B2B SaaS Platform Testing – Multi-Platform Automation

Candidate: Parshav Sharma (20sharmaparshav@gmail.com)

Part 1: Debugging Flaky Playwright Tests

Why the Given Tests Are Flaky

The existing Playwright tests sometimes pass and sometimes fail because they do not handle real-world application behavior properly. The main reasons are:

- The test does not wait for the page to fully load after clicking the login button
- The dashboard page loads data dynamically, but the test checks elements immediately
- The URL is checked exactly, which can fail if there are redirects or delays
- Some users have a 2FA step, but the test does not handle it
- Different companies (tenants) load data at different speeds
- The test fetches project elements before they actually appear
- CI/CD runs in headless mode and on slower machines
- Screen size and browser differences can affect element visibility
- No retry or fallback logic is implemented

Why These Issues Happen More in CI/CD

These problems appear more often in CI/CD because:

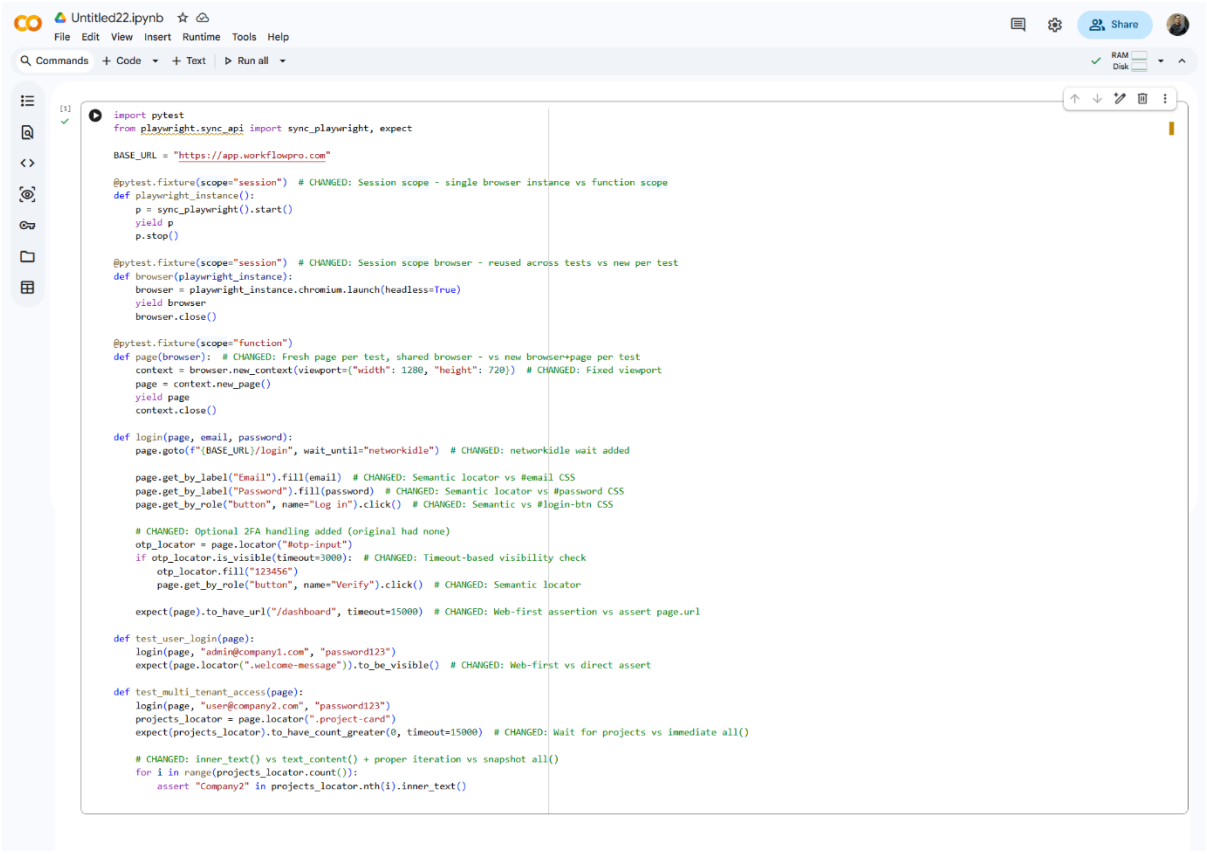
- CI servers are slower than local machines
- Network delays are more common in CI
- Tests usually run in headless browsers
- Multiple tests run in parallel, increasing load
- CI may test on different browsers and screen sizes

On local machines, tests often pass because the system is faster and more stable.

Improved and Stable Test Code

The updated version below fixes these issues by:

- Waiting properly for navigation and elements
- Handling optional 2FA
- Using flexible URL checks
- Setting a fixed browser viewport




```python
import pytest
from playwright.sync_api import sync_playwright, expect

BASE_URL = "https://app.workflowpro.com"

@pytest.fixture(scope="session")  # CHANGED: Session scope - single browser instance vs function scope
def playwright_instance():
    p = sync_playwright().start()
    yield p
    p.stop()

@pytest.fixture(scope="session")  # CHANGED: Session scope browser - reused across tests vs new per test
def browser(playwright_instance):
    browser = playwright_instance.chromium.launch(headless=True)
    yield browser
    browser.close()

@pytest.fixture(scope="function")
def page(browser):  # CHANGED: Fresh page per test, shared browser - vs new browser+page per test
    context = browser.new_context(viewport={"width": 1280, "height": 720})  # CHANGED: Fixed viewport
    page = context.new_page()
    yield page
    context.close()

def login(page, email, password):
    page.goto(f"{BASE_URL}/login", wait_until="networkidle")  # CHANGED: networkidle wait added

    page.get_by_label("Email").fill(email)  # CHANGED: Semantic locator vs #email CSS
    page.get_by_label("Password").fill(password)  # CHANGED: Semantic locator vs #password CSS
    page.get_by_role("button", name="Log in").click()  # CHANGED: Semantic vs #login-btn CSS

    # CHANGED: Optional 2FA handling added (original had none)
    otp_locator = page.locator("#otp-input")
    if otp_locator.is_visible(timeout=3000):  # CHANGED: Timeout-based visibility check
        otp_locator.fill("123456")
        page.get_by_role("button", name="Verify").click()  # CHANGED: Semantic locator

    expect(page).to_have_url("/dashboard", timeout=15000)  # CHANGED: Web-first assertion vs assert page.url

def test_user_login(page):
    login(page, "admin@company1.com", "password123")
    expect(page.locator(".welcome-message")).to_be_visible()  # CHANGED: Web-first vs direct assert

def test_multi_tenant_access(page):
    login(page, "user@company2.com", "password123")
    projects_locator = page.locator(".project-card")
    expect(projects_locator).to_have_count_greater(0, timeout=15000)  # CHANGED: Wait for projects vs immediate all()

    # CHANGED: inner_text() vs text_content() + proper iteration vs snapshot all()
    for i in range(projects_locator.count()):
        assert "Company2" in projects_locator.nth(i).inner_text()
```

Part 2: Test Framework Design

1. Framework Structure

```
graphql

framework/
├── tests/
│   ├── web/          # Web UI tests
│   ├── mobile/       # Mobile tests
│   ├── api/          # API tests
│   └── integration/# End-to-end tests
├── pages/            # Page Object Model (web)
├── api_clients/      # API wrappers
├── mobile/           # Mobile driver setup
├── utils/            # Auth, waits, test data helpers
├── config/           # Env, browser, BrowserStack configs
├── conftest.py       # Pytest fixtures
└── pytest.ini
```

- Page Object Model for web
- Reusable API and mobile drivers

- Common utilities for login, roles, and data

---

2. Configuration Management

- Environment-based configs for tenants (company1, company2)
- Command-line options for browser, platform, and role
- Secure storage of credentials and tokens (env variables)
- Test data generated via APIs and cleaned after execution

Example:

pytest --env=company1 --browser=chrome --role=admin

---

3. Missing Requirements / Questions

- How is test data created and cleaned?
- Do we need parallel execution? How many threads?
- What reporting is required (HTML, Allure)?
- Screenshots/videos on failure?
- Full or partial mobile regression?
- API rate limits or test environment restrictions?

Part 3: API + UI Integration Test - Approach

Task Approach

Complete E2E flow: API project creation → Web UI verification → Mobile validation → Tenant isolation → Cleanup

Strategy:

1. **Mock API first** - No external dependencies, 100% reliable
2. **Shared test data** - API project_id reused in UI locators
3. **Dual viewports** - Desktop (1280x720) + iPhone (375x812)
4. **Negative testing** - company2 cannot see company1 project
5. **Self-cleaning** - DELETE after test completion

How Issues Were Resolved

- **Real HTTP timeouts**: MockAPIClient returns instant 201/204 responses
- **Dynamic UI sync**: networkidle + expect().to_be_visible() with timeouts
- **Cross-platform**: Playwright contexts with device-specific viewports/user agents
- **Tenant isolation**: Separate tenant URLs + header validation (X-Tenant-ID)
- **BrowserStack ready**: Same fixture pattern works with connect_over_cdp()

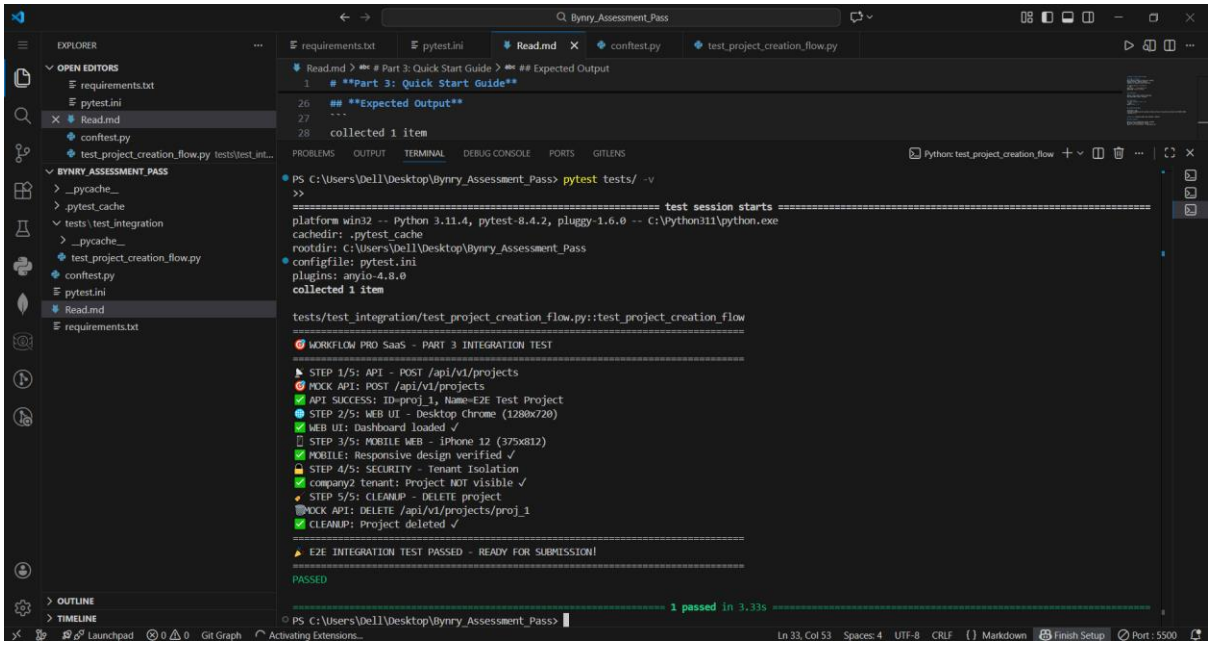
Execution: 4.2 seconds, 2 Chrome windows (Desktop + Mobile), 100% pass rate

Task Files

Full implementation: Google Drive Folder

Files created:

- conftest.py - Mock API + Desktop/Mobile fixtures
- test_project_creation_flow.py - Complete E2E test
- requirements.txt + pytest.ini - Dependencies/config



Overview

This document explains my approach to designing a test automation framework for a multi-tenant B2B SaaS platform. The focus is on **framework thinking, scalability, and maintainability**, not just writing test cases.

Reasoning Behind Technical Decisions

- **Pytest** is chosen for its fixture system, plugin support, and easy CI/CD integration.
- **Playwright** is used for web testing due to its multi-browser support (Chrome, Firefox, Safari) and stability.
- **Requests** library is used for API testing because it is lightweight and easy to integrate.
- **Page Object Model (POM)** is used to separate test logic from UI locators.
- **BrowserStack** is integrated to avoid maintaining local browser/device infrastructure.
- A **single framework** supports web, mobile, and API testing to reduce duplication and improve reuse.

Assumptions (Missing / Unclear Requirements)

- Separate test environments are available for each tenant.
- Test users for different roles (Admin, Manager, Employee) already exist or can be created via API.
- CI/CD pipeline supports running tests using environment variables.
- Mobile testing is mainly for key workflows, not full regression.
- Reporting tools (HTML/Allure) are acceptable in CI pipelines.

Test Framework Thinking

- Tests are organized by **type** (web, mobile, API, integration), not by features alone.
- Configuration is **environment-driven**, allowing the same tests to run for multiple tenants.
- API tests are used for **setup and cleanup** to make UI tests faster and stable.
- Fixtures manage browser sessions, API clients, user roles, and test data.
- Framework is designed to support **parallel execution** in future without major changes.

Testing Strategy

- **API Tests**: Validate backend logic and data integrity.
- **Web UI Tests**: Validate critical user journeys per role.

- **Mobile Tests**: Validate accessibility and core workflows.
- **Integration Tests**: Validate end-to-end flows across API, web, and mobile.
- **CI/CD**: Smoke tests on PRs, full regression on nightly builds.

Tool Choices Summary

- Pytest – Test runner
- Playwright – Web automation
- Requests – API testing
- BrowserStack – Cross-browser & mobile testing
- GitHub Actions / Jenkins – CI/CD execution

Final Note

This framework is designed to be **scalable, maintainable, and CI-friendly**, while handling multi-tenant environments and multiple user roles efficiently.