Testing Plan Using Jest

Introduction

This document outlines a comprehensive testing plan that incorporates unit, integration, and system testing methodologies, leveraging the Jest testing framework alongside automation strategies for efficient test execution. Jest, a delightful JavaScript Testing Framework with a focus on simplicity, offers features such as a zero-configuration setup, instant feedback, and fast execution of tests. Integrating Jest into our testing plan will ensure a streamlined and effective testing process, enhancing code quality and reliability.
Objectives

To ensure the software meets its design and behavioral specifications. To identify and fix defects at early stages of development. To facilitate code refactoring and ensure backward compatibility. To automate the testing process, reducing manual effort and speeding up development cycles.

Scope

- Unit Testing: Isolating and testing individual units/components for correctness.
- Integration Testing: Verifying the interactions between different units/components.
- System Testing: Testing the complete and integrated software to verify it meets requirements.

Tools and Technologies

- Jest: Primary framework for writing and executing tests.
- CI/CD Pipeline (e.g., Jenkins, GitHub Actions): For automating the testing and deployment processes.
- Mocking Libraries (e.g., jest.mock): For simulating interactions with modules or external services.

Test Strategy

1. Unit Testing
   - Objective: Test individual units/components in isolation to ensure they function correctly.
   - Approach: Use Jest to write and execute tests for each component. Utilize mocking and spies to isolate the component from its dependencies.
   - Execution: Developers are responsible for writing unit tests as they develop features. Continuous Integration (CI) pipelines will automatically run these tests on every commit to the version control system.

2. Integration Testing
   - Objective: Ensure that integrated components work together as intended.

- Approach: Identify critical integration points within the application and write tests that simulate the interaction between these components using Jest.
- Execution: Integration tests will be executed after successful completion of unit tests in the CI pipeline. They can also be run manually for specific components during development.

3. System Testing
- Objective: Validate the system as a whole against requirements.
- Approach: Use Jest in conjunction with end-to-end testing tools (e.g., Puppeteer, Selenium) to write tests that simulate real user scenarios.
- Execution: System tests will be executed as part of the deployment phase in the Continuous Deployment (CD) pipeline, ensuring the software is tested in an environment that closely mirrors production.

Automation Strategy

- Continuous Integration (CI): Integrate Jest tests into the CI pipeline to run automatically on every push to the main/version control branch, ensuring immediate feedback on the impact of changes.
- Continuous Deployment (CD): Configure the CD pipeline to run system tests automatically before deployment, ensuring that only thoroughly tested code is deployed to production.
- Scheduled Runs: Schedule regular test runs for system tests to identify issues that might be introduced by external dependencies or environment changes over time.

Best Practices

- Test Naming: Follow a consistent naming convention that clearly describes what each test does.
- Test Data Management: Use separate test data from production data, ensuring tests do not interfere with live environments.
- Version Control Integration: Integrate the testing suite with version control systems to ensure tests are versioned alongside the code they test.
- Documentation: Maintain thorough documentation for test cases, including the purpose and expected outcomes, to facilitate maintenance and understanding.

Conclusion

Implementing a structured testing plan using Jest and automation will significantly improve the quality and reliability of the software. By covering unit, integration, and system testing, we can ensure that individual components, their interactions, and the entire system function as expected. Automating these tests within CI/CD pipelines not only streamlines the development

process but also enables quick detection and resolution of issues, leading to a more efficient and error-free development lifecycle.
Here are some but not all of the tests already implemented:

Test Description	Inputs	Rule	Expected Output
Render home without crashing	None	The home component should render without throwing any errors.	Pass
Render Sign in when the user is not signed in.	None	It checks if the sign in button renders when there is no user session.	Pass
Render Sign in when the user is not signed in.	None	It checks if the sign up button renders when the user is not signed in.	Pass
Display drop down menu when profile picture is clicked.	Clicking the profile picture icon.	It checks if the drop down menu is displayed when the profile picture is clicked.	Pass
Render the login component	None	It checks if the “Sign in to access your account” renders on the page properly.	Pass
The sign up link when clicked	Clicking the “sign up” text	It checks that a link to the sign up page is present and is set up properly when clicked on the text.	Pass
Render profile information on page	None	It checks if the profile information such as first name, last name, email, phone are rendered on page.	Pass
Uploading a profile picture correctly	A image file	It checks if when a proper image file is uploaded, it's uploaded properly.	Pass
Toggle the registration key	A button	It checks the registration key visibility and can be toggled on and off.	Pass
Selecting account type	A button	It tests the functionality of an account type (renter	Pass

		or owner) and checks if the button styles change.	
Update First and Last name	First and Last name	It checks if the user updates his first and last name, the values are updated.	Pass
Password Mismatch	Mismatched password	It checks if the user inputs mismatched passwords, an error message pops up	Pass