

# **QA Automation Engineering Case Study**

## **B2B SaaS Platform – Multi-Platform Testing**

# INDEX

1.	Introduction.....	1
2.	Debugging Flaky Test Code.....	1
3.	Test Framework Design .....	3
4.	API + UI Integration Test.....	6
5.	Assumptions and Limitations.....	7
6.	What I Would Improve with More Time .....	8
7.	Final Conclusion.....	8

# 1. Introduction

This document presents my approach to solving the QA Automation Engineering case study for a multi-tenant B2B SaaS platform. The objective of this assignment is to demonstrate my understanding of real-world automation challenges, including flaky test debugging, scalable test framework design, and end-to-end validation across API, web, and mobile layers.

The platform under consideration is a project management SaaS application that supports multiple tenants, different user roles, dynamic UI behavior, and third-party integrations. Given these constraints, my focus throughout this case study is on reliability, maintainability, and scalability rather than on producing a fully runnable automation suite.

The solution is structured into three parts:

- Debugging and stabilizing flaky UI automation tests
- Designing a scalable and maintainable test automation framework
- Validating a complete business workflow using a combination of API and UI testing

Where requirements are incomplete or ambiguous, reasonable assumptions are made and clearly documented. The overall approach reflects practical QA decision-making commonly used in production environments.

## 2. Debugging Flaky Test Code

### 2.1 Overview

In this part of the assignment, an existing Playwright-based automation test for the login functionality of the application was provided. The test was reported to be *flaky*, meaning it sometimes passes and sometimes fails, especially when executed in a CI/CD pipeline.

The objective of this section is to:

- Identify the causes of flakiness in the given test code
- Explain why these issues occur more frequently in CI/CD environments
- Propose conceptual fixes to improve test stability and reliability

## 2.2 Flakiness Issues Identified

After analyzing the provided test code, the following issues were identified as potential causes of intermittent test failures:

- The test clicks the login button and immediately verifies the page URL. Since the login process is asynchronous, the dashboard page may not be fully loaded at the time the assertion is executed, leading to intermittent failures.
- The dashboard uses dynamic content loading, where elements such as welcome messages or project cards appear only after additional API calls. The test attempts to verify these elements immediately, without ensuring that they are fully rendered on the page.
- The test compares the current page URL with an exact expected URL. In real-world applications, redirects, query parameters, or delayed navigation can cause this assertion to fail even when the login is successful.
- The application may require two-factor authentication (2FA) for certain users. The test does not account for this conditional authentication flow, which can result in failures when 2FA is triggered during execution.
- The test assumes that all project data loads instantly and belongs to a specific tenant. In a multi-tenant environment, different tenants may have varying data volumes and loading times, which can affect test stability.

## 2.3 Root Causes of Flakiness in CI/CD Environments

The identified flakiness issues occur more frequently in CI/CD environments than in local execution due to several underlying factors:

- CI/CD environments generally have slower network speed and limited system resources compared to local machines, which increases page load times and delays UI rendering.
- Automated tests in CI/CD pipelines are often executed in parallel, which can lead to resource contention and timing inconsistencies across test runs.
- Tests may be executed across different browsers, operating systems, and screen resolutions in CI/CD pipelines, exposing race conditions and timing issues that may not appear in a single local setup.
- CI/CD executions do not benefit from unintentional manual delays that often occur during local debugging, making timing-related issues more visible.
- Dynamic UI elements and asynchronous backend calls are more sensitive to execution speed differences, causing assertions to run before the application reaches a stable state.

## 2.4 Proposed Fixes and Improvements

To reduce flakiness and improve test reliability, the following fixes and improvements are proposed:

- In my approach, I prioritize explicit waits and element-based validation instead of hardcoded delays, as this reduces flakiness while keeping the tests reliable in CI/CD environments.
- URL validations should be made more flexible by using URL patterns or by verifying the presence of dashboard-specific elements instead of exact URL matching.
- Tests should be designed to handle dynamic content gracefully by waiting for elements to become visible or interactable before accessing them.
- Conditional authentication flows such as two-factor authentication should be detected and handled to prevent unexpected test failures.
- Browser initialization and configuration should be centralized to ensure consistent behavior across different test environments.
- Reasonable timeout values should be configured to accommodate slower CI/CD environments without masking genuine application issues.

I chose explicit waits and API-based setup over hardcoded delays or UI-only flows because these approaches reduce flakiness while keeping tests maintainable in CI/CD environments.

## 2.5 Conclusion

By identifying the root causes of flakiness and applying the above strategies, the reliability of the automation tests can be significantly improved. Stabilizing flaky tests is critical for maintaining trust in automated test results, especially in CI/CD pipelines where automation is a key decision-making factor for deployments.

# 3. Test Framework Design

## 3.1 Overview

This part of the assignment focuses on designing a scalable and maintainable test automation framework for a multi-tenant B2B SaaS platform. The framework is intended to support web and mobile testing, multiple tenants, different user roles, API testing, and integration with CI/CD pipelines.

The emphasis in this section is on architectural design and testing strategy rather than on detailed implementation.

## 3.2 Framework Structure

The proposed test automation framework is organized in a modular and layered manner to ensure clarity, scalability, and ease of maintenance:

- The framework separates UI tests, API tests, and integration tests to ensure clear responsibility and easier debugging.
- UI tests focus on validating user interactions and visible behavior of the application across supported browsers.
- API tests validate backend services independently of the user interface, allowing faster execution and early detection of issues.
- Integration tests combine API and UI testing to validate complete business workflows from end to end.
- Page Object files are used to encapsulate UI elements and actions, reducing duplication and improving maintainability.
- Utility modules provide shared functionality such as API clients, authentication helpers, and reusable wait logic.

## 3.3 Configuration and Management Strategy

A flexible configuration approach is required to handle multiple environments, tenants, and execution platforms:

- Environment-specific configurations allow tests to run against development, staging, or production-like environments without code changes.
- Tenant-specific configurations support multiple company domains and ensure proper data isolation during test execution.
- User role configurations enable testing of different permission levels such as Admin, Manager, and Employee.
- Browser and device configurations allow tests to run across multiple browsers and mobile platforms.

These configurations can be managed using configuration files or environment variables to avoid hardcoded values in test scripts.

## 3.4 Cross-Browser and Mobile Testing Approach

I designed the framework to support cross-browser and multi-tenant testing from the beginning, as these are common sources of complexity in B2B SaaS platforms.:

- Web tests can be executed across different browsers to validate compatibility and stability.
- Mobile testing focuses on validating critical workflows rather than duplicating all desktop scenarios.
- Cloud-based platforms such as BrowserStack can be integrated to run tests on real devices and operating systems without maintaining local infrastructure.

This approach ensures broad coverage while keeping execution time and cost under control.

### **3.5 API Testing Integration**

I intentionally use API testing for test data setup and UI testing for validation, as this combination provides faster execution while still validating real user behavior:

- API tests provide fast and reliable validation of backend functionality.
- API calls can be used to create and manage test data required for UI and integration tests.
- Using APIs for test setup reduces dependency on UI flows and minimizes flakiness.

This strategy improves test stability and speeds up overall execution.

### **3.6 Missing Requirements and Clarifying Questions**

Some requirements are intentionally unspecified in the assignment, and the following questions would need clarification in a real-world scenario:

- How should test data be created, shared, and cleaned across test runs?
- Is parallel execution expected, and what level of parallelism is supported?
- What type of test reports are required for stakeholders?
- Are there usage or cost limitations when using BrowserStack?
- How should conditional authentication mechanisms such as two-factor authentication be handled for automation users?

Identifying these gaps early helps in designing a more robust and efficient automation framework.

### **3.7 Conclusion**

The proposed test framework design focuses on modularity, flexibility, and real-world scalability. By separating concerns, supporting multiple platforms, and integrating API testing, the framework is well-suited for validating complex multi-tenant B2B SaaS applications.

I intentionally avoided over-engineering the framework with advanced features such as custom retry engines or complex reporting pipelines. Given the time constraints of the assignment, the focus was placed on clarity, scalability, and correctness of design rather than exhaustive implementation.

## **4. API + UI Integration Test**

### **4.1 Overview**

This part of the assignment focuses on validating a complete business workflow by integrating API testing with UI testing. The objective is to ensure that a project created through the backend API is correctly reflected in the web and mobile user interfaces while maintaining strict tenant isolation in a multi-tenant B2B SaaS platform.

The approach reflects real-world testing practices where API and UI layers are combined to achieve both reliability and user-level validation.

### **4.2 End-to-End Test Scenario**

The integration test scenario follows a logical sequence that represents an actual user workflow:

- A new project is created using the backend API to ensure fast, reliable, and UI-independent test data creation.
- After the project is created, the web application is accessed, and the user logs in to verify that the newly created project appears correctly on the dashboard.
- The same project is then validated on a mobile platform to ensure consistent behavior across different devices.
- Finally, tenant isolation is verified by logging in as a user from a different company and confirming that the project is not visible, ensuring data security between tenants.

### **4.3 Integration Test Logic**

The logical flow of the integration test can be summarized as follows:

- Use an authenticated API request to create a project and capture the project details returned in the response.
- Perform a login operation through the web UI and navigate to the dashboard page.
- Verify that the project created via the API is displayed in the user interface.
- Execute the same validation on a mobile environment to confirm cross-platform accessibility.
- Log in with a different tenant and verify that the project is not accessible, validating proper tenant isolation.

This structured approach ensures that both backend functionality and frontend visibility are validated together.

## 4.4 Cross-Platform Validation Strategy

Cross-platform testing is an important aspect of this integration scenario and is handled with the following considerations:

- Web UI validation ensures that the project is visible across supported desktop browsers.
- Mobile validation focuses on verifying critical workflows rather than duplicating all desktop test cases.
- Cloud-based testing platforms such as BrowserStack can be used to execute tests on real mobile devices and browsers without maintaining local infrastructure.

This strategy balances test coverage, execution time, and cost efficiency.

## 4.5 Handling Edge Cases and Reliability Considerations

To ensure the reliability of the integration test, the following edge cases are taken into account:

- Network delays or temporary failures during API calls used for project creation.
- Delayed UI rendering due to dynamic content loading on the dashboard.
- Differences in layout, responsiveness, or behavior between desktop and mobile devices.
- Ensuring that test data created during execution does not leak across tenants or persist beyond the test lifecycle.

## 4.6 Conclusion

By combining API-based setup with UI-based validation, this integration test approach provides comprehensive coverage of critical business workflows. It ensures functional correctness, cross-platform consistency, and tenant-level data isolation, which are essential requirements for multi-tenant B2B SaaS applications.

## 5. Assumptions and Limitations

The following assumptions and limitations were considered while designing the testing approach described in this document:

- It is assumed that all automation tests are executed in a non-production environment with stable and isolated test data.
- Test users are assumed to have pre-generated authentication credentials and valid access tokens available for API testing.
- Two-factor authentication is assumed to be disabled for automation users to avoid interruptions in automated login flows.

- It is assumed that each tenant has isolated data and that no test data is shared across different companies.
- Mobile testing is assumed to focus only on critical user workflows rather than full regression coverage due to time and execution constraints.
- Integration with external platforms such as BrowserStack is assumed to be available, with reasonable execution limits.
- Test data cleanup and environment reset mechanisms are assumed to be handled outside the scope of this assignment.

These assumptions help define the scope of the solution while acknowledging real-world constraints commonly faced in automation projects.

## **6. What I Would Improve with More Time**

Given additional time, I would focus on improving parallel test execution to reduce overall runtime in CI/CD pipelines. I would also introduce structured reporting, such as Allure or HTML-based reports, to provide better visibility into test results for stakeholders.

Additionally, I would implement automated test data cleanup mechanisms to ensure consistent test environments and reduce dependency on manual resets. Mobile test coverage would be expanded for critical workflows, and flaky test monitoring would be introduced to continuously identify and stabilize unstable tests.

## **7. Final Conclusion**

This case study demonstrates a structured approach to solving real-world QA automation challenges for a multi-tenant B2B SaaS platform. By analyzing flaky test behavior, proposing stability improvements, designing a scalable test automation framework, and validating an end-to-end business workflow using both API and UI testing, the solution addresses key quality assurance concerns.

The emphasis throughout this assignment is placed on reliability, maintainability, and clear testing strategy rather than on full implementation. This approach reflects practical industry practices where time constraints, system complexity, and incomplete requirements are common.

Overall, the proposed solution highlights the importance of thoughtful test design, effective use of automation tools, and clear documentation when building and maintaining automation frameworks for complex applications.