M4-Logic Legends

Frontend Testing:

For front-end testing we are using Cypress, which is an open-source end-to-end testing framework for web applications. It enables developers to write, run, and debug tests that simulate user interactions within the browser. Cypress provides a powerful and intuitive testing experience with features like automatic waiting, real-time reloads, and built-in assertions. It's particularly well-suited for testing modern web applications, including those built with frameworks like React, Angular, and Vue.js. Because our web application is web based we decided to go with cypress.

Frontend Approaches used:

- Component Testing: Cypress allows you to write tests that interact with individual React components. So we have written tests to ensure that our components render correctly, handle user interactions, and update state as expected.

- End-to-End (E2E) Testing: Cypress is well-suited for E2E testing, So we are simulating user interactions across multiple components and pages to ensure that our application functions correctly as a whole. We have written tests to simulate user journeys, such as signing up, logging in, or completing a purchase.

- Visual Regression Testing: With Cypress, we can incorporate visual regression testing to detect unintended visual changes in your application. This involves taking screenshots of our application during tests and comparing them against baseline images to identify any visual discrepancies.

- API Testing: While Cypress primarily focuses on frontend testing, we can also use it to test backend APIs that your application relies on. We have written tests to verify that API endpoints return the expected data and handle various edge cases correctly.

Future Improvements:

- Parameterized Tests: Parameterize your tests to make them more flexible and reusable. Instead of hardcoding test data or interaction sequences, use parameters to test a variety of inputs and scenarios.

- Mocking and Stubbing: Incorporate mocking and stubbing techniques to isolate your tests from external dependencies such as APIs or third-party services. This can improve test reliability and speed by removing external dependencies.

- Performance Testing: Consider adding performance testing to your testing strategy to ensure that your application meets performance requirements under various conditions. This could involve measuring page load times, network requests, or rendering performance.

What works?

As of the time of submitting this assignment, our front end testing covers all the major processes that have been implemented so far. For instance, the process of buying a ticket for a new event and getting the confirmation for that event. Getting your Event tickets that you have bought in your profile. The entire process of creating a new account and logging in. Furthermore, the homepage and other pages are also tested for functionality and rendering of visual elements. All these tests have worked and what does not pass these test have been corrected and made sure all current tests are being passed.

Example code for testing registering process:

```
describe('Signup Page', () => {
 beforeEach(() => {
  cy.visit('http://localhost:3000/sign-up');
 });

 it('renders signup form correctly', () => {
  cy.contains('h2', 'Sign Up').should('exist');
  cy.get('input[name="email"]').should('exist');
  cy.get('input[name="password"]').should('exist');
  cy.get('input[name="firstName"]').should('exist');
  cy.get('input[name="lastName"]').should('exist');
  cy.get('select[name="country"]').should('exist');
  cy.get('input[name="zipCode"]').should('exist');
  cy.contains('button', 'Sign Up').should('exist');
  cy.contains('Already have an Eventify Account? Login').should('exist');
  cy.contains('Login in as Admin?').should('exist');
 });

 it('allows user to sign up', () => {
  // Fill in the signup form
  cy.get('input[name="email"]').type('test@example.com');
  cy.get('input[name="password"]').type('testpassword');
  cy.get('input[name="firstName"]').type('John');
  cy.get('input[name="lastName"]').type('Doe');
  cy.get('select[name="country"]').select('Canada');
  cy.get('input[name="zipCode"]').type('12345');
```

```
  // Submit the form
  cy.contains('button', 'Sign Up').click();

  // Check if the user is navigated to the login page after successful signup
  cy.url().should('include', '/login');
 });

 it('displays error message for unsuccessful signup attempt', () => {
  // Intercept the POST request to register and force it to fail
  cy.intercept('POST', 'http://localhost:3002/register', {
   statusCode: 400,
   body: { message: 'Error creating account.' }
  }).as('signupFailure');

  // Fill in the signup form
  cy.get('input[name="email"]').type('test@example.com');
  cy.get('input[name="password"]').type('testpassword');
  cy.get('input[name="firstName"]').type('John');
  cy.get('input[name="lastName"]').type('Doe');
  cy.get('select[name="country"]').select('Canada');
  cy.get('input[name="zipCode"]').type('12345');

  // Submit the form
  cy.contains('button', 'Sign Up').click();

  // Check if the error message is displayed
  cy.contains('Error creating account.').should('exist');

  // Check if the user is not navigated to the login page after unsuccessful signup
  cy.url().should('not.include', '/login');
 });
});
```

Backend Testing:

To perform backend testing we have used Mocha and Chai. Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases. Mocha provides the test environment and the ability to describe test suites (describe blocks) and test cases (it blocks).

Chai is an assertion library for Node.js and the browser that can be paired with any JavaScript testing framework, for our case: Mocha. Chai provides developers with a range of assertions that can be used to write tests that are readable and expressive.

Backend Approaches used:

- Unit Testing: We are testing discrete actions(registration, login, accessing protected routes, and event creation), here each it block represents a unit test that verifies a single piece of functionality.

- API Testing: We are also targeting the HTTP endpoints with chai.request(server) to simulate client requests and test server responses.

- Positive and Negative Testing: For the registration and login endpoints, we are conducting positive tests by sending the correct data and expecting a successful response (HTTP 200 status code) and are checking how the system behaves under failure scenarios by giving 403 Forbidden response.

- Functional Testing: We are also assessing whether the system functions correctly by registering a new user, logging in a user, and checking role-based access control.

Future improvements:

- Expand Test Coverage: We aim to include tests for all the other functionalities of your application to ensure comprehensive coverage.

- Error Handling: We aim to test the application's response to various types of errors, such as database failures, network issues, and invalid request data.

- Integration Testing: We aim to integrate more tests to verify that different parts of the application work together as expected, such as the interaction between the authentication service and database.

What works?

At the time of the submission of the project, we have written tests to ensure that the user registration process works correctly. We have also written tests to validate the user login functionality. We are also performing role based access testing, which checks for specific functions to be allowed by certain roles.

Example code for testing registering process:

```
import * as chai from 'chai';
import chaiHttp from 'chai-http';
import server from '/Users/shreyasaxena/Desktop/LL/LogicLegends/backend/index.js';
const { expect } = chai;
chai.use(chaiHttp);

describe('User Registration', () => {
  it('should register a new user', (done) => {
    const newUser = {
      email: "test@example.com",
      firstName: "John",
      lastName: "Doe",
      password: "123456",
      country: "USA",
      zipCode: "12345"
    };

    chai.request(server)
      .post('/register')
      .send(newUser)
      .end((err, res) => {
        expect(res).to.have.status(200);
        expect(res.body).to.be.an('object');
        expect(res.body.message).to.equal('Account created successfully.');
        done(); // Indicates that the test is complete
      });
  });

  describe('User Login', () => {
    it('should login successfully and return a token', (done) => {
      const loginDetails = {
        email: 'user@example.com',
        password: 'password123'
      };
```

```
    chai.request(server) // Use `server` instead of `app`
      .post('/login')
      .send(loginDetails)
      .end((err, res) => {
        expect(res).to.have.status(200);
        expect(res.body).to.have.property('token');
        done();
      });
  });
});
```

Automation testing

Overview of Automation Testing Implementation:

Our automation testing setup revolves around the usage of GitHub Actions, a powerful automation tool provided by GitHub. GitHub Actions allows us to define custom workflows to automate various tasks such as building, testing, and deploying our application. Currently we have fully implemented the frontend testing and backend testing. Even though it is failing a few backend tests right now but we are working that also as some of those features are still in development.

Workflow Configuration:

We have configured a YAML file named "Automated Testing" that defines our automation testing workflow. This workflow is triggered on every push event to the repository, ensuring that tests are run automatically whenever new code changes are pushed.

```yaml
name: Automated Testing

on: [push]

jobs:
  build:
    runs-on: ubuntu-latest  # Specify the operating system you want to run the job on

    steps:
      - name: Checkout code
        uses: actions/checkout@v2  # This step checks out your repository's code

      - name: Set CI environment variable to false
        run: echo "CI=false" >> $GITHUB_ENV

      - name: Install dependencies
        run: npm install  # Install project dependencies using npm


      - name: Cypress run
        uses: cypress-io/github-action@v6
        with:
          # build: npm run build
          start: npm start

      - name: backend testing
        run: npm run test:backend
```

Steps of Automation Testing Workflow:

Checkout Code:
This step fetches the latest code from our repository so that subsequent steps can access it.

Set CI Environment Variable:
We set the CI environment variable to false to ensure that tests are not run in CI mode during the workflow execution. This is essential for certain testing scenarios where CI-specific behaviors need to be bypassed.

Install Dependencies
We install project dependencies using npm to ensure that all required packages are available for testing.

Cypress End-to-End Testing:
We utilize the Cypress GitHub Action to perform end-to-end testing of our application. This step includes starting the application using "npm start" and then running Cypress tests to ensure that critical user journeys are functioning as expected.

Backend Testing:
We execute backend testing by running a suite of tests specifically designed for testing the backend of our application. This step ensures that backend functionalities, such as database interactions, are thoroughly tested.

Summary:
During the project, we've discussed plenty of strategies we could use to make sure that we can work together smoothly as a team and that our code will be able to work harmoniously when put together, for that reason, we try to meet regularly at least twice a week to keep up with what we've done so far and explain our code to each other that way there is less chance of conflicts furthermore, to ensure the quality of the work done we constantly carry out code reviews for anything pushed into the repository and make sure it works in everyones system too. Version control was a big focus for our group. Through GitHub, this became a smooth process as managing branches and limiting our interactions with each other's code while having to work remotely simultaneously saved us so much time. Among the many things this project made us realize is that testing is key to the process of coding, it helps pinpoint a mistake that might cause the whole app to crash and on the level of personal experience, it became clear that mistakes you don't find quickly during the programming process later on become WAY harder to deal with which ends up costing exponentially more time hence why we are so meticulous with testing as shown above. Regarding what is coming to be the final project, most of the features we needed to implement are currently working correctly and have passed the testing and the website is coming together great and is almost completely functional.