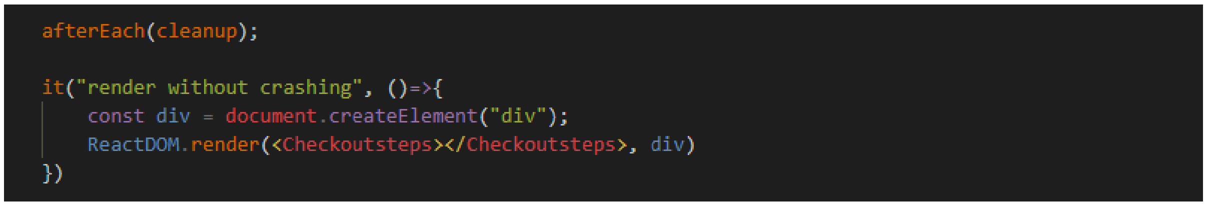
In order to do the unit testing, jest library was used. Jest is a library written by facebook and has become very popular in the past few years. Jest can be used to write several test cases like the ones we tested here.

Front End Testing

Testing Components

1.

To verify whether the component are rendering successfully the following test case was written

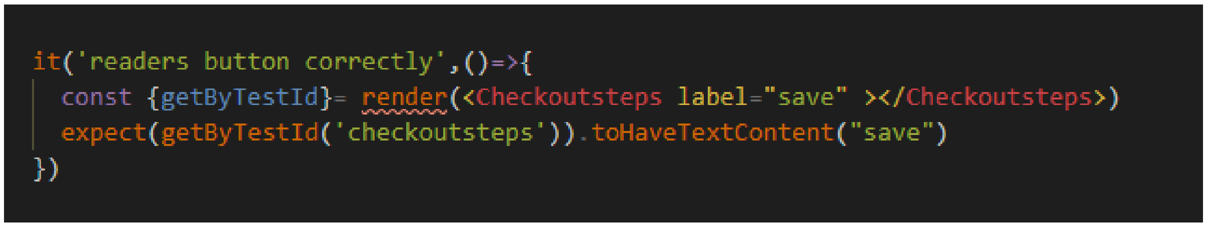


```
afterEach(cleanup);

it("render without crashing", ()=>{
    const div = document.createElement("div");
    ReactDOM.render(<Checkoutsteps></Checkoutsteps>, div)
})
```

If there is no exception thrown then we can assume the test case has passed.

2.

This test case, test the button functionality. Pass the value the test id and expected to have the “save” text.

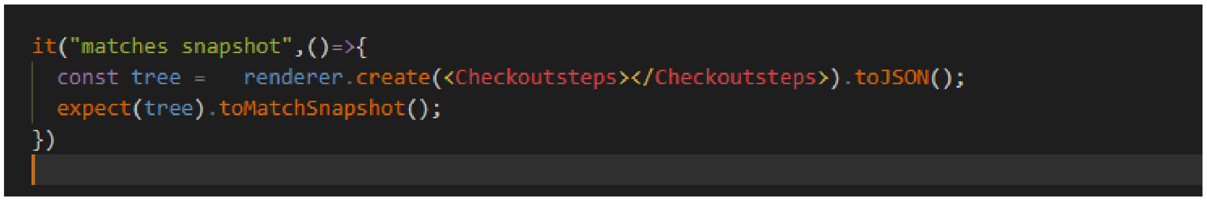


```
it('readers button correctly',()=>{
  const {getByTestId}= render(<Checkoutsteps label="save" ></Checkoutsteps>)
  expect(getByTestId('checkoutsteps')).toHaveTextContent("save")
})
```

Snapshot Testing

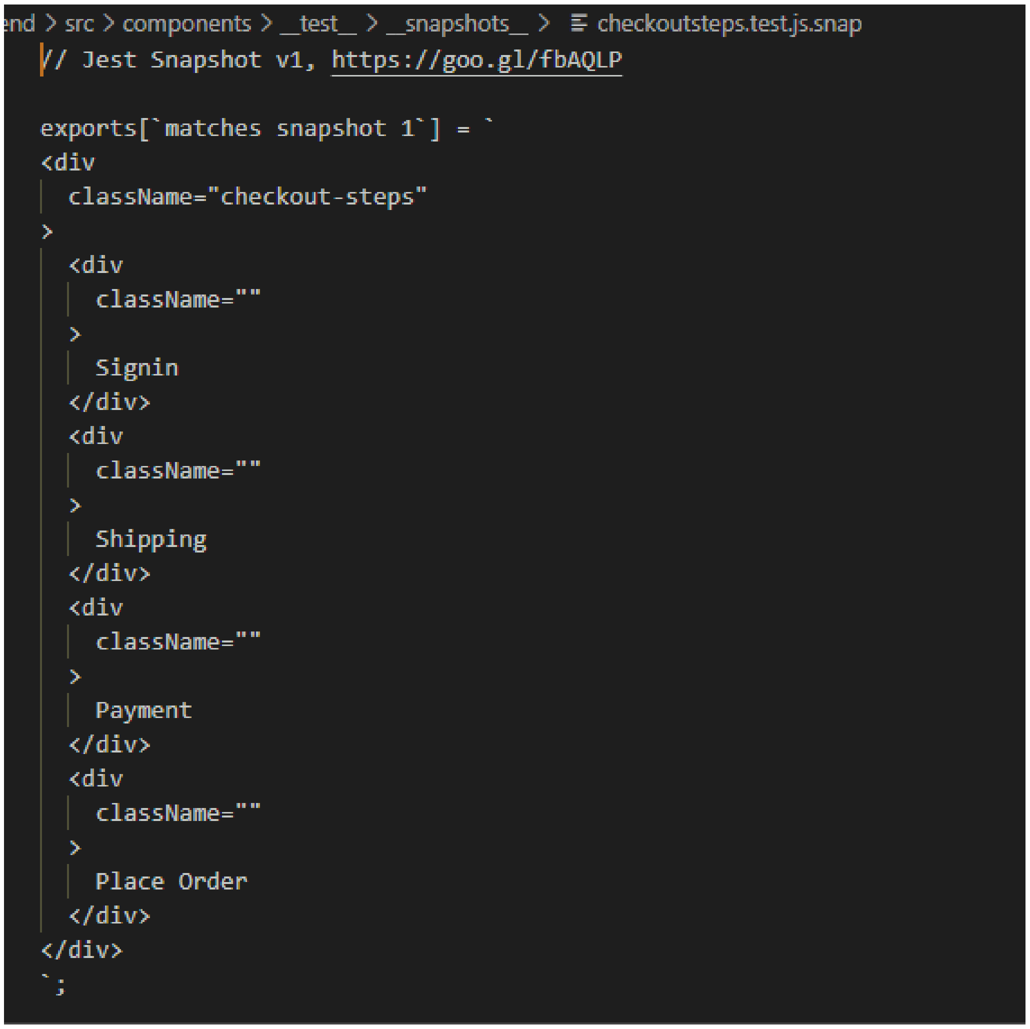
Snapshot testing is another feature of Jest which automatically generates text snapshots of the components and saves them to disk so if the UI output changes later on, it will get notified without manually writing any assertions on the component output. When running a snapshot test for the first time, Jest renders the component and saves the output as a JavaScript object. Each time the test is run again, Jest will compare its output to the saved snapshot and if the component's output is different from the snapshot, the test will fail. This may be an indicator that the component has a bug somewhere and you can go ahead and fix it until its output matches the snapshot.

With snapshot testing, we will always know when you accidentally change a component's behavior and it also saves you from writing a lot of assertions that check if your components are behaving as expected. We include one snapshot test for the checkout steps component.



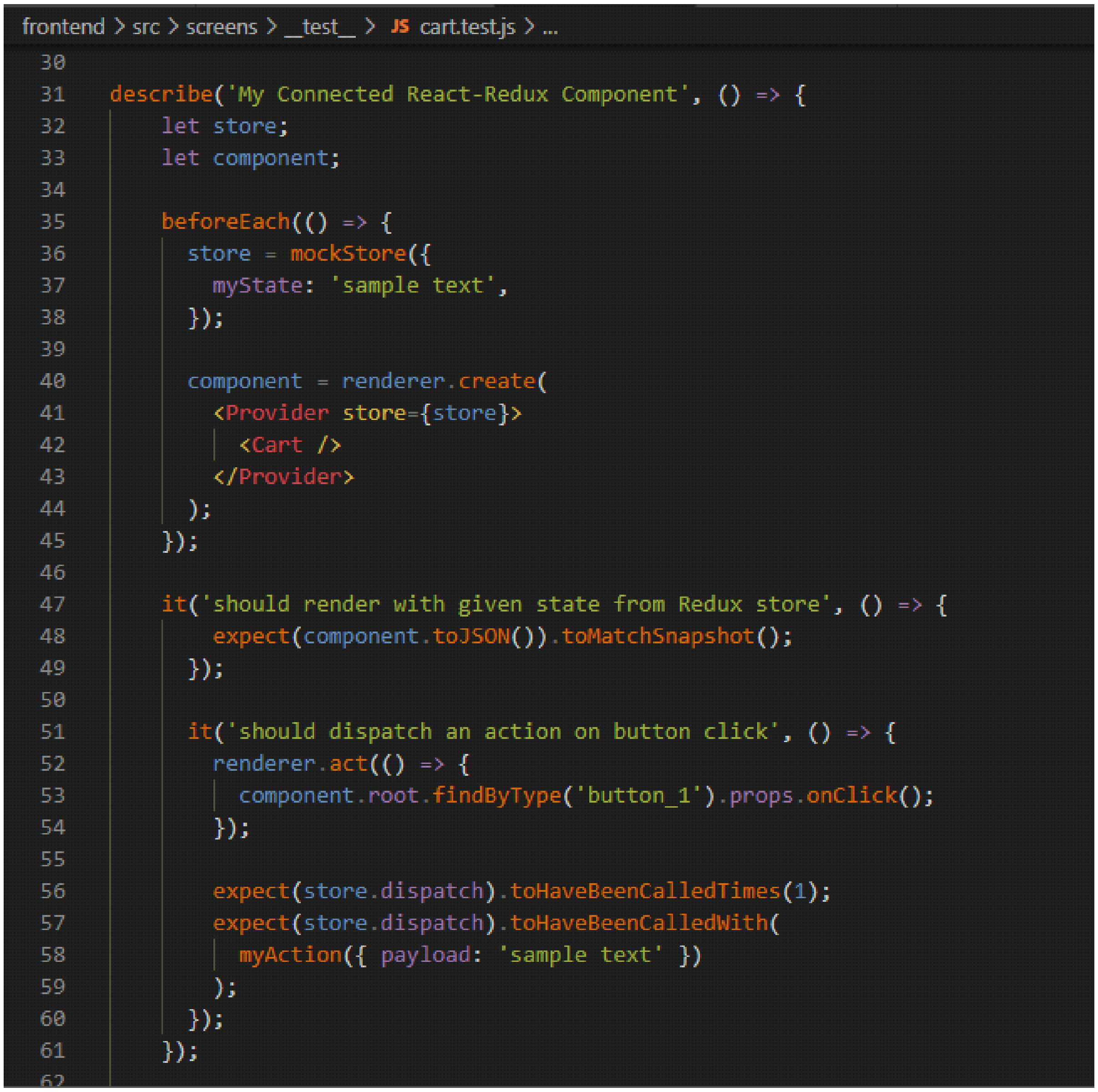
```
it("matches snapshot",()=>{
  const tree =   renderer.create(<Checkoutsteps></Checkoutsteps>).toJSON();
  expect(tree).toMatchSnapshot();
})
```

Run tests and the output will show that a snapshot has been added.



```
nd > src > components > __test__ > __snapshots__ > checkoutsteps.test.js.snap
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`matches snapshot 1`] = `
<div
  className="checkout-steps"
>
  <div
    className=""
  >
    Signin
  </div>
  <div
    className=""
  >
    Shipping
  </div>
  <div
    className=""
  >
    Payment
  </div>
  <div
    className=""
  >
    Place Order
  </div>
</div>
`;
```

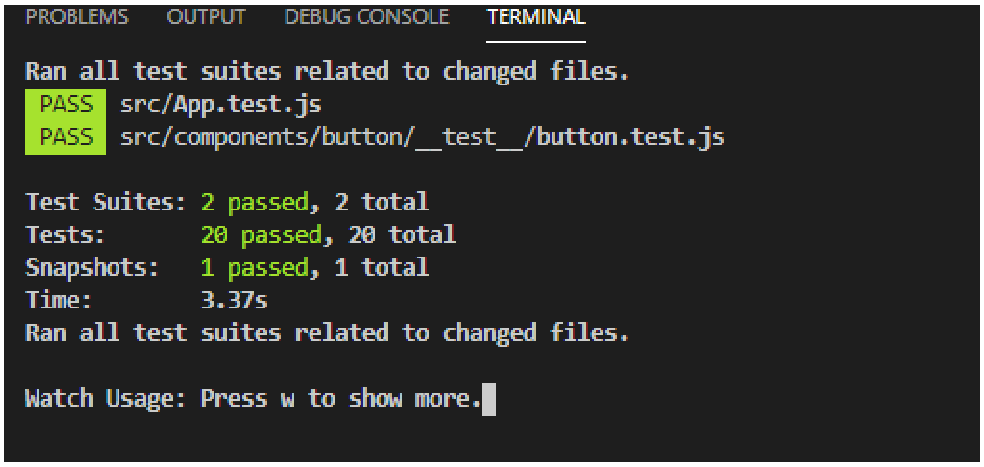
Test React-Redux connected Components



```
frontend > src > screens > __test__ > JS cart.test.js > ...

describe('My Connected React-Redux Component', () => {
    let store;
    let component;

    beforeEach(() => {
      store = mockStore({
        myState: 'sample text',
      });

      component = renderer.create(
        <Provider store={store}>
          <Cart />
        </Provider>
      );
    });

    it('should render with given state from Redux store', () => {
      expect(component.toJSON()).toMatchSnapshot();
    });

    it('should dispatch an action on button click', () => {
      renderer.act(() => {
        component.root.findByType('button_1').props.onClick();
      });

      expect(store.dispatch).toHaveBeenCalledTimes(1);
      expect(store.dispatch).toHaveBeenCalledWith(
        myAction({ payload: 'sample text' })
      );
    });
    });
```

Everything passed into mockStore will be the Redux store's initial state. So make sure we provide everything that's needed by our connected React component to render without any problems. Then, create the React component with a renderer of our choice for the test. Thus, the mocked Redux store is provided for our React component for the purpose of this test. For our unit test, the simplest snapshot test of the rendered component.

Check the snapshot test's output whether everything got rendered as expected with the given state from the mocked Redux store. We check whether a given HTML input field receives the state from the Redux store as its initial state. Now, for our second unit test, it will be check with Jest whether a HTML button click will dispatch a specific Redux action.



```
Ran all test suites related to changed files.
 PASS  src/App.test.js
 PASS  src/components/button/__test__/button.test.js

Test Suites: 2 passed, 2 total
Tests:       20 passed, 20 total
Snapshots:   1 passed, 1 total
Time:        3.37s
Ran all test suites related to changed files.

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

All test cases written above are passed

Backend Testing

Backend testing was divided into 3 parts

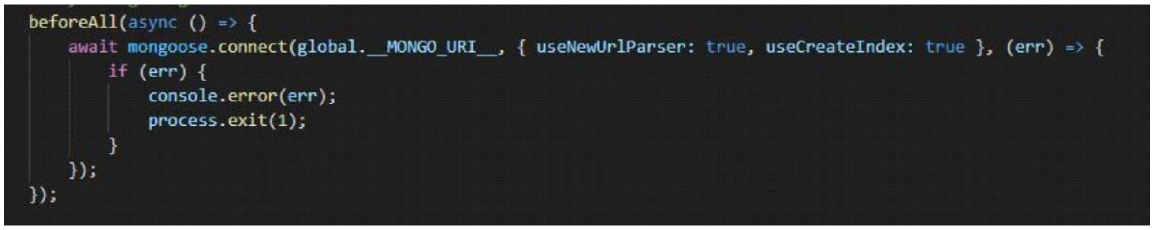
1.Testing the database connectivity

2.Testing api endpoints.

3.Testing JWT.

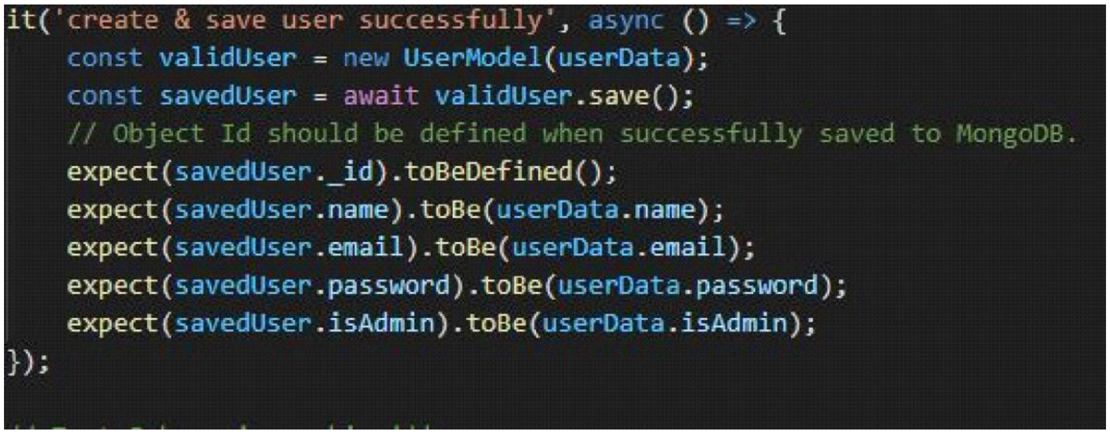
Database connectivity

First the mongodb database connection was tested within the test code using beforeAll function



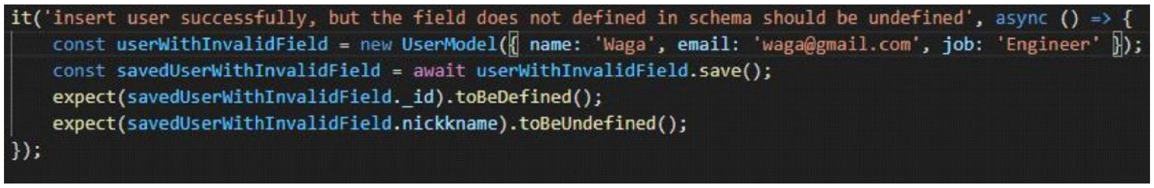
```
beforeAll(async () => {
    await mongoose.connect(global.__MONGO_URI__, { useNewUrlParser: true, useCreateIndex: true }, (err) => {
        if (err) {
            console.error(err);
            process.exit(1);
        }
    });
});
```

Then creating a user document and inserting details into the document was tested.



```
it('create & save user successfully', async () => {
    const validUser = new UserModel(userData);
    const savedUser = await validUser.save();
    // Object Id should be defined when successfully saved to MongoDB.
    expect(savedUser._id).toBeDefined();
    expect(savedUser.name).toBe(userData.name);
    expect(savedUser.email).toBe(userData.email);
    expect(savedUser.password).toBe(userData.password);
    expect(savedUser.isAdmin).toBe(userData.isAdmin);
});
```

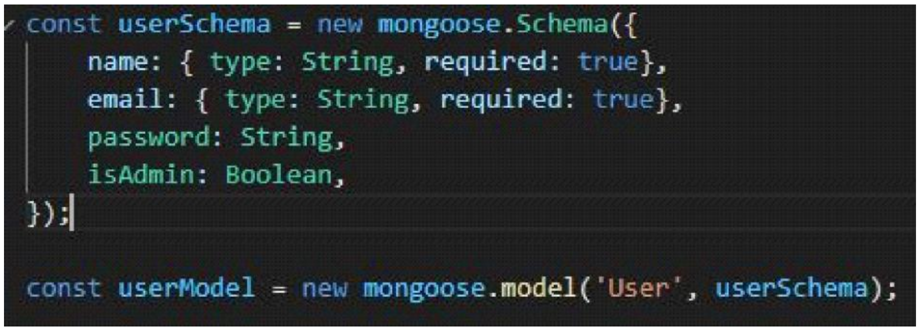
After that database insertion is tested with a field that was not defined in the schema and expected to be failed



```
it('insert user successfully, but the field does not defined in schema should be undefined', async () => {
    const userWithInvalidField = new UserModel({ name: 'Waga', email: 'waga@gmail.com', job: 'Engineer' });
    const savedUserWithInvalidField = await userWithInvalidField.save();
    expect(savedUserWithInvalidField._id).toBeDefined();
    expect(savedUserWithInvalidField.nickkname).toBeUndefined();
});
```

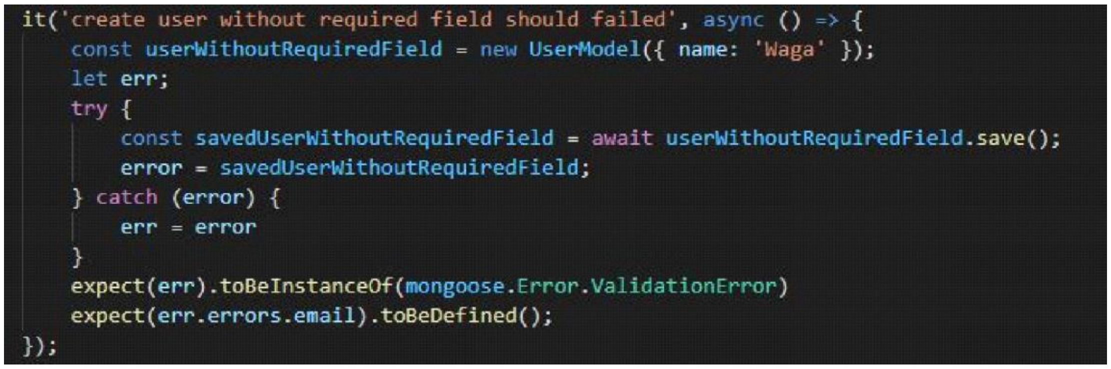
Here the job field wasn't defined in the user model therefore it was expected to be undefined.

The user schema:



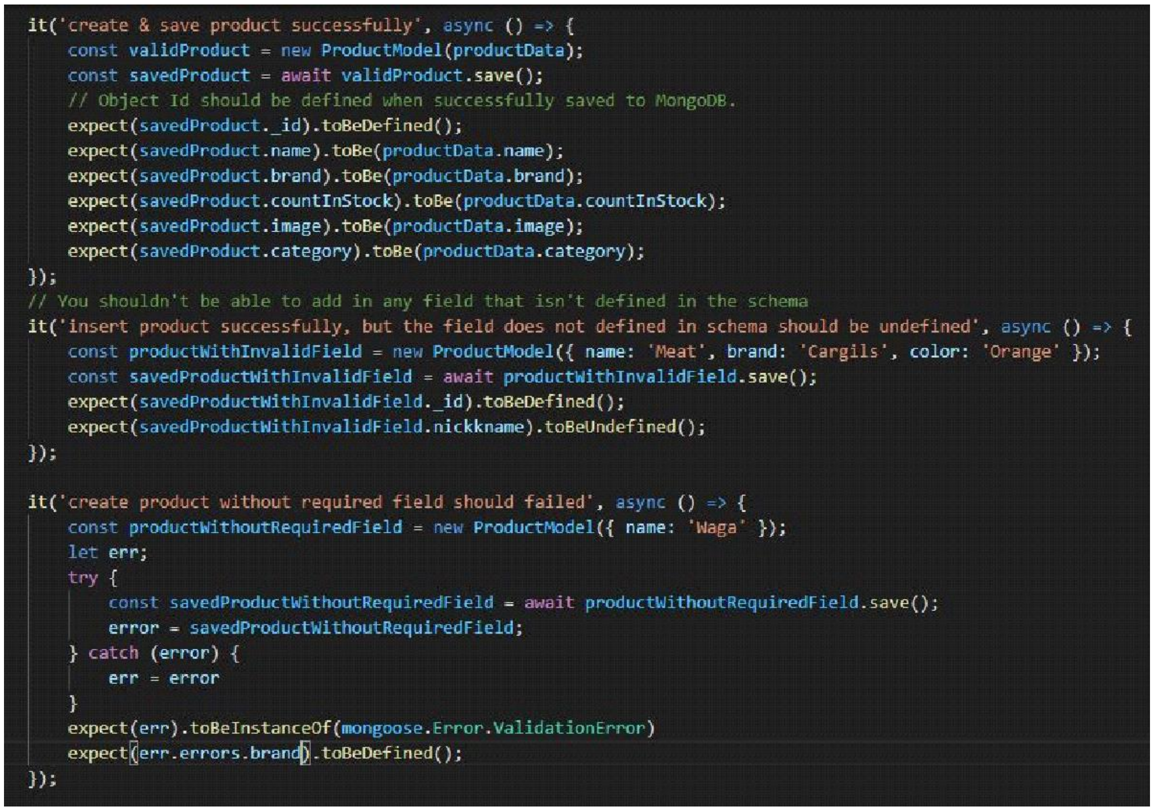
```
const userSchema = new mongoose.Schema({
    name: { type: String, required: true},
    email: { type: String, required: true},
    password: String,
    isAdmin: Boolean,
});

const userModel = new mongoose.model('User', userSchema);
```

Here there are two required fields name and email so one required field(email) was not inserted to the database and tested to be failed



```
it('create user without required field should failed', async () => {
    const userWithoutRequiredField = new UserModel({ name: 'Waga' });
    let err;
    try {
        const savedUserWithoutRequiredField = await userWithoutRequiredField.save();
        error = savedUserWithoutRequiredField;
    } catch (error) {
        err = error
    }
    expect(err).toBeInstanceOf(mongoose.Error.ValidationError)
    expect(err.errors.email).toBeDefined();
});
```

Same tests of user model was done to products model

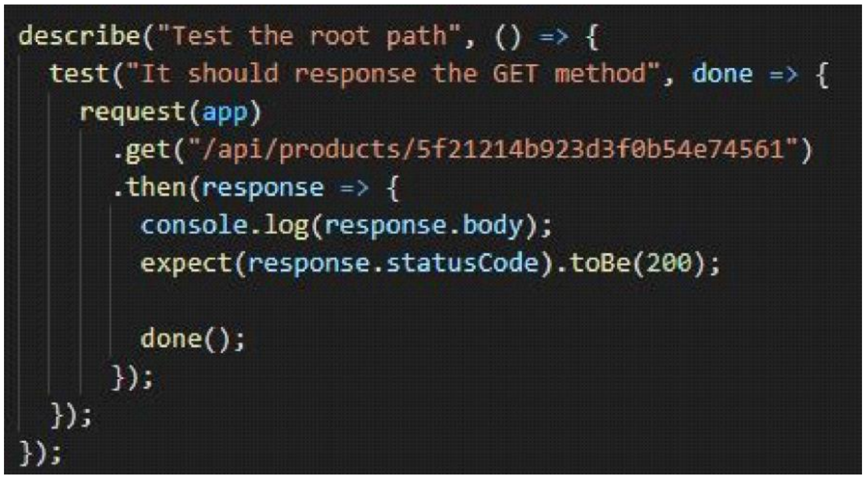


```
it('create & save product successfully', async () => {
    const validProduct = new ProductModel(productData);
    const savedProduct = await validProduct.save();
    // Object Id should be defined when successfully saved to MongoDB.
    expect(savedProduct._id).toBeDefined();
    expect(savedProduct.name).toBe(productData.name);
    expect(savedProduct.brand).toBe(productData.brand);
    expect(savedProduct.countInStock).toBe(productData.countInStock);
    expect(savedProduct.image).toBe(productData.image);
    expect(savedProduct.category).toBe(productData.category);
});
// You shouldn't be able to add in any field that isn't defined in the schema
it('insert product successfully, but the field does not defined in schema should be undefined', async () => {
    const productWithInvalidField = new ProductModel({ name: 'Meat', brand: 'Cargils', color: 'Orange' });
    const savedProductWithInvalidField = await productWithInvalidField.save();
    expect(savedProductWithInvalidField._id).toBeDefined();
    expect(savedProductWithInvalidField.nickkname).toBeUndefined();
});

it('create product without required field should failed', async () => {
    const productWithoutRequiredField = new ProductModel({ name: 'Waga' });
    let err;
    try {
        const savedProductWithoutRequiredField = await productWithoutRequiredField.save();
        error = savedProductWithoutRequiredField;
    } catch (error) {
        err = error
    }
    expect(err).toBeInstanceOf(mongoose.Error.ValidationError)
    expect(err.errors.brand).toBeDefined();
});
```

Testing Endpoints

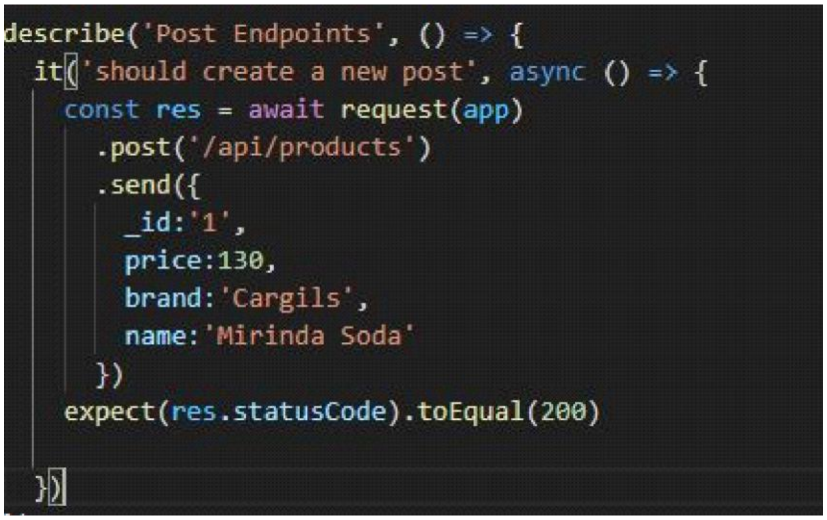
GET and POST are tested by sending and receiving data. If the sending and receiving are successful then test will expect for the respective http codes.

A Get request was send to /api/products/5f21214b923d3f0b54e74561 route since product of that "5f21214b923d3f0b54e74561 " available it was expected to receive a "200" code



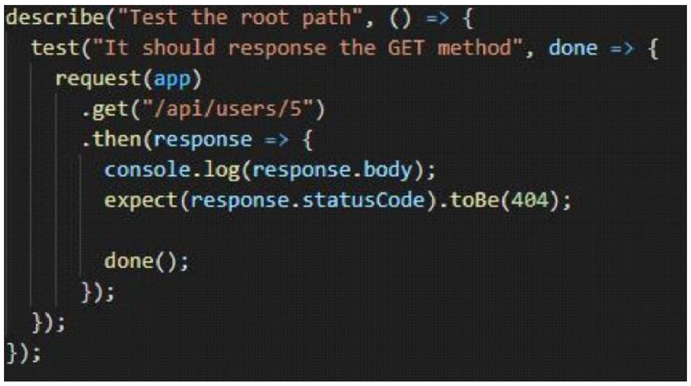
```
describe("Test the root path", () => {
  test("It should response the GET method", done => {
    request(app)
      .get("/api/products/5f21214b923d3f0b54e74561")
      .then(response => {
        console.log(response.body);
        expect(response.statusCode).toBe(200);

        done();
      });
  });
});
```

Thereafter post method was checked by sending the "/api/products" route and expected to receive "200" code.



```
describe('Post Endpoints', () => {
  it('should create a new post', async () => {
    const res = await request(app)
      .post('/api/products')
      .send({
        _id:'1',
        price:130,
        brand:'Cargils',
        name:'Mirinda Soda'
      })
    expect(res.statusCode).toEqual(200)

  })
```

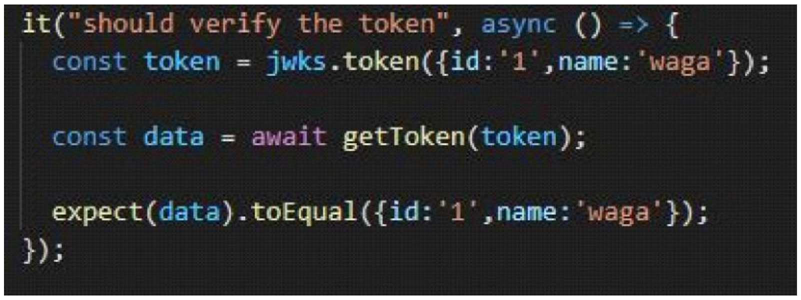
Then a non existing id "5" was send to " /api/users/5" route and expected to receive "404" code.



```
describe("Test the root path", () => {
  test("It should response the GET method", done => {
    request(app)
      .get("/api/users/5")
      .then(response => {
        console.log(response.body);
        expect(response.statusCode).toBe(404);

        done();
      });
  });
});
```

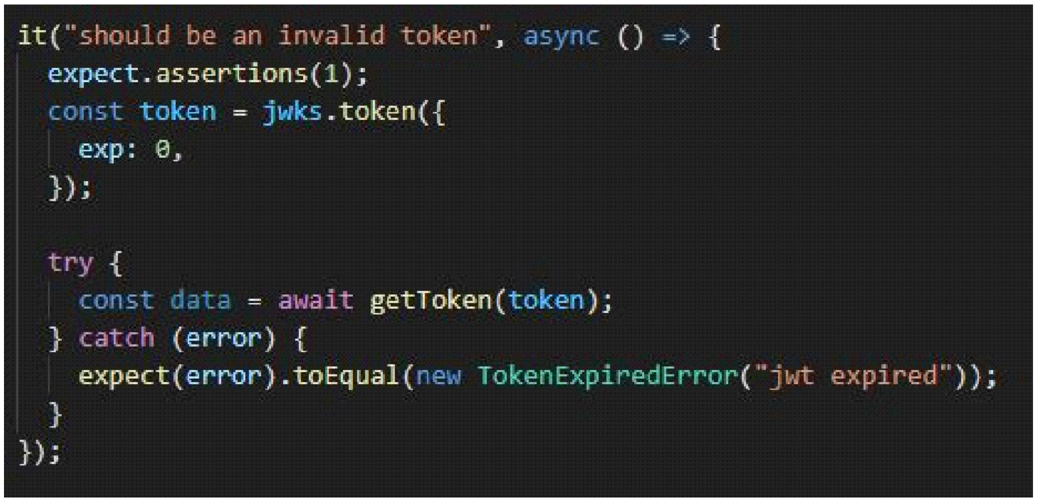
JWT testing

A JWT is created and expected the same token to be returned



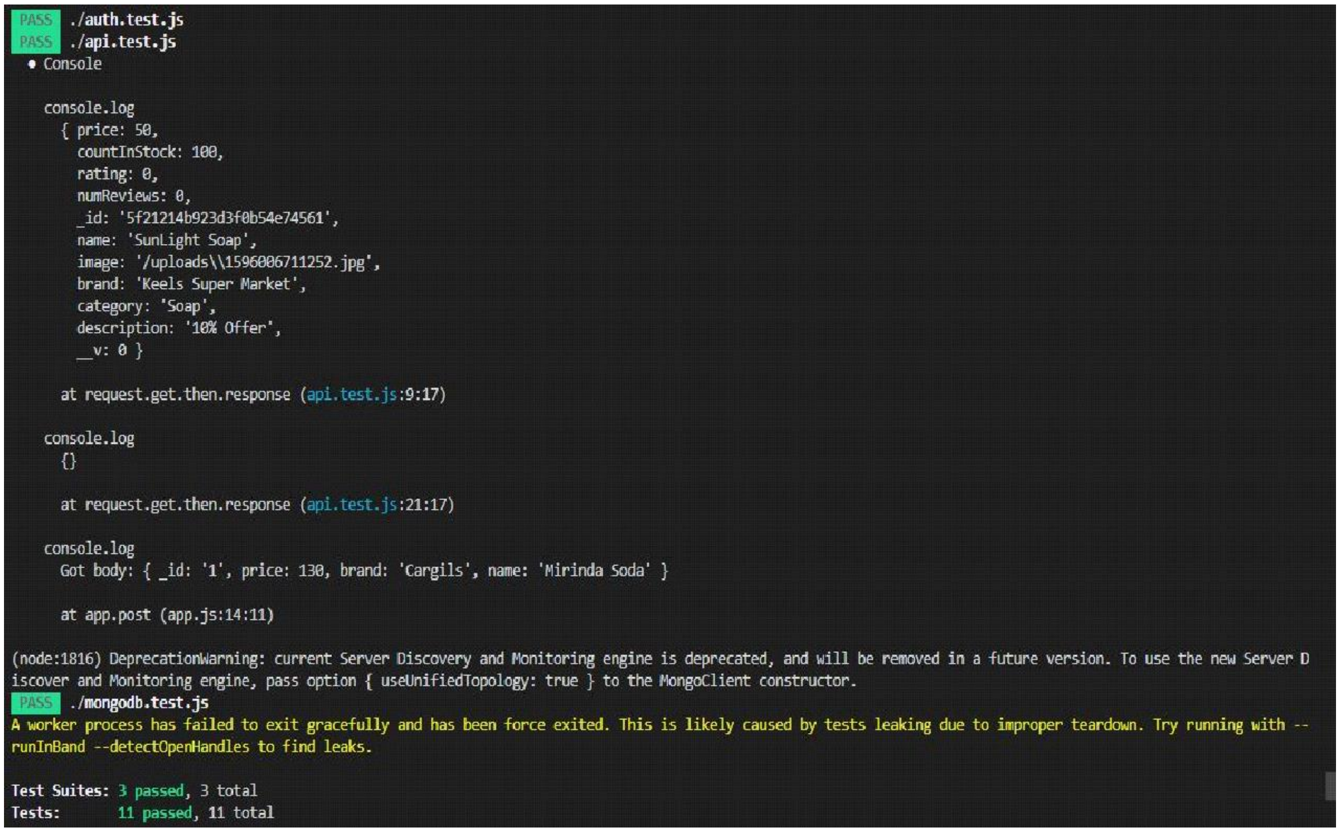
```
it("should verify the token", async () => {
  const token = jwks.token({id:'1',name:'waga'});

  const data = await getToken(token);

  expect(data).toEqual({id:'1',name:'waga'});
});
```

Token expiry time is set to zero and check if the token is valid. Test is expected to get an error.



```
it("should be an invalid token", async () => {
  expect.assertions(1);
  const token = jwks.token({
    exp: 0,
  });

  try {
    const data = await getToken(token);
  } catch (error) {
    expect(error).toEqual(new TokenExpiredError("jwt expired"));
  }
});
```

All test cases above were passed



```
PASS  ./auth.test.js
PASS  ./api.test.js
  ● Console

    console.log
      { price: 50,
        countInStock: 100,
        rating: 0,
        numReviews: 0,
        _id: '5f21214b923d3f0b54e74561',
        name: 'SunLight Soap',
        image: '/uploads\\1596006711252.jpg',
        brand: 'Keels Super Market',
        category: 'Soap',
        description: '10% Offer',
        __v: 0 }

      at request.get.then.response (api.test.js:9:17)

    console.log
      {}

      at request.get.then.response (api.test.js:21:17)

    console.log
      Got body: { _id: '1', price: 130, brand: 'Cargils', name: 'Mirinda Soda' }

      at app.post (app.js:14:11)

(node:1816) DeprecationWarning: current Server Discovery and Monitoring engine is deprecated, and will be removed in a future version. To use the new Server D
iscover and Monitoring engine, pass option { useUnifiedTopology: true } to the MongoClient constructor.
PASS  ./mongodb.test.js
A worker process has failed to exit gracefully and has been force exited. This is likely caused by tests leaking due to improper teardown. Try running with --
runInBand --detectOpenHandles to find leaks.

Test Suites: 3 passed, 3 total
Tests:       11 passed, 11 total
```