September 2023

ENCOMPASS

PERFECT STRANGERS

EPI-USE LABS

value through innovation

Name	Student Number
JULIANNA VENTER	U20433752
MORGAN BENTLEY	U18103007
RONIN BROOKES	U19069686
SAMEET KESHAV	U21479373

Team: Perfect Strangers
perfectstrangers.tuks@gmail.com

COS 301
Capstone Project
University of Pretoria

Introduction

This document outlines the testing policy for the Encompass application. Testing is a critical aspect of our development process to ensure the reliability, stability, and quality of our application. This policy covers the testing strategies, tools, and procedures employed for both the backend and frontend components of the Encompass project.

Backend Testing

Testing Tools

We have adopted Jest as the primary testing framework for both backend unit and integration testing in our application.

Testing Environment

To maintain a clean and isolated testing environment, we follow these procedures:

1. Build and Teardown: We build and tear down the application for each integration test suite, ensuring a fresh start for every test.

2. Separate Test Database: We utilize a separate test database to prevent interference with our working database. This ensures that our tests do not affect the actual application data.

3. Database Reset: After each individual test within a suite, the test database is reset, and it is updated with the necessary data for the subsequent test.

4. Database Clean-up: At the conclusion of each test suite, the test database is thoroughly cleared to prepare it for the next suite.

Test Coverage

For backend testing, we ensure comprehensive coverage:

- All controllers are fully tested to guarantee their functionality.

Negative Testing

As part of our testing strategy, we incorporate negative tests where necessary. This involves testing scenarios where the application is expected to fail gracefully or handle errors appropriately.

Example Backend Tests:

Unit-test:

```
describe('getRecommendedBooks', () => {
  Run | Debug
  it('should return an array of BookDto', async () => {
    const id = '1';
    const expectedResult = [{ id: '1', title: 'Book 1' }];
    jest.spyOn(queryBus, 'execute').mockResolvedValue(expectedResult);

    const result = await controller.getRecommendedBooks(id);

    expect(result).toEqual(expectedResult);
    expect(queryBus.execute).toHaveBeenCalledWith(
      new GetRecommendedBooksQuery(id),
    );
  });
});
```

Integration-test:

```
describe('getCommunitiesByKeyword', () => {
  Run | Debug
  it('should get communities by keyword', async () => {
    // Insert multiple community DTO instances into the database
    const communityStubs = [
      communityDtoStub({ customName: "Example Community 1" }),
      communityDtoStub({ customName: "Example Community 2" }),
      communityDtoStub({ customName: "Another Community" }),
    ];
    await dbConnection.collection('community').insertMany(communityStubs);

    // Choose a keyword to search for communities
    const keyword = 'Example';

    // Fetch communities by keyword using the API endpoint
    const response = await request(app.getHttpServer()).get(`/community/get-communities-by-keyword/${keyword}`);

    expect(response.status).toBe(200);

    expect(response.body).toHaveLength(2); // Two communities match the keyword "Example"

    // Assert specific properties of the first and second community
    expect(response.body[0].name).toContain('Example');
    expect(response.body[1].name).toContain('Example');

  });
  });
```

Frontend Testing

Testing Tools

We employ both Cypress and Jest for frontend testing:

- Cypress: Used for end-to-end tests, simulating user interactions such as button controls and validating if URL changes meet expectations.
- Jest: Utilized for frontend unit testing, where we evaluate the individual functionality of each frontend feature to ensure they perform as expected.

Test Types

We use both positive and negative testing for both Cypress and Jest:

- Positive Testing: To verify that features work as expected when users interact with the application.
- Negative Testing: To ensure that features fail gracefully and produce expected errors when encountering exceptional conditions.

Example Frontend tests:

Unit tests:

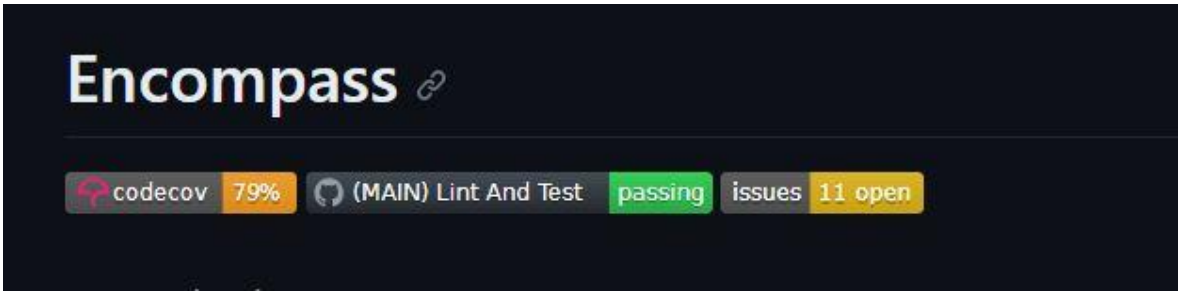
```
it('should create', () => {
  expect(component).toBeTruthy();
});

Run | Debug
it('should disable the "Sign Up" button when form is incomplete', () => {
  component.user.email = 'test@example.com';
  component.user.password = '12345678';
  component.user.username = '';
  component.user.firstName = '';
  component.user.lastName = '';
  component.checked = true;

  component.checkInput();
  fixture.detectChanges();

  const signUpButton: HTMLButtonElement = fixture.nativeElement.querySelector('.btn2');
  expect(signUpButton.disabled).toBe(true);
});
```

Cypress tests:

```
describe('Login', () => {
  beforeEach(() => {
    cy.visit('/login');
  });

  it('should log in with valid credentials', () => {
    const validAccount: AccountDto = {
      _id: '123456789abc', // Mock user ID
      email: 'valid@example.com',
      password: 'validpassword',
    };
    cy.intercept('POST', '/api/account/login', {
      statusCode: 200,
      body: validAccount,
    }).as('loginApi');

    cy.get('ion-input[name="email"]').type(validAccount.email);
    cy.get('ion-input[name="password"]').type(validAccount.password);
    cy.get('.btn1').click();

    cy.wait('@loginApi');

    cy.url().should('include', '/home');
  });
```

Test Coverage

Our overall test coverage for the application is 79%, indicating a substantial coverage of our codebase.



Continuous Integration and Deployment

- Automated Testing: Tests are automated and integrated into our GitHub pipeline. When a pull request is submitted for merging into the dev or main branch, these tests are executed to ensure that no regressions or issues are introduced.
- Code Merging: Only after the tests have passed successfully will code be merged into the respective branches.
- Continuous Deployment: Continuous Deployment is triggered whenever changes are pushed to the main branch, ensuring that our application is always up-to-date and reliable.

Conclusion

This testing policy is an integral part of the Encompass software engineering project. It ensures that our software is thoroughly tested, reliable, and ready for deployment. By following these testing practices and automation, we aim to deliver a high-quality application that meets our clients' expectations.

PERFECT STRANGERS