CS331 TESTING REPORT

PEERKART

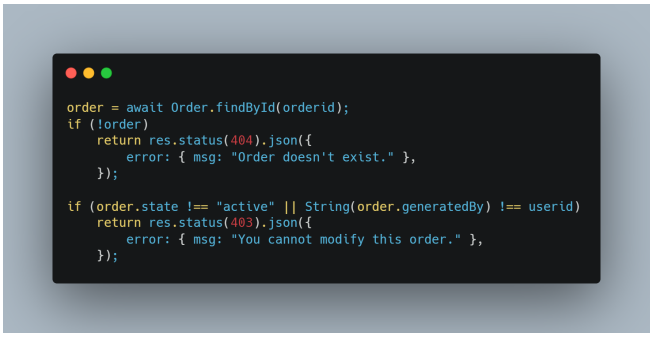
Unit Testing

Backend:

We performed a thorough data validation of the request body for every endpoint in the backend of the app, to avoid crashing of the app because of some invalid input because of the user. Below is an example of such data validation. Here I have used the express-validator library for the data validation of the requests and the unit testing is done using ***chai*** and ***supertest.***

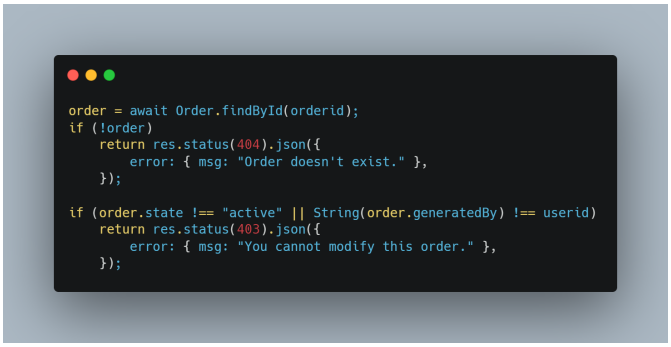
```
router.post(
  "/",
  authController.authMiddleware,
  [
    body("name", "Order name has to be atleast 3 char long")
      .trim()
      .isLength({ min: 3 }),
    body("items").isArray({ min: 1 }),
    body("category")
      .trim()
      .custom((value, { req }) => {
        const categories = [
          "Grocery",
          "Medicines",
          "Fish and Meat",
          "Stationary",
        ];
        if (!categories.includes(value)) {
          throw new Error("Invalid category");
        }
        return true;
      }),
    body("address.address").trim(),
    body("contact", "Invalid contact number")
      .trim()
      .isNumeric()
      .isLength({ min: 10, max: 10 }),
    check("paymentMethod.paymentType").isLength({ min: 1 }),
    check("paymentMethod.paymentId").isLength({ min: 1 }),
  ],
  orderController.addOrder
);
```

Here for the ***"/orders/"*** endpoint, we confirm that the order ***'name'*** is at least 3 character long while removing all the extra blanks, there is a check if the ***'items'*** is an array with at least 1 item and if the ***'category'*** is one of the few we specify. The ***'address'***, ***'contact'***, ***'paymentMethod'*** are also checked to see if they are in valid format



```
order = await Order.findById(orderid);
if (!order)
    return res.status(404).json({
        error: { msg: "Order doesn't exist." },
    });

if (order.state !== "active" || String(order.generatedBy) !== userid)
    return res.status(403).json({
        error: { msg: "You cannot modify this order." },
    });
```

Here is another example.
I check if the ***'id'*** in the API call is a proper Mongold.



```
order = await Order.findById(orderid);
if (!order)
    return res.status(404).json({
        error: { msg: "Order doesn't exist." },
    });

if (order.state !== "active" || String(order.generatedBy) !== userid)
    return res.status(403).json({
        error: { msg: "You cannot modify this order." },
    });
```

After the above check if ***'id'*** is a proper Mongold, here we check if it really exists in the database and if the other required conditions of the endpoint are met. That is if the ***state*** of the order is ***"active"*** and if the order was really created by the current user.

Here is a part of the unit tests file. These are a few of the tests which check if the overall API works together fine and well. In this example,

```
it("POST /orders once again", () => {
    return request
      .post("orders")
      .set({ Authorization: `Bearer ${newUser.token}` })
      .send(newOrder)
      .then((res) => {
        expect(res.body.data.generatedBy).to.be.eq(newUser.id);
        expect(res.body.data.state).to.be.eq("active");
        expect(res.body.data.acceptedBy).to.be.null;
        newOrder = res.body.data;
      });
  });

  it("GET /users/orders/created", () => {
    return request
      .get("users/orders/created")
      .set({ Authorization: `Bearer ${newUser.token}` })
      .then((res) => {
        expect(res.body.data).to.have.lengthOf(1);
        expect(res.body.data).to.deep.include(newOrder);
      });
  });

  it("GET /orders/:id", () => {
    return request
      .get(`orders/${newOrder._id}`)
      .set({ Authorization: `Bearer ${newUser.token}` })
      .send(newOrder)
      .then((res) => {
        expect(res.body.data).to.deep.include(newOrder);
      }
  });
```

By making a call to the ***"/orders/"*** endpoint. I confirm that the ***state*** is ***"active"*** and that the ***generatedBy*** attribute matches the current user

In this part of the code, I check if the created order is properly associated with the current user. ***"/users/orders/created".*** By checking if the same order is received in the response.

In the third part, I make a call to the ***"/orders/:id"*** endpoint and check if the response body has the same order as the ***id*** provided

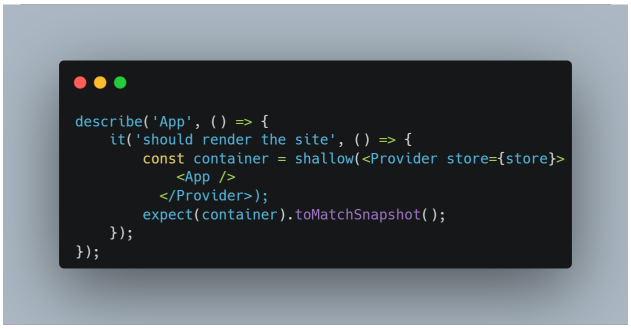
This is the final report of all the tests (20 tests) I wrote to test my API.
There are various tests like logging in with wrong credentials gives the expected error response and creating an order with invalid category gives valid error, to name a few of the tests.

Frontend Website (React):

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`App should render the site 1`] = `ShallowWrapper {}`;

exports[`CreateOrderModal CreateOrderModal component is rendered 1`] = `ShallowWrapper {}`;

exports[`Dashboard Dashboard page is rendered 1`] = `ShallowWrapper {}`;

exports[`Home Home page is rendered 1`] = `ShallowWrapper {}`;

exports[`Login Login page is rendered 1`] = `ShallowWrapper {}`;

exports[`Modal Modal component is rendered 1`] = `ShallowWrapper {}`;

exports[`ModalAccepted ModalAccepted component is rendered 1`] = `ShallowWrapper {}`;

exports[`ModalTransaction ModalTransaction component is rendered 1`] = `ShallowWrapper {}`;

exports[`ProfileDetail ProfileDetail component is rendered 1`] = `ShallowWrapper {}`;

exports[`Register Register page is rendered 1`] = `ShallowWrapper {}`;

exports[`Sidebar Sidebar component is rendered 1`] = `ShallowWrapper {}`;
```

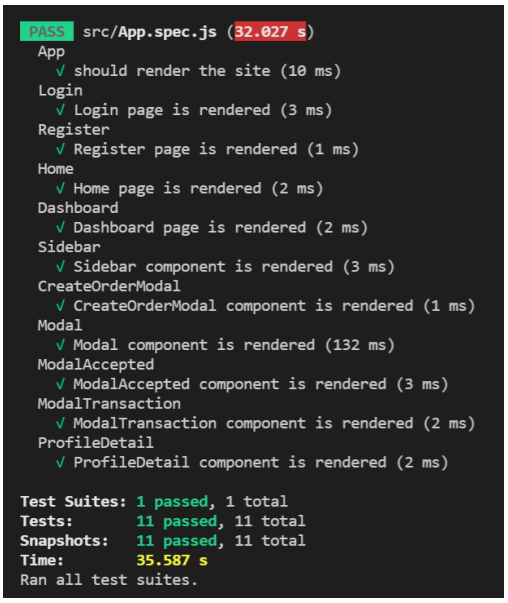
We performed a thorough testing of rendering the components and various pages of the website like ***Login***, ***Register***, ***Home***, ***Dashboard***, etc by using the snapshot technique ***"(toMatchSnapShot())"*** . Due to this it makes sures that our UI doesn't change unexpectedly. A typical snapshot test case renders a UI component, takes a snapshot, then compares it to a reference snapshot file stored alongside the test.

Test:



```
describe('App', () => {
    it('should render the site', () => {
        const container = shallow(<Provider store={store}>
            <App />
          </Provider>);
        expect(container).toMatchSnapshot();
    });
});
```

Result:

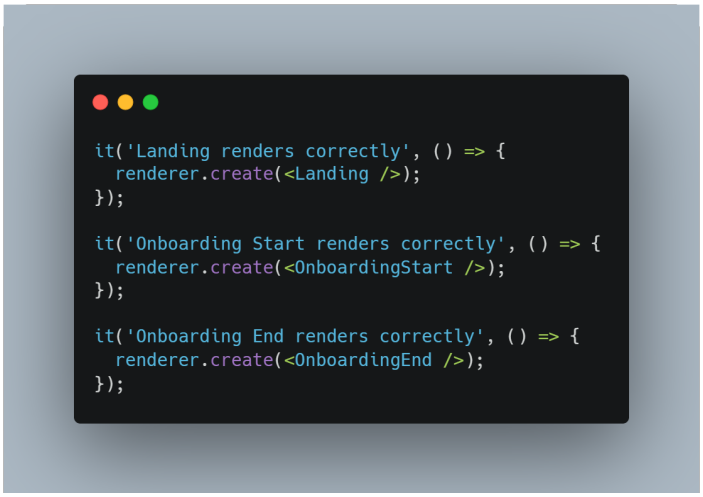


```
PASS src/App.spec.js (32.027 s)
  App
    √ should render the site (10 ms)
  Login
    √ Login page is rendered (3 ms)
  Register
    √ Register page is rendered (1 ms)
  Home
    √ Home page is rendered (2 ms)
  Dashboard
    √ Dashboard page is rendered (2 ms)
  Sidebar
    √ Sidebar component is rendered (3 ms)
  CreateOrderModal
    √ CreateOrderModal component is rendered (1 ms)
  Modal
    √ Modal component is rendered (132 ms)
  ModalAccepted
    √ ModalAccepted component is rendered (3 ms)
  ModalTransaction
    √ ModalTransaction component is rendered (2 ms)
  ProfileDetail
    √ ProfileDetail component is rendered (2 ms)

Test Suites: 1 passed, 1 total
Tests:       11 passed, 11 total
Snapshots:   11 passed, 11 total
Time:        35.587 s
Ran all test suites.
```

Mobile Application (React - Native):

The testing of the mobile application was performed using a library called jest.

The testing mainly focused around the correct rendering of the UI components and the proper working of the functions and hooks.



```
it('Landing renders correctly', () => {
  renderer.create(<Landing />);
});

it('Onboarding Start renders correctly', () => {
  renderer.create(<OnboardingStart />);
});

it('Onboarding End renders correctly', () => {
  renderer.create(<OnboardingEnd />);
});
```

This snippet of code provides the test for the correctness of the screens and uses the renderer method of the react native.

```
it('Login renders correctly', () => {
  const store = configureStore({ reducer: reducers });
  renderer.create(
    <Provider store={store}>
      <Login />
    </Provider>,
  );
});

it('Register renders correctly', () => {
  const store = configureStore({ reducer: reducers });
  renderer.create(
    <Provider store={store}>
      <Register />
    </Provider>,
  );
});

it('Home renders correctly', () => {
  const store = configureStore({ reducer: reducers });
  renderer.create(
    <Provider store={store}>
      <Home />
    </Provider>,
  );
});

it('Cart renders correctly', () => {
  const store = configureStore({ reducer: reducers });
  renderer.create(
    <Provider store={store}>
      <Cart />
    </Provider>,
  );
});
```

Since the application uses redux we needed to pass the store while creating the mocks for the same. This has been demonstrated in the snippet next to us.

```
import { login } from '../src/screens/Login';
import axios from 'axios';

describe('Login tests', () => {
  describe('Login function ', () => {
    describe('with success', () => {
      const data = {
        token: '',
        id: '',
        username: '',
        email: '',
        points: '',
      };

      beforeEach(() => {
        axios.post.mockImplementationOnce(() => Promise.resolve(data));
      });

      it('should call endpoint with given email & password', async () => {
        const email = 'test@gmail.com';
        const password = 'test1234';

        await login(email, password);
        expect(axios.post).toBeCalledWith(
          'https://peerkart-bee.herokuapp.com/api/v1/auth/login',
          { email, password },
        );
      });

      it('should return response data', () => {
        const testLogin = async () => {
          const response = await login('test@gmail.com', 'test1234');
          expect(response).toStrictEqual(data);
        };

        testLogin();
      });
    });
  });
});
```

The well defined and proper behaviour of the functions used in the application is extremely important. The snippet provides one test which I wrote in order to check the behaviour of the login function.

RESULT

```
ARSH KUMAR@LAPTOP-G2U36CL6 MINGW64 ~/Documents/GitHub/peerkartandroid (rest-backend)
$ npm test

> peerkartandroid@0.0.1 test
> jest

 PASS  __tests__/login.test.js
 PASS  __tests__/screen.tests.js (8.159 s)

Test Suites: 2 passed, 2 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        9.347 s
Ran all test suites.
```

Manual Testing

Two accounts were created for the purpose of testing. One of them was used to generate the order and the second one was used to accept the same.

Order Generator Account:

- Filled the account with complete profile details.
- Tested the cart system in place by adding various items into the cart and then later modified them.
- Order was then generated and then verified that it was indeed generated by the same user and not by someone else.
- We generated a couple of orders so that the person accepting can have multiple options to choose from.

Order Acceptor Account:

- Filled the account with complete profile details.
- After the completion of the profile the person could accept the orders which are available.
- After accepting the order the person has an access to the page where he can contact the person who has placed the order and also sees an optimal route to the delivery location.
- After completing the delivery he/she can mark the order as delivered.

The accounts went through the complete process from signing up to testing every feature which ensured that none of the features were breaking in production and that they were all working fine.

We also ran the tests in various devices and resolutions to make sure the UI did not work incorrectly. This process was repeated with varying the inputs and corner case inputs to check for integrity of the application. We employed various techniques to test input validation. We also ran bug tests and resolved them..

Integration Testing

We ran all unit test modules. The observations implied that all the modules are loosely coupled and don't require separate integration testing.

In order to perform manual integration testing,we have tested all the modules separately and made sure that they are working properly in combination with each other.

We prepared a top-down flowchart of all the modules to check the flow of our web application.

System Testing

A complete manual testing was performed on the application on various devices in order to validate the working of the application and also to check its compatibility across devices.

We tested the design and behaviour of the application in accordance with the expectations of the customers and tried to make the experience of the application as close to the one that we would like to have.

USER REGISTRATION

CASE: User with unique email address and valid username, password

RESULT: The user was registered successfully.

CASE: User with already registered email.

RESULT: Email already in use error.

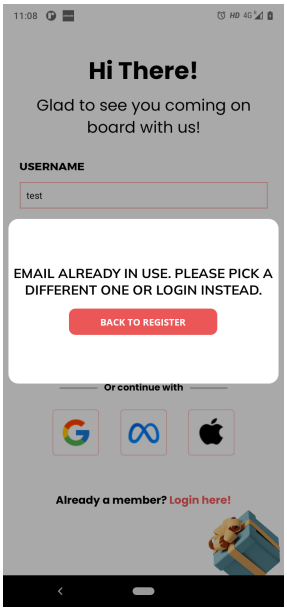
CASE: User with unique email but username less than 5 characters long.

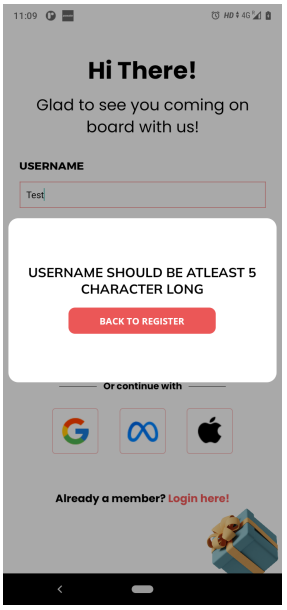
RESULT: Username too short error.

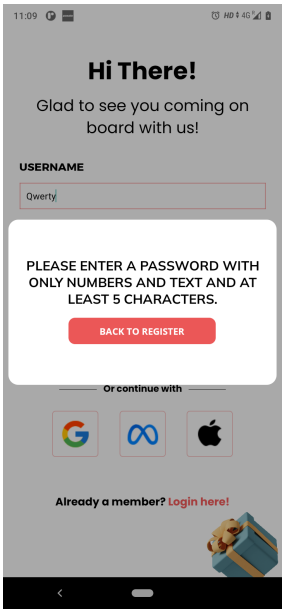
CASE: User with unique email but weak password.

RESULT: Password too short or missing a digit error.

SCREENSHOTS







LOGIN

CASE: User with correct email and password

RESULT: The user was logged in successfully.

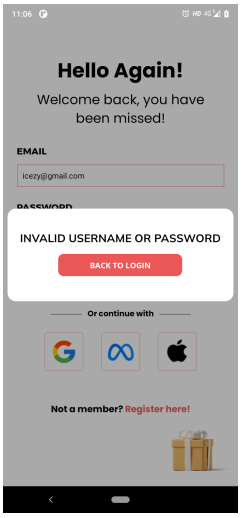
CASE: User with wrong email and password

RESULT: Wrong credentials error

CASE: User with correct email and wrong password

RESULT: Wrong credentials error

SCREENSHOT



HOME PAGE ACCESS:

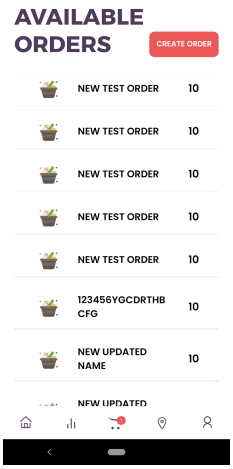
CASE: Tried to access the home page with location permission provided.

RESULT: Correct display of orders on the basis of proximity.

CASE: Tried to access the home page without location permission provided.

RESULT: No orders were shown.

SCREENSHOTS



CREATE ORDER

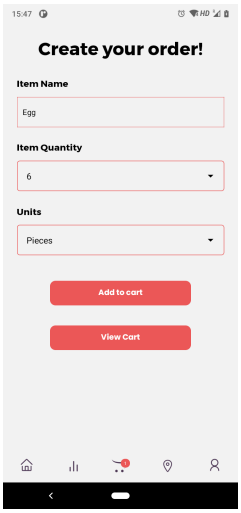
CASE: Create order with all valid information as per the required constraints.

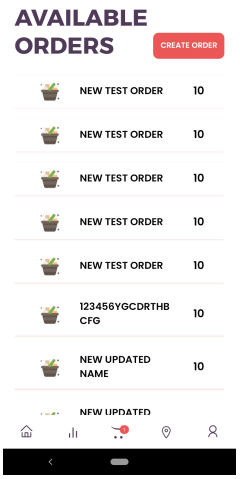
RESULT: The order was created successfully,

CASE: Invalid or missed data while creating order.

RESULT: User was asked to fill the correct details while creating the order.

SCREENSHOTS





ACCEPT ORDER

CASE: Access accept order without logging in

RESULT: Redirected to login page

CASE: Access accept order after logging in

RESULT: Order accepted successfully

TRACKING / MAP

CASE: Accepted the order with location permissions granted.

RESULT: Correct route to the destination was shown.