UNIVERSITY of SAN CARLOS

SCIENTIA • VIRTUS • DEVOTIO

DCISM

DEPARTMENT OF COMPUTER, INFORMATION SCIENCES AND MATHEMATICS

WALLETZ

SOFTWARE QUALITY ASSURANCE

IT3202N: TTH 4:30PM - 6:00PM

Submitted By:

Barrera, Carl Dave

Caballes, Ian Lemuel

Pacampara, Rolan Dave

Submitted to:

Mr. Keenan Paul O. Mendiola

May 29, 2024

I. INTRODUCTION

Every application comprises different parts and components with distinct functions integrated as a single working system. For a system to deliver its intended function, these varying parts work together, performing specific tasks and communicating with each other to provide a streamlined experience. To ensure that these processes are not hindered, software testing is needed to make sure the system can perform its expected functions as well as to protect the system from being damaged and exploited.

Unit testing tests each component or module individually in an isolated environment. This way, it ensures smaller code blocks can function properly within the broader software system instead of the whole. This also allows easier fixes for errors and bugs, as a fully functioning system sometimes suffers as a whole over a single broken component. Unit testing stands out as a fundamental practice within software quality assurance, as it is the first line of defense against software defects. Software developers need to detect and address issues as early as possible before the deployment of the system.

For our team to achieve proper quality assurance, we set out objectives to carry out consistent work processes.

- **Meet Requirements:** Knowing the requirements of the individual part and its expected function is important to figuring out what function is not intended or not part of the process.

- **Identify Defects:** Identify problems within the component or modules as early as possible to prevent these issues from propagating in the later stages, which can be difficult.
- **Documentation:** Providing documentation can help developers keep track of errors and define the expected behavior of each unit.
- **Consistent Quality**: Maintain a consistent quality standard across the system, ensuring all parts of the software meet the same level of scrutiny.
- **Prevent Error Regression:** Ensure previously fixed errors and problems don't reappear in later stages, such as with the introduction of new additions or changes to the code.
- **Refactoring Confidence**: Enable safe code refactoring by ensuring that changes do not introduce new defects, thereby maintaining code integrity.
- **Team Collaboration**: Encourage team collaboration by providing clear, automated feedback on code changes, facilitating better communication and coordination within the team.

II. OVERVIEW OF THE PROJECT

Walletz is a transaction tracking application that helps users track their savings and expenses, providing a current balance, total savings and expenses, and history of all transactions made. Walletz is not an e-wallet application. It is simply a tracker for users to track their transactions through the use of a software application instead of having to rely on written records. Walletz features key functionalities such as:

- **Account System:** Walletz requires an account to be used. Users can create an account on the Sign Up page and use an existing account to log in, which leads them to their account dashboard.
- **Cloud Database:** This application uses Atlas MongoDB for its database needs. Data is stored in the cloud instead of locally. This ensures that the data is stored in a safe and secure place.
- **Calendar and Date:** The calendar will always update to the latest day and provide date data for any transactions made during that day.
- **Transaction History:** On the Wallet page, a list of transactions is displayed to track all the transactions made by that user.
- **Transaction Dashboard:** The dashboard provides helpful insights and information about their transactions, such as the total amount of savings and expenses made, current balance, and other visualizations.

III. Testing Framework

We will be using Jest as our testing framework. Due to our application using React and NodeJS frameworks, as well as many JavaScript libraries used, Jest is a perfect testing framework to use as it is designed to be used around projects built using JavaScript. It has many functionalities that can help improve our test by also going beyond the testing scope and providing insights as to why such an event happened. Jest requires little to no configuration, allowing it to be easily used by our team. Jest also includes a snapshot testing feature that allows developers to capture and compare the outputs of a component or function and mocking capabilities that allow developers to isolate and test components individually.

Before you can start using and testing with Jest, installation must be done first. The installation directory will depend on where you want to start writing your unit tests. In this scenario, we will be installing it to test our front end.

1. Navigate to your front-end folder; this can be done by running a terminal (e.g., Command Prompt) directly inside your directory or by navigating by entering this command in your terminal.

   ```
   cd frontend
   ```

2. Install Jest by running this command through your terminal.

   ```
   npm install --save-dev jest
   ```

   If using Yarn, use this instead.

   ```
   yarn add --dev jest
   ```

3. Add a test script to your frontend's 'package.json'. This is just the minimap configuration for Jest; a separate file can be created if you opt for more configuration.

   ```
   {
   	"scripts": {
   		"test": "jest"
   	}
   }
   ```

IV. TEST FILE STRUCTURE

Organization of Test Files

To ensure clarity and maintainability, test files should be organized in a manner that mirrors the structure of the source files. This facilitates locating the corresponding tests for each component, function, or module.

Example Directory Structure

Here's an example of how the test files can be organized within the "**3202N-Walletz**" project:

```
3202N-Walletz/
│
├── frontend/
│   ├── src/
│   │   ├── components/
│   │   │   ├── __tests__/
│   │   │   │   └── Header.test.js
│   │   │   ├── Header.js
│   │   │   └── Footer.js
│   │   ├── services/
│   │   │   ├── __tests__/
│   │   │   │   └── apiService.test.js
│   │   │   └── apiService.js
│   │   └── utils/
│   │       ├── __tests__/
│   │       │   └── calculateTotal.test.js
│   │       └── calculateTotal.js
│   ├── package.json
│   └── jest.config.js
│
└── README.md
```

In this structure:

- Test files are placed in "**__tests__**" directories adjacent to the files they test.
- Each component, service, or utility has a corresponding test file with the "**.test.js**" extension.

V. WRITING UNIT TESTS

Guidelines for Writing Unit Tests

- **Isolate Units:** Tests should focus on a single "**unit**" of code (function, component, etc.) to ensure isolation.
- **Mock Dependencies:** Use mocks for external dependencies to avoid side effects.
- **Assertions:** Use assertions to check the expected outcomes.
- **Descriptive Names:** Test case names should clearly describe the scenario being tested.
- **Repeatable:** Tests should be deterministic and produce the same results regardless of the environment.

Naming Conventions

- Test files: "**ComponentName.test.js**"
- Test functions: "**shouldDoSomethingWhenCondition()**"

Example Test Cases

- **Component Test (Header)**

```
import React from 'react';
import { render, screen } from '@testing-library/react';
import Header from '../Header';

test('renders header component', () => {
    render(<Header />);
    const headerElement = screen.getByText(/Walletz/i);
    expect(headerElement).toBeInTheDocument();
});
```

- Service Test (apiService)

```
import { fetchData } from '../apiService';

test('fetches data from API', async () => {
    const data = await fetchData('endpoint');
    expect(data).toEqual({ key: 'value' });
});
```

VI. EXAMPLE TEST CASES

Controllers

- User Controller Test (“userController.test.js”)

```
const { getUser, createUser } =
require('../controllers/userController');

test('should create a new user', () => {
    const newUser = { name: 'John', email: 'john@example.com' };
    const result = createUser(newUser);
    expect(result).toEqual({ id: 1, ...newUser });
});
test('should get an existing user', () => {
    const userId = 1;
    const result = getUser(userId);
        expect(result).toEqual({ id: 1, name: 'John', email:
'john@example.com' });
});
```

Models

- **User Model Test ("userModel.test.js")**

```
const User = require('../models/userModel');

test('should create a user model instance', () => {
    const user = new User({ name: 'John', email: 'john@example.com' });
    expect(user.name).toBe('John');
    expect(user.email).toBe('john@example.com');
});
```

Routes

- **Auth Routes Test ("authRoutes.test.js")**

```
const request = require('supertest');
const app = require('../app');

test('should return 200 for login route', async () => {
    const response = await request(app).post('/login').send({ email: 'john@example.com', password: 'password' });
    expect(response.status).toBe(200);
});
```

Services

- **Auth Service Test ("authService.test.js")**

```
const { generateToken, verifyToken } = require('../services/authService');

test('should generate a valid JWT token', () => {
    const user = { id: 1, name: 'John' };
```

```
        const token = generateToken(user);
        expect(token).toBeDefined();
    });

    test('should verify a valid JWT token', () => {
        const token = 'some-valid-jwt-token';
        const result = verifyToken(token);
        expect(result).toBeTruthy();
    });
```

VII. RUNNING TESTS

Instructions on How to Run Tests

- Navigate to the frontend directory:

  ```
  cd frontend
  ```

- Install dependencies:

  ```
  npm install
  ```

- Run all tests:

  ```
  npm test
  ```

Continuous Integration (CI) Setup (if applicable)

- Integrate Jest with CI tools like Travis CI, Circle CI, or GitHub Actions to automate the running of tests on each commit.

VIII. BEST PRACTICES

Tips for Effective Unit Testing

- Write tests alongside or before writing the implementation code (Test-Driven Development).
- Ensure high test coverage, but focus on meaningful tests rather than aiming for 100% coverage.
- Regularly run tests to catch regressions early.

Common Pitfalls to Avoid

- Avoid writing overly complex tests.
- Do not test implementation details; focus on the output and behavior.
- Keep tests independent and avoid dependencies on the order of execution.

IX. CONCLUSION

Summary of the Importance of Unit Testing

Unit testing is crucial for maintaining the quality and reliability of the software. It helps catch the issue early, facilitates safe refactoring, and ensures that individual components function correctly.

Encouragement to Maintain and Update Tests Regularly

Regularly updating and maintaining tests is essential to keeping them relevant and effective, especially as the codebase evolves.

X. REFERENCES

Links to Documentation for the Testing Framework

- Jest Documentation

Additional Resources on Unit Testing Principles and Practices

- Unit Testing Best Practices
- Introduction to Test-Driven Development (TDD)