Automation Engineering Internship Case Study Solution

WorkFlow Pro - B2B SaaS Platform Testing Assessment
Name: Atharva Gaikwad
Date: July 24, 2025

Executive Summary

- Flaky Playwright tests are primarily caused by race conditions, dynamic content loading, and inconsistent environment configurations
- A scalable test framework requires Page Object Model implementation, environment-driven configuration, and proper CI/CD integration
- Comprehensive integration testing must validate multi-tenant isolation while supporting cross-platform execution

Part 1: Debugging Flaky Test Code

1.1 Identified Flakiness Issues

Issue	Root Cause	Impact
URL assertion fires before redirect	No navigation wait mechanism	Intermittent failures in CI
.welcome-message selector occasionally absent	Element loads via XHR after dashboard mount	Race condition on slow networks
Hard-coded credentials bypass 2FA	2FA randomly enforced for privileged users	Environment-specific behavior
Project card iteration without wait	No retry between card rendering	Timing-dependent failures
Inconsistent browser/viewport settings	No explicit configuration	Layout-dependent test breaks
No error recovery mechanisms	Hard failures without retry logic	Poor test reliability

1.2 Root Cause Analysis

Why CI/CD ≠ Local Environment:

1. Resource Constraints: Shared CI executors throttle CPU/network, extending XHR completion times beyond Playwright's implicit waits
2. Browser Matrix Differences: CI may execute on WebKit/Safari while local uses Chromium, causing selector/CSS inconsistencies
3. Viewport Variations: Mobile emulation in CI causes responsive layout shifts that hide elements
4. Security Configuration: 2FA toggles often enabled only in non-development environments

1.3 Technical Solutions

Enhanced Configuration & Fixtures:

Python

```python
# conftest.py - Shared test fixtures
import os, pytest
from playwright.sync_api import sync_playwright, expect

@pytest.fixture(scope="session")
def browser():
    with sync_playwright() as p:
        headless = os.getenv("HEADLESS", "true") == "true"
        browser = p.chromium.launch(headless=headless)
        yield browser
        browser.close()

@pytest.fixture
def page(browser):
    context = browser.new_context(
        viewport={"width": 1280, "height": 800},
        locale="en-US"
    )
    page = context.new_page()
    yield page
    context.close()
```

Improved Test Implementation:

python

```python
# test_login.py - Reliable login tests
import pytest, os
from playwright.sync_api import expect

BASE_URL = os.getenv("BASE_URL", "https://app.workflowpro.com")
```

```python
def do_login(page, email, password):
    page.goto(f"{BASE_URL}/login", wait_until="domcontentloaded")
    page.locator("#email").fill(email)
    page.locator("#password").fill(password)
    page.click("#login-btn")

    # Handle optional 2FA
    if page.locator("text=Verify Code").is_visible(timeout=2000):
        code = os.getenv("TOTP_CODE", "000000")
        page.locator("#totp").fill(code)
        page.click("button:text('Verify')")

@pytest.mark.flaky(reruns=2)
def test_user_login(page):
    do_login(page, "admin@company1.com", "password123")
    expect(page).to_have_url(f"{BASE_URL}/dashboard")
    expect(page.locator(".welcome-message")).to_be_visible()

def test_multi_tenant_access(page):
    do_login(page, "user@company2.com", "password123")
    cards = page.locator(".project-card")
    expect(cards).to_have_count_greater_than(0)

    cards.filter(has_text="Company2").first.wait_for(state="visible")
    for card in cards.all():
        assert "Company2" in card.text_content()
```

Technical Reasoning:

- wait_until="domcontentloaded" ensures page structure is ready before interaction
- Playwright's "web-first" assertions provide automatic retry logic
- Conditional 2FA handling makes tests environment-agnostic
- Fixed viewport prevents responsive layout issues
- pytest-rerun marks legitimate flakes while trending metrics

Part 2: Test Framework Design

2.1 Framework Architecture

Directory Structure:

Text

```
tests/
├── ui/
│   ├── pages/
│   │   ├── login_page.py
│   │   ├── dashboard_page.py
│   │   └── projects_page.py
│   ├── dashboard/
│   │   └── test_dashboard_functionality.py
│   └── auth/
│   └── test_authentication.py
├── api/
│   ├── clients/
│   │   └── project_client.py
│   └── test_projects_api.py
├── mobile/
│   └── test_mobile_workflows.py
├── data/
│   ├── tenants.yml
│   ├── users.json
│   └── test_projects.json
├── configs/
│   ├── local.toml
│   ├── staging.toml
│   └── production.toml
├── utils/
│   ├── auth_helpers.py
│   ├── data_factory.py
│   └── reporting.py
├── conftest.py
├── playwright.config.ts
└── pytest.ini
```

2.2 Configuration Management Strategy

Hierarchical Configuration System:

1. Base configuration in configs/*.toml files
2. Environment-specific overrides via CLI flags (--env=staging)
3. Runtime secrets via environment variables

4. CI/CD pipeline integration through Docker/Jenkins

Sample Configuration:
Text

```
# configs/staging.toml
[environment]
base_url = "https://staging.workflowpro.com"
api_url = "https://api-staging.workflowpro.com"
timeout = 30000

[browsers]
desktop = ["chromium@latest", "firefox@latest-1", "webkit"]
mobile = ["ios@16.4", "android@13"]

[browserstack]
username = "${BS_USERNAME}"
access_key = "${BS_ACCESS_KEY}"
project = "WorkFlow Pro Staging"
```

2.3 Core Framework Components
Page Object Implementation:
Python

```
# ui/pages/base_page.py
class BasePage:
    def __init__(self, page):
        self.page = page
        self.timeout = 10000

    def wait_for_load(self):
        self.page.wait_for_load_state("domcontentloaded")

    def get_tenant_context(self):
        return self.page.locator("[data-tenant-id]").get_attribute("data-tenant-
id")
```

API Client Abstraction:
Python

```
# api/clients/base_client.py
import requests
from tenacity import retry, stop_after_attempt, wait_exponential
```

```python
class BaseAPIClient:
    def __init__(self, base_url, tenant_id, auth_token):
        self.base_url = base_url
        self.headers = {
            "Authorization": f"Bearer {auth_token}",
            "X-Tenant-ID": tenant_id,
            "Content-Type": "application/json"
        }

    @retry(stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1,
min=4, max=10))
    def post(self, endpoint, data):
        response = requests.post(f"{self.base_url}{endpoint}",
                                 json=data, headers=self.headers, timeout=30)
        response.raise_for_status()
        return response.json()
```

2.4 Missing Requirements Analysis

Critical Questions for Stakeholders:

1. Test Data Management
    a. Data retention policy for test tenants?
    b. Database seeding/cleanup strategy?
    c. PII handling in test environments?
2. Reporting & Monitoring
    a. Required KPIs (pass rate, duration, flake rate)?
    b. Integration with existing dashboards?
    c. Alert mechanisms for test failures?
3. Parallel Execution
    a. BrowserStack concurrency budget?
    b. Test isolation requirements?
    c. Resource allocation per test suite?
4. CI/CD Integration
    a. Pipeline trigger conditions?
    b. Test environment provisioning?
    c. Artifact retention policies?
5. Security & Compliance
    a. Access control for test environments?
    b. Audit logging requirements?

c. GDPR/SOC2 considerations?

Part 3: API + UI Integration Test

3.1 Comprehensive Integration Test Implementation

Python

```python
# test_project_creation_flow.py
import os, pytest, requests
from playwright.sync_api import expect
from utils.auth_helpers import get_api_token
from utils.data_factory import cleanup_test_data

# Test configuration
TENANT_CONFIG = {
    "company1": {"base": "https://app.workflowpro.com", "id": "company1"},
    "company2": {"base": "https://app.workflowpro.com", "id": "company2"}
}
API_BASE = "https://api.workflowpro.com/api/v1"

@pytest.fixture(scope="session")
def api_auth():
    """Retrieve API authentication token"""
    return {"Authorization": f"Bearer {get_api_token()}"}

@pytest.fixture
def test_project_data():
    """Generate unique test project data"""
    import uuid
    project_id = str(uuid.uuid4())
    return {
        "name": f"Test Project {project_id[:8]}",
        "description": "Integration test project",
        "team_members": ["test-user-123"]
    }

@pytest.fixture(autouse=True)
def cleanup_after_test():
    """Ensure test data cleanup"""
    created_projects = []
    yield created_projects
    cleanup_test_data(created_projects)
```

```python
def create_project_via_api(auth_headers, tenant_id, project_data):
    """Create project using API endpoint"""
    headers = {**auth_headers, "X-Tenant-ID": tenant_id}
    response = requests.post(
        f"{API_BASE}/projects",
        headers=headers,
        json=project_data,
        timeout=30
    )
    response.raise_for_status()
    return response.json()["id"]

@pytest.mark.parametrize("tenant", ["company1", "company2"])
@pytest.mark.parametrize("browser_name", ["chromium", "webkit"])
def test_project_creation_flow(page, browser_name, tenant, api_auth,
                               test_project_data, cleanup_after_test):
    """
    Complete integration test: API → Web UI → Mobile → Security Validation
    """
    tenant_config = TENANT_CONFIG[tenant]

    # Step 1: Create project via API
    project_id = create_project_via_api(
        api_auth,
        tenant_config["id"],
        test_project_data
    )
    cleanup_after_test.append(project_id)

    # Step 2: Verify project in Web UI
    page.goto(f'{tenant_config["base"]}/login')
    page.fill("#email", f"user@{tenant}.com")
    page.fill("#password", "password123")
    page.click("#login-btn")

    # Navigate to projects page
    page.locator(".sidebar-nav >> text=Projects").click()
    expect(page.locator(f"[data-project-id='{project_id}']")).to_be_visible()

    # Verify project details
    project_card = page.locator(f"[data-project-id='{project_id}']")
    expect(project_card.locator(".project-
name")).to_have_text(test_project_data["name"])

    # Step 3: Mobile verification (BrowserStack integration)
```

```
    mobile_context = page.context.browser.new_context(
        viewport={"width": 390, "height": 844},
        is_mobile=True,
        user_agent="Mozilla/5.0 (iPhone; CPU iPhone OS 16_0 like Mac OS X)"
    )

    mobile_page = mobile_context.new_page()
    mobile_page.goto(f'{tenant_config["base"]}/m/projects')

    # Mobile-specific assertions

expect(mobile_page.locator(f"text={test_project_data['name']}")).to_be_visible(
)
    mobile_context.close()

    # Step 4: Security validation - tenant isolation
    page.context.clear_cookies()
    page.goto(f'{tenant_config["base"]}/logout')

    # Login as different tenant user
    other_tenant = "company1" if tenant == "company2" else "company2"
    page.goto(f'{TENANT_CONFIG[other_tenant]["base"]}/login')
    page.fill("#email", f"user@{other_tenant}.com")
    page.fill("#password", "password123")
    page.click("#login-btn")

    # Attempt to access project from different tenant
    page.goto(f'{TENANT_CONFIG[other_tenant]["base"]}/projects/{project_id}')
    expect(page.locator("text=403")).to_be_visible()

@pytest.mark.slow
def test_project_creation_with_network_resilience(page, api_auth,
test_project_data):
    """Test with network failures and recovery"""
    # Simulate network delays
    page.route("**/api/v1/projects", lambda route: route.fulfill(status=200,
body='{"id": "test-123"}'))

    # Test with artificial latency
    page.context.set_default_timeout(60000)  # Extended timeout

    project_id = create_project_via_api(
        api_auth,
        TENANT_CONFIG["company1"]["id"],
        test_project_data
```

```
    )

    # Verify UI handles slow API responses gracefully
    page.goto("https://app.workflowpro.com/projects")
    expect(page.locator(".loading-spinner")).to_be_visible()
    expect(page.locator(f"[data-project-
id='{project_id}']")).to_be_visible(timeout=30000)
```

3.2 Cross-Platform Strategy

BrowserStack Integration:

Javascript

```
// playwright.config.ts
export default {
 projects: [
   {
     name: 'Desktop Chrome',
     use: { ...devices['Desktop Chrome'] }
   },
   {
     name: 'BrowserStack iOS',
     use: {
       channel: 'browserstack',
       ...devices['iPhone 13'],
       connectOptions: {
         wsEndpoint: `wss://cap-
playwright:${process.env.BROWSERSTACK_ACCESS_KEY}@hub.browserstack.com/playwrig
ht`
       }
     }
   }
 ]
};
```

3.3 Test Data Management Strategy

Isolation Approach:

- Unique test data generation using UUIDs
- Tenant-specific data seeding

- Automated cleanup after test completion
- Database transaction rollback for integration tests

Data Factory Pattern:

```
python
# utils/data_factory.py
class TestDataFactory:
    @staticmethod
    def create_unique_project(tenant_id):
        return {
            "name": f"Test-{tenant_id}-{uuid.uuid4().hex[:8]}",
            "tenant_id": tenant_id,
            "created_by": "automation-test"
        }
```

Technical Assumptions & Considerations

4.1 Authentication Assumptions

- API token authentication is available for test environments
- Test user accounts exist for each tenant with appropriate permissions
- 2FA can be disabled or bypassed in test environments

4.2 Environment Assumptions

- Staging environment mirrors production configuration
- Test data isolation is enforced at the database level
- BrowserStack account provides sufficient concurrent sessions

4.3 Performance Considerations

- API response times under 2 seconds for project creation
- UI loading times under 5 seconds for dashboard elements
- Mobile responsive design supports viewport widths 320px-1920px

4.4 Security Assumptions

- Test environments use HTTPS exclusively
- Cross-tenant data access returns 403/404 responses
- Session management includes proper token expiration