Test Document

for

Route Mate

Version 1.0

Prepared by

Group 8: **Group Name:** Access_Denied

Palagiri Tousif Ahamad	220744	atousif22@iitk.ac.in
Tamidala Venkata Sai Pawan Chanukya Reddy	221120	venkatasai22@iitk.ac.in
Vetcha Pankaj Nath	221188	vpankaj22@iitk.ac.in
Sudhamandu Sai Nikhil	221095	ssain22@iitk.ac.in
Mohammed Anas	220654	mohdanas22@iitk.ac.in
Himanshu Shekhar	220454	shimanshu22@iitk.ac.in
Swayamsidh Pradhan	221117	spradhan22@iitk.ac.in
Kawaljeet Singh	220514	kawaljeets22@iitk.ac.in
Anya Rajan	220191	anyarajan22@iitk.ac.in
Daksh Kumar Singh	220322	dakshks22@iitk.ac.in

Course: CS253

Mentor TA: Sumit

Date: 29/03/2024

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Access denied team	First draft	31-03-2024
1.1	Access denied team	Final draft	01-04-2024

1 Introduction

Test Strategy

We have automated the Unit Testing and Integration testing using the **Jest** Framework for React/Node. We used the framework to set some test cases which include but are not limited to checking if the components in React are rendered as expected without crashing when rendered under different circumstances and React props, checking if the apis from the Node server is sending responses as expected without fail.

System testing is done manually by hosting the application on a local machine and trying to find any unexpected behaviour and crashes. We tried to find most of the instances during which the application can crash and tried to resolve them.

When was the testing conducted?

During the implementation phase, we tested the software's basic functions to ensure its functionality. The bulk of our testing, however, occurred post-implementation. This approach allowed us to effectively address exceptional circumstances by incorporating conditional blocks. These blocks seamlessly integrate into the software design, requiring only localized changes without disrupting the overall system.

Who were the testers?

Given our constraints, the testers and developers were the same team. However, we adhered to industry standards by ensuring that no feature was tested by the same person or team who implemented it. More specifically, the team which worked on Frontend was involved in Backend testing and similarly the team which worked on Backend was involved in Frontend testing. This approach allowed for a diverse range of perspectives to analyze each feature thoroughly. As a result, we were able to uncover incorrect assumptions made during development. Furthermore, upon identifying these errors, developers were able to anticipate related issues that might arise under exceptional inputs.

What coverage criteria were used?

We used functional coverage for testing.

Tools used for testing

Jest framework for React and Node.

2 Unit Testing

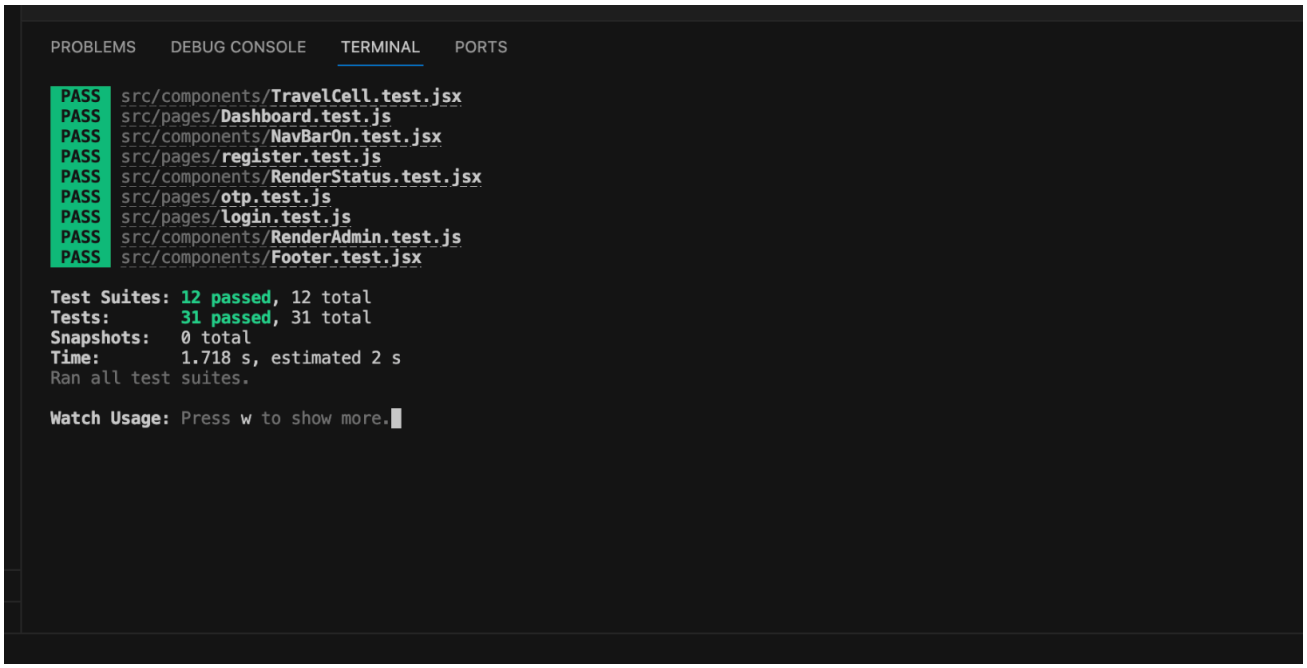
Overview:

Unit Testing is conducted using the **Jest Framework for React** and a **custom bash script for testing the apis from the Node server**. Most of the tests which are related to React are based on checking whether the components are rendered as expected and verifying the validity of the props sent. Any tests which require response from the backend and tests which can't be mocked are omitted.

How to run the tests:

- Open a terminal window in `/client` directory.
- Check if the node modules are installed, and install them if not installed.
- Run '`npm test`'

Test Summary:



Test Suites: 12 Total

Tests: 31 passed

Time: 1.718s

Status: Tests passed

Additional Comments: No Comments

Testing React Components:

1. Login Page rendering:

This modules contain 3 tests which are aimed at testing whether the Login page is rendered as expected without crashing when trying to render. Additionally, two other tests are aimed at testing the contents of the Login page by checking if the 'Sign In' button in the page is rendered and the other test is aimed at testing the presence of 'Register' link which redirects to the register page.

Mocks Used:

- `'NavBarOff'` component is mocked.
- `'Footer'` component is mocked.
- `'Fetch'` request is mocked

login.test.js

client > src > pages > login.test.js > describe('Login') callback

```
import { render, screen } from '@testing-library/react';
import { BrowserRouter as Router } from 'react-router-dom';
import Login from './login';
import '@testing-library/jest-dom';

jest.mock('../components/NavBarOff', () => () => <div>NavBarOff</div>);
jest.mock('../components/Footer', () => () => <div>Footer</div>);

describe('Login', () => {
  test('renders without crashing', () => {
    render(
      <Router>
        <Login />
      </Router>
    );
  });

  test('renders Sign in button', () => {
    render(
      <Router>
        <Login />
      </Router>
    );
    const button = screen.getByText(/Sign in/i);
    expect(button).toBeInTheDocument();
  });

  test('renders Register here link', () => {
    render(
      <Router>
        <Login />
      </Router>
    );
    const link = screen.getByText(/Register here/i);
    expect(link).toBeInTheDocument();
  });
});
```

Module Details:

Test Owner: Tousif

Test Date: 25-03-2024

Test Results: All tests passed

Additional Comments: No Comments

2. Register Page rendering:

This module contains two tests that are focused on verifying the rendering of the Register page. The first test ensures that the Register page is rendered without any crashes. The second test checks the contents of the Register page by verifying the presence of the 'Sign Up' button. These tests are designed to ensure that the Register page and its key components are properly displayed to the user.

Mocks Used:

- `'NavBarOff'` component is mocked.
- `'Footer'` component is mocked.
- `'Fetch'` request is mocked

register.test.js

client > src > pages > register.test.js > ...

```
import { render, screen } from '@testing-library/react';
import { BrowserRouter as Router } from 'react-router-dom';
import Register from './register';
import '@testing-library/jest-dom';

jest.mock('../components/NavBarOff', () => () => <div>NavBarOff</div>);
jest.mock('../components/Footer', () => () => <div>Footer</div>);

// Mock the useNavigate hook
jest.mock('react-router-dom', () => ({
  ...jest.requireActual('react-router-dom'),
  useNavigate: () => jest.fn(),
}));

// Mock the fetch function
global.fetch = jest.fn(() =>
  Promise.resolve({
    json: () => Promise.resolve({ success: true }),
    ok: true,
  })
);

describe('Register', () => {
  beforeEach(() => {
    fetch.mockClear();
  });

  test('renders without crashing', () => {
    render(
      <Router>
        <Register />
      </Router>
    );
  });

  test('renders Sign up button', () => {
    render(
      <Router>
        <Register />
      </Router>
    );
    const button = screen.getByText(/Sign up/i);
    expect(button).toBeInTheDocument();
  });
});
```

Module Details:

Test Owner: Tousif

Test Date: 25-03-2024

Test Results: All tests passed

Additional Comments: No Comments

3. OTP page rendering:

This module contains two tests that are focused on verifying the rendering of the OTP (One-Time Password) page. The first test ensures that the OTP page is rendered without any crashes. The second test checks the contents of the OTP page by verifying the presence of the 'Submit' button. These tests are designed to ensure that the OTP page and its key components are properly displayed to the user.

Mocks Used:

- `'NavBarOff'` component is mocked.
- `'Footer'` component is mocked.
- `'Fetch'` request is mocked

otp.test.js

client > src > pages > otp.test.js > ...

```
import { render, screen } from '@testing-library/react';
import { BrowserRouter as Router } from 'react-router-dom';
import Otp from './otp';
import '@testing-library/jest-dom';

jest.mock('../components/NavBarOff', () => () => <div>NavBarOff</div>);
jest.mock('../components/Footer', () => () => <div>Footer</div>);

// Mock the useNavigate and useLocation hooks
jest.mock('react-router-dom', () => ({
  ...jest.requireActual('react-router-dom'),
  useNavigate: () => jest.fn(),
  useLocation: () => ({ state: { name: 'test', password: 'test', email: 'test@test.com' } }),
}));

// Mock the fetch function
global.fetch = jest.fn(() =>
  Promise.resolve({
    json: () => Promise.resolve({ success: true }),
    ok: true,
  })
);

describe('Otp', () => {
  beforeEach(() => {
    fetch.mockClear();
  });

  test('renders without crashing', () => {
    render(
      <Router>
        <Otp />
      </Router>
    );
  });

  test('renders Submit button', () => {
    render(
      <Router>
        <Otp />
      </Router>
    );
    const button = screen.getByText(/Submit/i);
    expect(button).toBeInTheDocument();
  });
});
```

Module Details:

Test Owner: Tousif

Test Date: 25-03-2024

Test Results: All tests passed

Additional Comments: No Comments

4. Dashboard page rendering:

This module contains a test suite for the DashBoard component. Currently, there is one test that verifies if the DashBoard component renders without crashing. The test uses the render function from '@testing-library/react' to render the DashBoard component within a Router. Mocks have been set up for 'react-router-dom', 'react-bootstrap', and several other components to isolate the DashBoard component for testing.

Mocks Used:

- `'React-Router-DOM'` is mocked.
- `'React-Bootstrap'` is mocked.

Dashboard.test.js M

client > src > pages > Dashboard.test.js > describe('DashBoard') callback

```
import React from 'react';
import { render, fireEvent, waitFor, screen } from '@testing-library/react';
import '@testing-library/jest-dom';
import DashBoard from './DashBoard';
import { BrowserRouter as Router } from 'react-router-dom';

jest.mock('react-router-dom', () => ({
  ...jest.requireActual('react-router-dom'),
  useNavigate: () => jest.fn(),
}));

jest.mock('react-bootstrap', () => ({
  Container: ({ children }) => <div>{children}</div>,
  Row: ({ children }) => <div>{children}</div>,
  Col: ({ children }) => <div>{children}</div>,
}));

jest.mock('../components/NavBarOn', () => () => <div>NavBarOn</div>);
jest.mock('../components/TravelCell', () => () => <div>TravelCell</div>);
jest.mock('../components/trainCell', () => () => <div>TrainCell</div>);
jest.mock('../components/Footer', () => () => <div>Footer</div>);

describe('DashBoard', () => {
  it('renders without crashing', () => {
    render(<Router><DashBoard /></Router>);
  });
});
```

Module Details:

Test Owner: Tousif
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments: No Comments

5. Chatroom page rendering:

This module contains the test suite for the ChatRoom component. It includes the import statement for '@testing-library/jest-dom', which is a library that extends Jest's native 'expect' functionality with more assertions. As it stands, there is only a single test defined for the ChatRoom component which aims to verify if the component is rendered correctly or not.

Mocks Used:

- `'React-Router-DOM'` is mocked.
- `'React-Bootstrap'` is mocked.

Dashboard.test.js M

client > src > pages > Dashboard.test.js > describe('DashBoard') callback

```
1  import React from 'react';
2  import { render, fireEvent, waitFor, screen } from '@testing-library/react';
3  import '@testing-library/jest-dom';
4  import DashBoard from './DashBoard';
5  import { BrowserRouter as Router } from 'react-router-dom';
6
7  jest.mock('react-router-dom', () => ({
8    ...jest.requireActual('react-router-dom'),
9    useNavigate: () => jest.fn(),
10 }));
11
12 jest.mock('react-bootstrap', () => ({
13   Container: ({ children }) => <div>{children}</div>,
14   Row: ({ children }) => <div>{children}</div>,
15   Col: ({ children }) => <div>{children}</div>,
16 }));
17
18 jest.mock('../components/NavBarOn', () => () => <div>NavBarOn</div>);
19 jest.mock('../components/TravelCell', () => () => <div>TravelCell</div>);
20 jest.mock('../components/trainCell', () => () => <div>TrainCell</div>);
21 jest.mock('../components/Footer', () => () => <div>Footer</div>);
22
23 describe('DashBoard', () => {
24   it('renders without crashing', () => {
25     render(<Router><DashBoard /></Router>);
26   });
27 });
```

Module Details:

Test Owner: Daksh
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments: No Comments

6. Navigation Bar rendering when not logged in:

This module contains the test suite for the Navigation Bar component when the user is not logged in. It contains four tests aimed at testing whether different parts of the components are rendered or not. The four tests include testing if the components renders without crashing, testing if the 'Logo' image is displayed, testing if the links for 'Login' and 'Register' page are displayed.

Mocks Used:

- `'Logo.png'` is mocked.

Dashboard.test.js M | NavBarOff.test.jsx

client > src > components > NavBarOff.test.jsx > ...

```jsx
import React from 'react';
import { render, screen } from '@testing-library/react';
import { BrowserRouter } from 'react-router-dom';
import NavBarOff from './NavBarOff';
import '@testing-library/jest-dom';

jest.mock('../Assets/logo.png', () => ({
  default: 'logo.png',
}));

describe('NavBarOff', () => {
  beforeEach(() => {
    render(
      <BrowserRouter>
        <NavBarOff />
      </BrowserRouter>
    );
  });

  test('renders without crashing', () => {
    expect(screen.getByText('Routemate')).toBeInTheDocument();
  });

  test('renders logo', () => {
    expect(screen.getByAltText('Logo')).toBeInTheDocument();
  });

  test('renders login button', () => {
    expect(screen.getByText('Login')).toBeInTheDocument();
  });

  test('renders register button', () => {
    expect(screen.getByText('Register')).toBeInTheDocument();
  });
});
```

Module Details:

Test Owner: Tousif
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments: No Comments

7. Navigation Bar rendering when logged in:

This module contains the test suite for the Navigation Bar component when the user is logged in. It contains five tests aimed at testing whether different parts of the components are rendered or not. The five tests include testing if the components renders without crashing, testing if the 'Logo' image is displayed, testing if the links for 'Dashboard', 'Travel', 'Itinerary' and 'Blogs' pages are displayed correctly, testing if the 'Logout' button is displayed correctly and also testing if the 'Logout' button calls 'handleLogout' function.'

Dashboard.test.js M | NavBarOn.test.jsx

client > src > components > NavBarOn.test.jsx > ...

```jsx
import React from 'react';
import { render, fireEvent, screen } from '@testing-library/react';
import { BrowserRouter } from 'react-router-dom';
import NavBarOn from './NavBarOn';
import '@testing-library/jest-dom';

jest.mock('react-router-dom', () => ({
  ...jest.requireActual('react-router-dom'),
  useNavigate: () => jest.fn(),
}));

describe('NavBarOn', () => {
  beforeEach(() => {
    render(
      <BrowserRouter>
        <NavBarOn />
      </BrowserRouter>
    );
  });

  test('renders without crashing', () => {
    expect(screen.getByText('RouteMate')).toBeInTheDocument();
  });

  test('renders navigation links', () => {
    expect(screen.getByText('Dashboard')).toBeInTheDocument();
    expect(screen.getByText('Travel')).toBeInTheDocument();
    expect(screen.getByText('Itinerary')).toBeInTheDocument();
    expect(screen.getByText('Blogs')).toBeInTheDocument();
  });

  test('renders logo', () => {
    expect(screen.getByAltText('Logo')).toBeInTheDocument();
  });

  test('renders logout button', () => {
    expect(screen.getByText('Logout')).toBeInTheDocument();
  });

  test('calls handleLogout on logout button click', () => {
    const logoutButton = screen.getByText('Logout');
    fireEvent.click(logoutButton);
  });
});
```

Module Details:

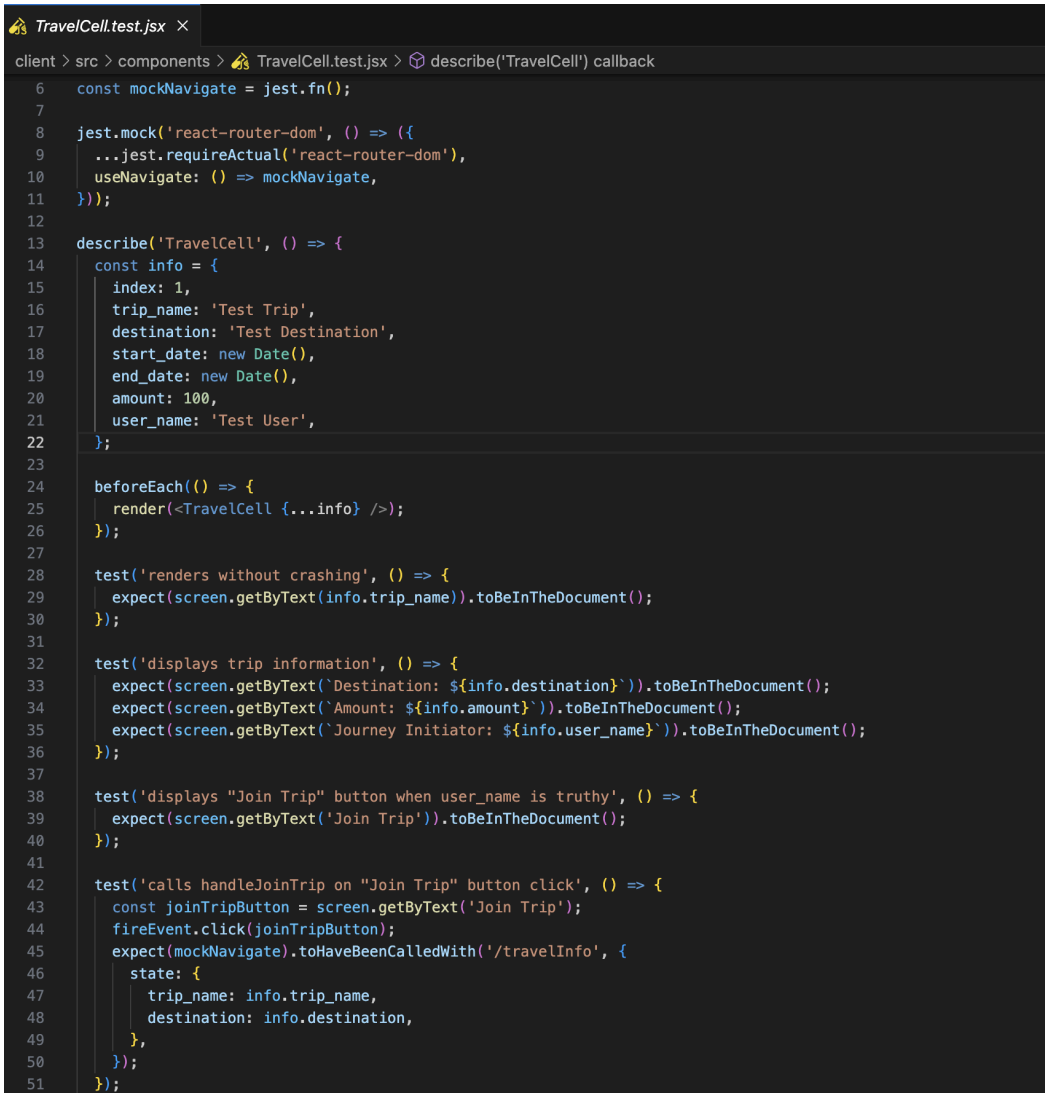
Test Owner: Tousif
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments: No Comments

8. TravelCell component rendering:

This module contains a test suite for the 'TravelCell' component. It includes four tests. The first test ensures that the 'TravelCell' component renders without crashing. The second test verifies that the trip information is correctly displayed. The third test checks if the 'Join Trip' button is displayed when the 'user_name' is truthy. The final test confirms that the 'handleJoinTrip' function is called with the correct parameters when the 'Join Trip' button is clicked. A mock function is used to simulate the 'useNavigate' function from 'react-router-dom'.

Mocks Used:

- `'useNavigate'` is mocked.



Module Details:

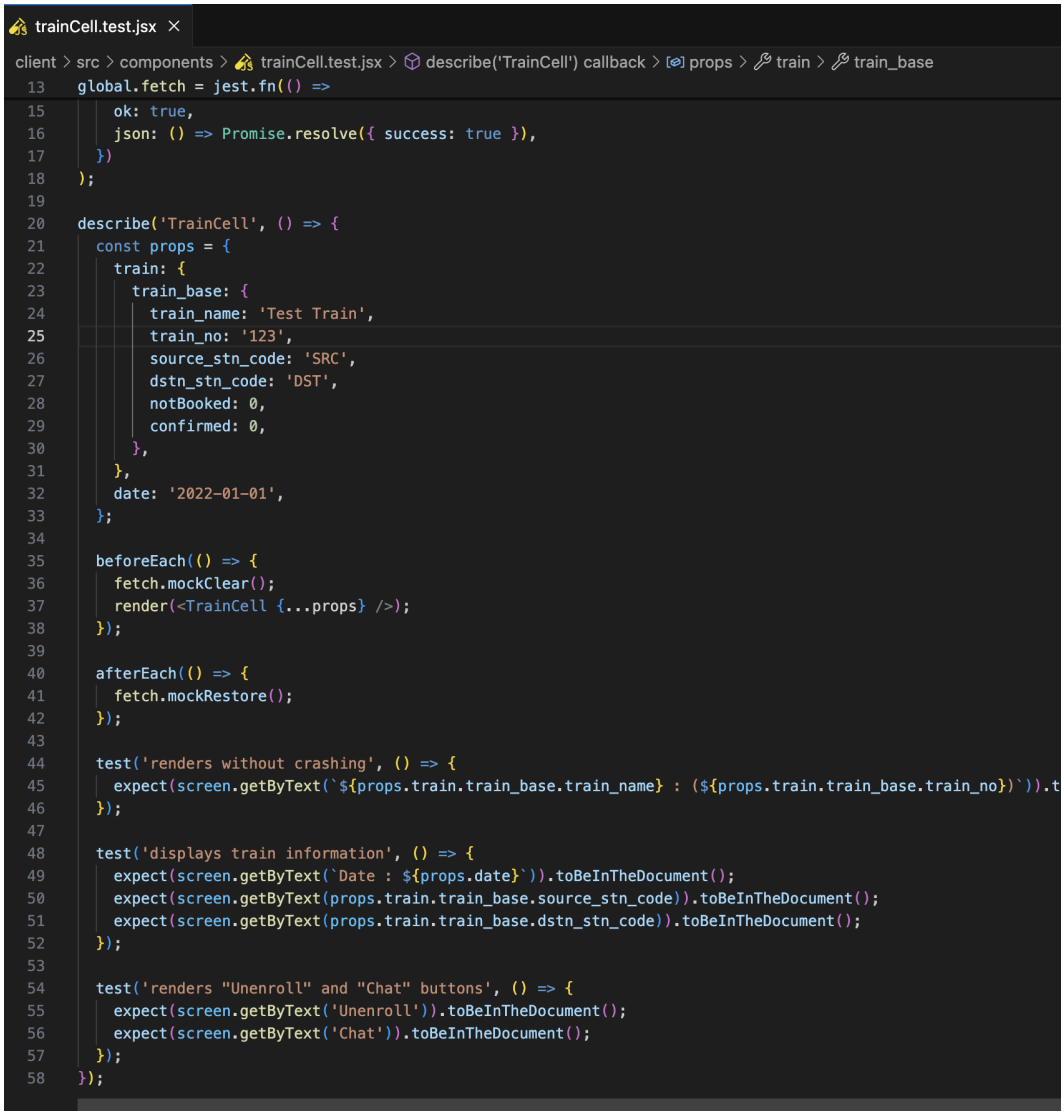
Test Owner: Daksh
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments: No Comments

9. TrainCell component rendering:

This module contains a test suite for the ‘TrainCell’ component. It includes three tests. The first test ensures that the ‘TrainCell’ component renders without crashing. The second test verifies that the trip information is correctly displayed. The third test checks if the 'Unenroll' button and ‘Chat’ button is displayed. A mock function is used to simulate the 'useNavigate' function from 'react-router-dom'.

Mocks Used:

- `‘useNavigate’` is mocked.



Module Details:

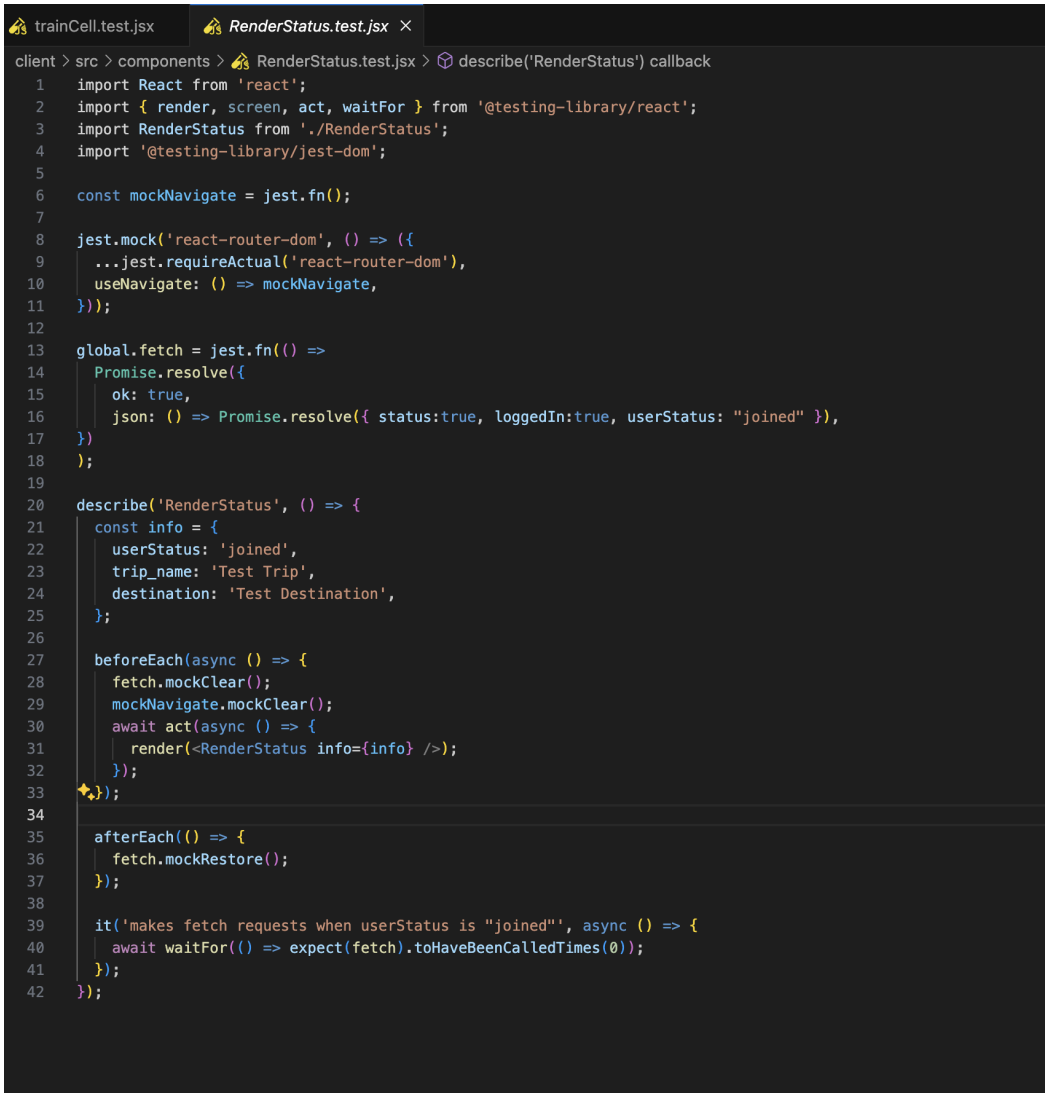
Test Owner: Daksh
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments: No Comments

10. RenderStatus component rendering:

This module contains a test suite for the 'RenderStatus' component. It includes one test currently. The test ensures that no fetch requests are made when the 'userStatus' is 'joined'. The 'fetch' function and 'useNavigate' function from 'react-router-dom' are both mocked to control their behavior during testing. The 'RenderStatus' component is rendered with a prop 'info' that includes 'userStatus', 'trip_name', and 'destination'. The 'fetch' function is cleared before each test and restored after each test to ensure a clean environment for each test.

Mocks Used:

- `'useNavigate'` is mocked.



Module Details:

Test Owner: Daksh
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments: No Comments

11. RenderAdmin component rendering:

This module contains a test suite for the 'RenderAdmin' component. It includes two tests. The first test ensures that the 'RenderAdmin' component renders without crashing and the text elements 'Applied Users', 'Joined Users', and 'Declined Users' are present in the document. The second test verifies that when the 'userStatus' is 'admin', three fetch requests are made to the 'appliedUsers', 'joinedUsers', and 'declinedUsers' endpoints respectively. The 'fetch' function is mocked to control its behavior during testing. The 'RenderAdmin' component is rendered with a prop 'info' that includes 'userStatus', 'trip_name', and 'destination'. The 'fetch' function is cleared before each test and restored after each test to ensure a clean environment for each test.

Mocks Used:

- `'useNavigate'` is mocked.

trainCell.test.jsx | RenderAdmin.test.js

client > src > components > RenderAdmin.test.js > ...

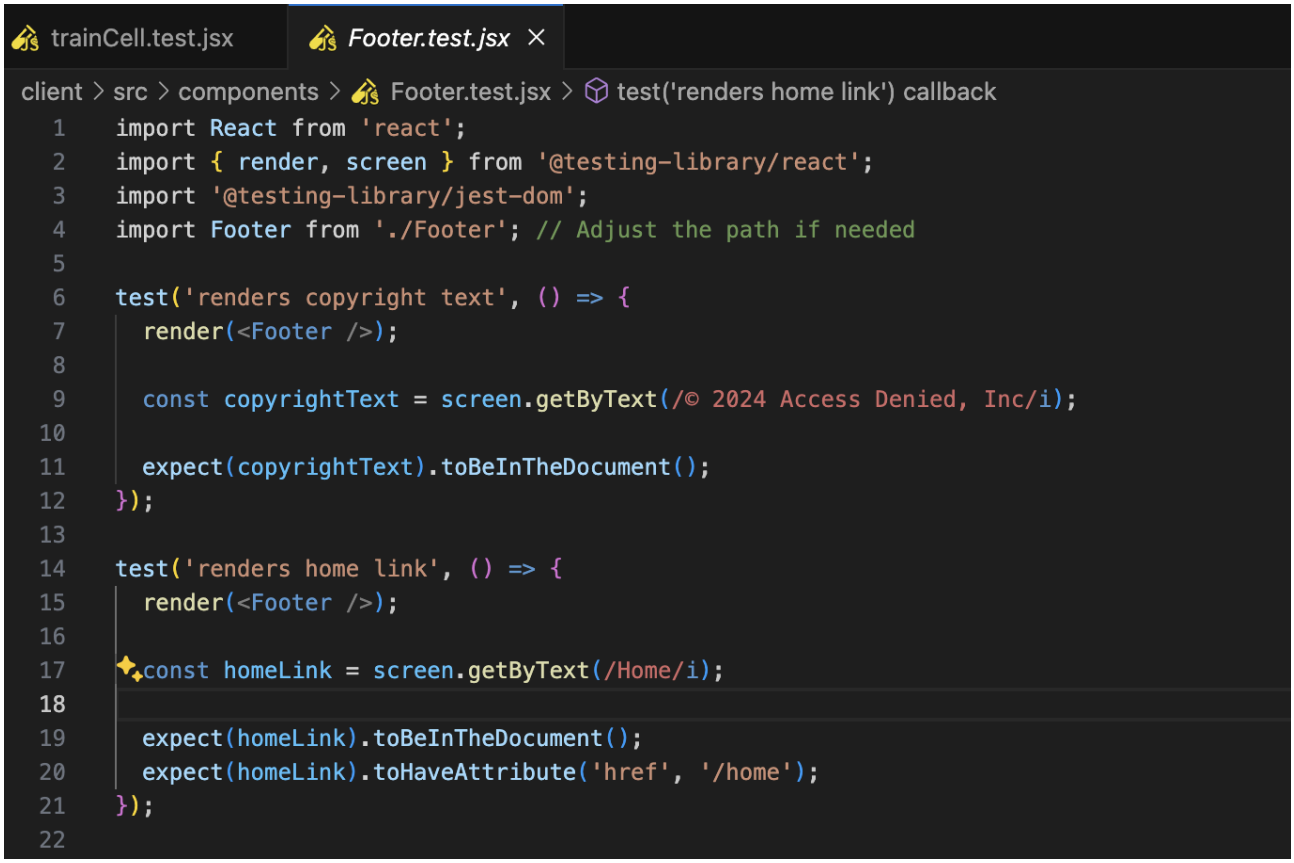
```
beforeEach(() => {
    global.fetch = jest.fn().mockImplementation(() =>
        ok: true,
        json: () => Promise.resolve({ status:true, loggedIn:true, users: [] }),
      })
    );
});

describe('RenderAdmin', () => {
  const info = {
    userStatus: 'admin',
    trip_name: 'Test Trip',
    destination: 'Test Destination',
  };

  beforeEach(async () => {
    fetch.mockClear();
    await act(async () => {
      render(<RenderAdmin info={info} />);
    });
  });


  afterEach(() => {
    fetch.mockRestore();
  });

  test('renders without crashing', () => {
    expect(screen.getByText('Applied Users')).toBeInTheDocument();
    expect(screen.getByText('Joined Users')).toBeInTheDocument();
    expect(screen.getByText('Declined Users')).toBeInTheDocument();
  });

  test('makes fetch requests when userStatus is "admin"', () => {
    expect(fetch).toHaveBeenCalledTimes(3);
    expect(fetch).toHaveBeenCalledWith(`http://${process.env.REACT_APP_IP}:8000/api/travel/appliedUsers?trip_name=${encodeURIComp
      credentials: 'include',
    });
    expect(fetch).toHaveBeenCalledWith(`http://${process.env.REACT_APP_IP}:8000/api/travel/joinedUsers?trip_name=${encodeURICompo
      credentials: 'include',
    });
    expect(fetch).toHaveBeenCalledWith(`http://${process.env.REACT_APP_IP}:8000/api/travel/declinedUsers?trip_name=${encodeURICom
      credentials: 'include',
    });
  });
```

Module Details:

Test Owner: Daksh
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments: No Comments

12. Footer component rendering:

This module contains a test suite for the 'Footer' component. It includes two tests. The first test ensures that the copyright text is correctly rendered in the 'Footer' component. The second test verifies that the 'Home' link is correctly rendered and has the correct 'href' attribute. The 'render' and 'screen' functions from '@testing-library/react' are used to render the 'Footer' component and access its elements, respectively.



Module Details:

Test Owner: Daksh
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments: No Comments

3 Integration Testing

(All the tests conducted to test the application can be found here)

Overview:

To conduct Integration testing we used the **Jest** framework. Most of the tests are automated hence almost all the endpoints are covered. For endpoints that include external apis or interaction with the database, the respective interactions are mocked to improve the efficiency of testing and isolate the test environment from the actual website environment. Here are the modules involved in integration testing.

1. Checking the login Status:

This module tests whether the user is logged in or not logged in. It consists of two tests, one for a user who is logged in and another for a user not logged in. Behavior of passport.js and database are mocked to handle all possible scenarios.

Module Details:

Test Owner: Anya

Test Date: 25-03-2024

Test Results: All tests passed

Additional Comments:

```
describe('/api/login', () => {
  test('responds with json', async () => {
    const mockIsAuthenticated = jest.fn().mockReturnValue(true);
    const mockUser = { name: 'test name' };
    app.request.isAuthenticated = mockIsAuthenticated;
    app.request.user = mockUser;

    const response = await request(app).get('/api/login');
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: true, loggedIn: true, name: mockUser.name });
  });
  test('handles not logged users', async () => {
    const mockIsAuthenticated = jest.fn().mockReturnValue(false);
    app.request.isAuthenticated = mockIsAuthenticated;

    const response = await request(app).get('/api/login');
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: true, loggedIn: false });
  }
);
});
```

Using jest framework, the code handles two tests with mocked behavior of database and passport.js. Correct responses are obtained for all the tests.

2. Adding a new trip :

This module tests whether trips are successfully added to the database. It contains two tests to check if the behavior is the case of both successful operation and error while adding is handled correctly.

Module Details:

Test Owner: Pawan

Test Date: 25-03-2024

Test Results: All tests passed

Additional Comments:

```
describe('POST /api/travel/addTrip', () => {
  let client;
  it('should respond with json and status true when the trip is successfully added', async () => {
    // Arrange
    client = new Client();
    client.query.mockResolvedValueOnce({}); // Insert successful

    // Act
    const response = await request(app)
      .post('/api/travel/addTrip')
      .send({ tripName: 'test trip', destination: 'test destination', startDate: '2022-01-01', endDate:
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: true, loggedIn: false });
  });

  it('should respond with json and status false when there is a database error', async () => {
    // Arrange
    client = new Client();
    client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Database

    // Act
    const response = await request(app)
      .post('/api/travel/addTrip')
      .send({ tripName: 'test trip', destination: 'test destination', startDate: '2022-01-01', endDate:
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: false, error: 'There was an error while adding the trip to t
  });
  afterEach(() => {
    client.query.mockReset();
  });
});
```

Using jest framework, code handles two tests with mocked behavior of database to simulate all the conditions and all the tests were passed.

3. GET /api/travel/trips

This module tests the functionality of the endpoint GET /api/travel/trips. It verifies two scenarios: one where the trips are successfully retrieved from the database and another where there is a database error.

Module Details:

Test Owner: Pawan
Test Date: 25-03-2024
Test Results: All tests passed
Additional Comments:

```
describe('GET /api/travel/trips', () => {
  let client;
  it('should respond with json and status true when the trips are successfully retrieved', async () => {
    // Arrange
    const mockTrips = [{ id: 1, name: 'test trip' }, { id: 2, name: 'another test trip' }];
    client = new Client();
    client.query.mockResolvedValueOnce({ rows: mockTrips }); // Mock the database query

    // Act
    const response = await request(app).get('/api/travel/trips');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: true, loggedIn: true, trips: mockTrips });
  });

  it('should respond with json and status false when there is a database error', async () => {
    // Arrange
    client = new Client();
    client.query.mockRejectedValueOnce(new Error('Database error')); // Mock a database error

    // Act
    const response = await request(app).get('/api/travel/trips');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving trips fro
  });
  afterEach(() => {
    client.query.mockReset();
  });
});
```

```
var Error: ErrorConstructor
new (message?: string | undefined) => Error
```

This module mocks the request and database response to simulate all the possible cases in the endpoint and checks the responses.

4. GET /api/travel/joinedTrips

This module comprises tests for retrieving the joined trips of a user from the database, validating both successful and error database responses. It utilizes mocked database behavior to simulate various scenarios, alongside verifying user authentication status.

Module Details:

Test Owner: Pawan
Test Date: 26-03-2024
Test Results: All tests passed
Additional Comments:

```
describe('GET /api/travel/joinedTrips', () => {
  let client;

  beforeEach(() => {
    client = new Client();
    jest.spyOn(client, 'query');
    app.set('db', client); // Make sure your app uses this client
  });

  it('should respond with json and status true when the trips are successfully retrieved', async () => {
    // Arrange
    const mockTrips = [{ id: 1, name: 'test trip' }, { id: 2, name: 'another test trip' }];
    const mockIsAuthenticated = jest.fn().mockReturnValue(true);
    app.request.isAuthenticated = mockIsAuthenticated;
    client.query.mockResolvedValueOnce({ rows: mockTrips }); // Mock the database query

    // Act
    const response = await request(app)
      .get('/api/travel/joinedTrips')
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: true, loggedIn: true, trips: mockTrips });
  });

  it('should respond with json and status false when there is a database error', async () => {
    // Arrange
    client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

    // Act
    const response = await request(app)
      .get('/api/travel/joinedTrips')
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving trips fro
  });

  afterEach(() => {
    client.query.mockReset();
  });
```

This module contains two tests to check the behavior of the endpoint at different scenarios simulated by mocking the database and request.

5. GET /api/travel/hostedTrips :

This module comprises tests for retrieving the hosted trips of a user from the database, validating both successful and error database responses. It utilizes mocked database behavior to simulate various scenarios, alongside verifying user authentication status.

Module Details:

Test Owner: Pawan
Test Date: 26-03-2024
Test Results: All tests passed
Additional Comments:

```
it('should respond with json and status true when the trips are successfully retrieved', async () => {
  // Arrange
  const mockTrips = [{ id: 1, name: 'test trip' }, { id: 2, name: 'another test trip' }];
  const mockIsAuthenticated = jest.fn().mockReturnValue(true);
  client.query.mockResolvedValueOnce({ rows: mockTrips }); // Mock the database query
  app.request.isAuthenticated = mockIsAuthenticated;

  // Act
  const response = await request(app)
    .get('/api/travel/hostedTrips')
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: true, loggedIn: true, trips: mockTrips });
});

it('should respond with json and status false when there is a database error', async () => {
  // Arrange
  client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

  // Act
  const response = await request(app)
    .get('/api/travel/hostedTrips')
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving trips fro
});
```

This module contains two tests to check the behavior of the endpoint at different scenarios simulated by mocking the database and request.

6. GET /api/travel/specificTrip :

This module comprises tests for retrieving the specific trips as required by a user from the database, validating both successful and error database responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Anya
Test Date: 26-03-2024
Test Results: All tests passed
Additional Comments:

```
it('should respond with json and status true when the trip is successfully retrieved', async () => {
  // Arrange
  const mockTrip = [{ id: 1, name: 'test trip', destination: 'test destination' }];
  client.query.mockResolvedValueOnce({ rows: mockTrip }); // Mock the database query

  // Act
  const response = await request(app)
    .get('/api/travel/specificTrip')
    .query({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: true, loggedIn: true, trips: mockTrip });
});

it('should respond with json and status false when there is a database error', async () => {
  // Arrange
  client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

  // Act
  const response = await request(app)
    .get('/api/travel/specificTrip')
    .query({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving trips fro
});
```

This module contains two tests to check the behavior of the endpoint at different scenarios simulated by mocking the database and request.

7. GET /api/travel/searchTrip :

This module comprises tests for retrieving the trips that satisfy the searched condition entered by the user from the database, validating both successful and error database responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Nikhil
Test Date: 26-03-2024
Test Results: All tests passed
Additional Comments:

```
it('should respond with json and status true when the trips are successfully retrieved', async () => {
  // Arrange
  const mockTrips = [{ id: 1, name: 'test trip', destination: 'test destination' }];
  client.query.mockResolvedValueOnce({ rows: mockTrips }); // Mock the database query

  // Act
  const response = await request(app)
    .get('/api/travel/searchTrip')
    .query({ search: 'test' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: true, loggedIn: true, trips: mockTrips });
});

it('should respond with json and status false when there is a database error', async () => {
  // Arrange
  client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

  // Act
  const response = await request(app)
    .get('/api/travel/searchTrip')
    .query({ search: 'test' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving trips fro
});
```

This module contains two tests to check the behavior of the endpoint at different scenarios simulated by mocking the database and request.

8. GET /api/travel/userStatus :

This module comprises tests for checking the user status in the trip by retrieving it from the database, validating both successful and error database responses. It utilizes mocked database behavior to simulate various scenarios, alongside verifying user authentication status.

Module Details:

Test Owner: Nikhil
Test Date: 27-03-2024
Test Results: All tests passed
Additional Comments:

```
it('should respond with json and status true when the user status is successfully retrieved', async ()
  // Arrange
  const mockUserStatus = [{ status: 1 }];
  client.query.mockResolvedValueOnce({ rows: mockUserStatus }); // Mock the database query

  // Act
  const response = await request(app)
    .get('/api/travel/userStatus')
    .query({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: true, loggedIn: true, userStatus: 'admin' });
});

it('should respond with json and status false when there is a database error', async () => {
  // Arrange
  client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a dat

  // Act
  const response = await request(app)
    .get('/api/travel/userStatus')
    .query({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving user statu
});
```

This module contains two tests to check the behavior of the endpoint at different scenarios simulated by mocking the database and request.

9. GET /api/travel/applyToJoin :

This module comprises tests for checking the behavior of the server when a user applies to join a trip by retrieving and using information from the database, validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios, alongside verifying user authentication status.

Module Details:

Test Owner: Nikhil
Test Date: 27-03-2024
Test Results: All tests passed
Additional Comments:

```
it('should respond with json and status true when the user successfully applies to join the trip', asy
  // Arrange
  client.query.mockResolvedValueOnce({ rows: [] }); // Mock the database query to return no existing a
  client.query.mockResolvedValueOnce({}); // Mock the database query to insert the application

  // Act
  const response = await request(app)
    .post('/api/travel/applyToJoin')
    .send({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: true, loggedIn: true });
});

it('should respond with json and status false when the user has already applied to join the trip', asy
  // Arrange
  client.query.mockResolvedValueOnce({ rows: [{}] }); // Mock the database query to return an existing

  // Act
  const response = await request(app)
    .post('/api/travel/applyToJoin')
    .send({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: false, error: 'You have already applied to join this trip.'
});

it('should respond with json and status false when there is a database error', async () => {
  // Arrange
  client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

  // Act
  const response = await request(app)
    .post('/api/travel/applyToJoin')
    .send({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');
```

This module consists of three tests for checking if the user is logged in , logged out and error in database response. This will exhaust all the possibilities at the end point.

10. GET /api/travel/appliedUsers :

This module comprises tests for checking the behavior of the server when a user tries to retrieve information about users in a trip from the database, validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios, alongside verifying user authentication status.

Module Details:

Test Owner: Nikhil
Test Date: 27-03-2024
Test Results: All tests passed
Additional Comments:

```
describe('GET /api/travel/appliedUsers', () => {
  beforeEach(() => {
    client = new Client();
    jest.spyOn(client, 'query');
    app.set('db', client); // Make sure your app uses this client
  });

  it('should respond with json and status true when the applied users are successfully retrieved', async
    // Arrange
    const mockUsers = [{ user_id: 1, trip_name: 'test trip', destination: 'test destination', status: 3
    client.query.mockResolvedValueOnce({ rows: mockUsers }); // Mock the database query

    // Act
    const response = await request(app)
      .get('/api/travel/appliedUsers')
      .query({ trip_name: 'test trip', destination: 'test destination' })
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: true, loggedIn: true, users: mockUsers });
  });

  it('should respond with json and status false when there is a database error', async () => {
    // Arrange
    client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

    // Act
    const response = await request(app)
      .get('/api/travel/appliedUsers')
      .query({ trip_name: 'test trip', destination: 'test destination' })
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving applied u
  });

  afterEach(() => {
    client.query.mockReset();
  });
});
```

This module contains two tests to check if the end point behaves correctly in all the situations simulated by mocking the database and passport.js.

11. POST /api/travel/addUserToTrip :

This module comprises tests for checking the behavior of the server when a user tries to join a trip by adding information about users in a trip to the database, validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Anya
Test Date: 27-03-2024
Test Results: All tests passed
Additional Comments:

```
describe('POST /api/travel/addUserToTrip', () => {
  let client;

  beforeEach(() => {
    client = new Client();
    jest.spyOn(client, 'query');
    app.set('db', client); // Make sure your app uses this client
  });

  it('should respond with json and status true when the user is successfully added to the trip', async (
    // Arrange
    client.query.mockResolvedValueOnce({}); // Mock the database query

    // Act
    const response = await request(app)
      .post('/api/travel/addUserToTrip')
      .send({ trip_name: 'test trip', destination: 'test destination', user_id: 1 })
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: true, loggedIn: true });
  });

  it('should respond with json and status false when there is a database error', async () => {
    // Arrange
    client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

    // Act
    const response = await request(app)
      .post('/api/travel/addUserToTrip')
      .send({ trip_name: 'test trip', destination: 'test destination', user_id: 1 })
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: false, error: 'There was an error while adding the user to t
  });

  afterEach(() => {
    client.query.mockReset();
  });
});
```

This module contains two tests to check if the end point behaves correctly in all the situations simulated by mocking the database.

12. POST /api/travel/declineUserToTrip :

This module comprises tests for checking the behavior of the server when a user is rejected by the host of a trip to join in the trip , validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Pankaj
Test Date: 27-03-2024
Test Results: All tests passed
Additional Comments:

```
describe('POST /api/travel/declineUserToTrip', () => {
  let client;

  beforeEach(() => {
    client = new Client();
    jest.spyOn(client, 'query');
    app.set('db', client); // Make sure your app uses this client
  });

  it('should respond with json and status true when the user is successfully declined from the trip', as
    // Arrange
    client.query.mockResolvedValueOnce({}); // Mock the database query

    // Act
    const response = await request(app)
      .post('/api/travel/declineUserToTrip')
      .send({ trip_name: 'test trip', destination: 'test destination', user_id: 1 })
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: true, loggedIn: true });
  });

  it('should respond with json and status false when there is a database error', async () => {
    // Arrange
    client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

    // Act
    const response = await request(app)
      .post('/api/travel/declineUserToTrip')
      .send({ trip_name: 'test trip', destination: 'test destination', user_id: 1 })
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: false, error: 'There was an error while declining the user t
  });

  afterEach(() => {
    client.query.mockReset();
  });
```

This module contains two tests to check if the end point behaves correctly in all the situations simulated by mocking the database.

13. GET /api/travel/joinedUsers :

This module comprises tests for checking the behavior of the server when host of a trip queries to get all the users joined in the trip , validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Pawan
Test Date: 27-03-2024
Test Results: All tests passed
Additional Comments:

```
it('should respond with json and status true when the joined users are successfully retrieved', async
  // Arrange
  const mockUsers = [{ user_id: 1, trip_name: 'test trip', destination: 'test destination', status: 2
  client.query.mockResolvedValueOnce({ rows: mockUsers }); // Mock the database query

  // Act
  const response = await request(app)
    .get('/api/travel/joinedUsers')
    .query({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: true, loggedIn: true, users: mockUsers });
});

it('should respond with json and status false when there is a database error', async () => {
  // Arrange
  client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

  // Act
  const response = await request(app)
    .get('/api/travel/joinedUsers')
    .query({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving joined us
});

afterEach(() => {
  client.query.mockReset();
});
```

This module contains two tests to check if the end point behaves correctly in all the situations simulated by mocking the database.

14. GET /api/travel/declinedUsers :

This module comprises tests for checking the behavior of the server when the host of tripm queries to get all the users declined in the trip , validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Nikhil
Test Date: 27-03-2024
Test Results: All tests passed
Additional Comments:

```
it('should respond with json and status true when the declined users are successfully retrieved', asyn
  // Arrange
  const mockUsers = [{ user_id: 1, trip_name: 'test trip', destination: 'test destination', status: 4
  client.query.mockResolvedValueOnce({ rows: mockUsers }); // Mock the database query

  // Act
  const response = await request(app)
    .get('/api/travel/declinedUsers')
    .query({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: true, loggedIn: true, users: mockUsers });
});

it('should respond with json and status false when there is a database error', async () => {
  // Arrange
  client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a da

  // Act
  const response = await request(app)
    .get('/api/travel/declinedUsers')
    .query({ trip_name: 'test trip', destination: 'test destination' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving declined
});

afterEach(() => {
  client.query.mockReset();
});
```

This module contains two tests to check if the end point behaves correctly in all the situations simulated by mocking the database.

15. POST /api/travel/addChat :

This module comprises tests for checking the behavior of the server when the user tries to post a message in the chat server , validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Pankaj
Test Date: 28-03-2024
Test Results: All tests passed
Additional Comments:

```
beforeEach(() => {
  client = new Client();
  jest.spyOn(client, 'query');
  app.set('db', client); // Make sure your app uses this client
  app.set('user', { name: 'test user' }); // Mock the user
});

it('should respond with json and status true when the chat is successfully added', async () => {
  // Arrange
  client.query.mockResolvedValueOnce({}); // Mock the database query

  // Act
  const response = await request(app)
    .post('/api/travel/addChat')
    .send({ trip_name: 'test trip', destination: 'test destination', message: 'test message' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: true, loggedIn: true });
});

it('should respond with json and status false when there is a database error', async () => {
  // Arrange
  client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a d

  // Act
  const response = await request(app)
    .post('/api/travel/addChat')
    .send({ trip_name: 'test trip', destination: 'test destination', message: 'test message' })
    .set('Accept', 'application/json');

  // Assert
  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({ status: false, error: 'There was an error while adding the chat to
});
```

This module contains two tests to check if the end point behaves correctly in all the situations simulated by mocking the database.

16. GET /api/travel/chats :

This module comprises tests for checking the behavior of the server when the user tries to retrieve all the messages posted in the chat server , validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Pankaj
Test Date: 28-03-2024
Test Results: All tests passed
Additional Comments:

```
  it('should respond with json and status true when the chats are successfully retrieved', async () =>
    // Arrange
    const mockChats = [{ username: 'test user', message: 'test trip test destination test message' }];
    client.query.mockResolvedValueOnce({ rows: mockChats }); // Mock the database query

    // Act
    const response = await request(app)
      .get('/api/travel/chats')
      .query({ trip_name: 'test trip', destination: 'test destination' })
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: true, loggedIn: true, chats: mockChats });
  });

  it('should respond with json and status false when there is a database error', async () => {
    // Arrange
    client.query.mockImplementationOnce(() => Promise.reject(new Error('Database error'))); // Mock a d

    // Act
    const response = await request(app)
      .get('/api/travel/chats')
      .query({ trip_name: 'test trip', destination: 'test destination' })
      .set('Accept', 'application/json');

    // Assert
    expect(response.statusCode).toBe(200);
    expect(response.body).toEqual({ status: false, error: 'There was an error while retrieving chats fr
  });

  afterEach(() => {
    client.query.mockReset();
  });
});
```

This module contains two tests to check if the end point behaves correctly in all the situations simulated by mocking the database.

17. POST /verifyOTP :

This module comprises tests for checking the behavior of the server when it tries to verify the OTP entered by the user , validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Pankaj
Test Date: 28-03-2024
Test Results: All tests passed
Additional Comments:

```
test('should verify OTP successfully', async () => {
  const mockRequest = {
    body: {
      email: 'test@example.com',
      OTP: '123456',
    },
  };

  const mockSelectResponse = {
    rows: [{ email: 'test@example.com', OTP: '123456' }],
  };

  client.query.mockResolvedValueOnce(mockSelectResponse); // Mock the SELECT query
  client.query.mockResolvedValueOnce(); // Mock the DELETE query

  const response = await request(app)
    .post('/verifyOTP')
    .send(mockRequest.body);

  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({
    status: true,
    success: true,
    message: 'OTP verified successfully',
  });
});
```

This module contains two tests to check if the end point behaves correctly in both the situations when correct OTP is entered and incorrect OTP is entered.

18. POST /addNotBookedTrainUser :

This module comprises tests for checking the behavior of the server when a user tries to enroll in a train under the not-booked section , validating both successful and error responses. It utilizes mocked database behavior to simulate various scenarios.

Module Details:

Test Owner: Pankaj
Test Date: 28-03-2024
Test Results: All tests passed
Additional Comments:

```
it('should add not booked train user successfully', async () => {
  const mockRequest = {
    body: {
      train: {
        train_base: {
          train_no: '123',
        },
      },
      date: '2022-01-01',
      origin: 'originStation',
      destination: 'destinationStation',
    },
    user: {
      id: 'userId',
    },
  };
  const mockIsAuthenticated = jest.fn().mockReturnValue(true);
  app.request.isAuthenticated = mockIsAuthenticated;
  const mockDbResponse = {
    rows: [],
  };

  client.query.mockResolvedValueOnce(mockDbResponse); // For the first query
  client.query.mockResolvedValueOnce(mockDbResponse); // For the second query
  client.query.mockResolvedValueOnce(mockDbResponse); // For the third query
  client.query.mockResolvedValueOnce(mockDbResponse); // For the fourth query
  client.query.mockResolvedValueOnce(mockDbResponse); // For the fifth query
  client.query.mockResolvedValueOnce({ rows: [{ yet_to_book: 0 }] }); // For the sixth query
  client.query.mockResolvedValueOnce({ rows: [{ booked: 0 }] }); // For the seventh query

  const response = await request(app)
    .post('/addNotBookedTrainUser')
    .send(mockRequest.body);

  expect(response.statusCode).toBe(200);
  expect(response.body).toEqual({
    status: true,
    success: true,
    loggedIn: true,
    notBooked: 0,
    confirmed: 0,
  });
});
```

This module contains one test to check if the server correctly adds the user to the database and returns the response.

4 System Testing

1. **Requirement:** Site Launches properly and all the images and text are rendered properly on the website.

Test Owner: Swayamsidh Pradhan
Test Date: 26-03-2024
Test Results: The site was launched in localhost from our system. We tried launching the site on Windows and MacOS as well as on all the popular browsers(Chrome, Firefox, Safari,Edge e.t.c).
Additional Comments: NA

2. **Requirement:** Users can create accounts using the sign up page.

Test Owner: Swayamsidh Pradhan
Test Date: 26-03-2024
Test Results: Users can successfully create an account using the sign up page. OTP is correctly sent to the given email address. Invalid inputs in the registration form are rejected. If some random email address is provided then the user won't be able to get the OTP.
Additional Comments: Signing up with an Email Id that is already registered is not allowed, the user is directed to the home page in that case. However, two users can share the same Username. Registration works for any Email Id not just IITK Email Id and for that reason we do not ask for IITK Roll Number. Prevents SQL injection and cross-site scripting (XSS) attacks.

3. **Requirement:** Users can login to the web application if correct credentials are provided.

Test Owner: Swayamsidh Pradhan
Test Date: 27-03-2024
Test Results: Users are successfully able to login to the web application if correct credentials are provided. If the login is not successful then it redirects the user back to the login page. Rejects invalid Email Ids but takes in empty fields as valid inputs.
Additional Comments: The Email Id does not necessarily have to be an IITK Email Id and also the IITK Roll Number is not required for login. Prevents SQL injection and cross-site scripting (XSS) attacks.

4. **Requirement:** User gets automatically logged out after 7 days.

Test Owner: Swayamsidh Pradhan
Test Date: 27-03-2024
Test Results: The maxAge for cookies is set to 7 days, so the user is automatically logged out after 7 days. This feature was tested by setting the maxAge to 5s, 10s and 30s. The tests passed for these dummy values.
Additional Comments: The user stays on the page he/she is currently on even after the cookies expire and is redirected back to the home page and is logged out from the web application only when some action is taken.

5. **Requirement:** Users can log out from the web application.

Test Owner: Swayamsidh Pradhan
Test Date: 27-03-2024
Test Results: The user is able to logout from any of the main pages like Dashboard, Travel, Itinerary and Blogs. Users are prevented from logging out in between some action. On logging out the user is redirected to the home page and he/she cannot access the application again by changing the URL.
Additional Comments: NA

6. **Requirement:** Users can add trips on the travel page.

Test Owner: Swayamsidh Pradhan
Test Date: 28-03-2024
Test Results: Users are able to create a new trip by filling in all the required details. Invalid details, like negative amounts or having the end date before the start date, are rejected.
Additional Comments: The user must provide an image for the trip.

7. **Requirement:** Travel page is maintained in real time.

Test Owner: Himanshu Shekhar
Test Date: 28-03-2024
Test Results: Trips created by all the users are present on the Travel page. The search bar searches for a specific trip based on the Trip Title. A user can check his/her hosted trips and can see/update the applied, joined and declined users. Other users can apply to join a trip and are shown their current application status.
Additional Comments: NA

8. **Requirement:** Users can search for trains.

Test Owner: Himanshu Shekhar
Test Date: 28-03-2024
Test Results: Users are able to search for trains given the Origin, Destination and the Departure Data. Empty fields are rejected.
Additional Comments: The search feature for flights is yet to be developed. We were not able to find an API that gave flight details.

9. **Requirement:** Users can add blogs on the Blog Page.

Test Owner: Himanshu Shekhar
Test Date: 28-03-2024
Test Results: Users are able to create blogs on the Blog Page. Users are asked to enter the markdown for the content and are shown the Preview in real-time. Empty fields are rejected.
Additional Comments: There is no word limit on the Blog title and the Blog content.

10. **Requirement:** Blog Page is maintained in real time.

Test Owner: Himanshu Shekhar
Test Date: 29-03-2024
Test Results: Blogs created by all the users are present on the Blog page. The search bar searches for a specific blog based on phrases in Blog title, Blog content or Username of the user who created the blog.
Additional Comments: There is no word limit on the Blog title and the Blog content.

11. **Requirement:** Users can chat in real time.

Test Owner: Himanshu Shekhar
Test Date: 29-03-2024
Test Results: Users are able to access the chat server of the trip they are in and can chat with others in real time. Users can also access the chat servers of Booked and Not Booked trains.
Additional Comments: There is no word limit on the chat.

12. **Requirement:** Users can change the status of their upcoming journeys.

Test Owner: Himanshu Shekhar
Test Date: 29-03-2024
Test Results: Users are able to change the status of their upcoming train journeys in the dashboard page and their status is updated and modified for other users.
Additional Comments: The change is not visible instantly but is visible after reloading the webpage.

5 Conclusion

How Effective and exhaustive was the testing?

Most of the unit and integration testing has been automated using Jest framework covering almost all of the components and end-points. All the requirements specified in the Requirement document have been carefully examined and documented during the system testing. Using the Jest framework has been very effective in ensuring all the components were behaving as expected.

Which components have not been tested adequately?

Some of the components and end-points that require the use of third-party APIs and frameworks like “indian-rail-api”, “reactMarkdown”, “otp-generator” etc had to be tested manually as their behavior was difficult to mock while testing. Hence these components were excluded from the automation process and have not been tested adequately.

What difficulties have you faced during testing?

Main problems that we faced while testing was to learn how to mock the behavior of various components but after carefully configuring the test environment the testing phase has been relatively smooth. Some problems arise while dealing with third-party APIs and frameworks but they have been dealt with manually ensuring they were free of bugs as much as possible.

How could the testing process be improved?

Testing process can be improved by completely removing the manual testing and fully automating it to have better code coverage. Also separating the testing team from the developers team would have been more productive as either team can work on their respective domains irrespective of the other team.

Appendix A - Group Log

SNo	Date	Agenda	Duration	Members Attended
1	23-03-2024	Discussed which tools to use for testing	60 min	10
2	24-03-2024	Divided the tasks among ourselves	90 min	10
3	29-03-2024	Finished all the tests and started documenting the tests	60 min	10
4	31-03-2024	Made first draft of test document	60 min	10
5	01-04-2024	Made final draft of the document	60 min	10