Case Study Response: QA Automation Engineer

Candidate: YASH SHARAD PATIL

EMAIL: yashpatil1395@gmail.com

College : Pune Institute Of Computer Technology

Date: January 6, 2026

Part 1: Debugging and Technical Analysis

Optimized Test Implementation

The following code addresses stability issues by implementing robust waiting strategies and error handling to ensure consistent performance across local and CI environments.

CODE:

```
import pytest
import logging
from playwright.sync_api import Page, expect, sync_playwright

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def test_user_login():
    """Validates authentication flow with synchronization safeguards."""
    with sync_playwright() as p:
        browser = p.chromium.launch(headless=True, slow_mo=100)
        context = browser.new_context(viewport={"width": 1920, "height": 1080})
        page = context.new_page()

        try:
            logger.info("Opening login portal...")
            page.goto("https://app.workflowpro.com/login",
                      wait_until="networkidle",
                      timeout=30000)

            # Wait for and fill email
            email_field = page.locator("#email")
            email_field.wait_for(state="visible", timeout=10000)
            email_field.fill("admin@company1.com")

            # Wait for and fill password
            password_field = page.locator("#password")
    password_field.wait_for(state="visible", timeout=5000)
    password_field.fill("password123")

    logger.info("Submitting credentials...")
    # Click and wait for navigation
    login_button = page.locator("#login-btn")
    login_button.click()

    # Wait for navigation to dashboard
    page.wait_for_url("https://app.workflowpro.com/dashboard",
                    timeout=15000)

    # Verify welcome message
    welcome_msg = page.locator(".welcome-message")
    welcome_msg.wait_for(state="visible", timeout=10000)
    expect(welcome_msg).to_be_visible()

    logger.info("✅ Authentication successful")

except Exception as e:
    # Create directory if it doesn't exist
    import os
    os.makedirs("failure_reports", exist_ok=True)
    page.screenshot(path="failure_reports/login_error.png", full_page=True)
    logger.error(f"❌ Test failed: {str(e)}")
    raise
finally:
    browser.close()
```

Identification of Instability Factors

Primary Root Causes

The original test failures stem from a mismatch between script execution speed and application response times, particularly in resource-constrained environments like Jenkins or GitHub Actions.

Variable	Local Testing	CI/CD Pipeline	Result
Network Latency	Low/Stable	Variable/High	Timed-out elements
Hardware Performance	High CPU/RAM	Shared Containers	Slower DOM rendering
Execution Mode	Headed (Visual)	Headless	Timing/Rendering shifts

Implemented Solutions and Justification

- **Network Idle State:** Instead of simple URL navigation, the script now waits for the network to be quiet. This ensures that background API calls (common in React/Angular apps) finish before we attempt to interact with the UI.
- **Web-First Assertions:** Moving from standard Python `assert` to Playwright's `expect()` API allows for "auto-retrying." The test will now wait up to 5 seconds for a condition to be met rather than failing instantly.
- **Contextual Synchronization:** Using `expect_navigation` alongside a click action prevents a common race condition where the script tries to find elements on the next page before the browser has even begun the transition.
- **Diagnostic Artifacts:** Added automated screenshots and detailed logging. In a CI environment, seeing the state of the UI at the exact second of failure is critical for differentiating between a code bug and a system slowdown.

Long-term Quality Assurance Strategy

To prevent the re-emergence of "flaky" tests, the following standards should be adopted:

1. **Strict Locator Policy:** Use resilient selectors (like data-test-ids) rather than brittle CSS paths or XPaths that change with UI updates.
2. **Modular Architecture:** Shift toward a Page Object Model (POM) to centralize selector logic and reduce maintenance overhead.

3. **Environment Parity:** Run local tests in headless mode and limit bandwidth occasionally to simulate the constraints of the CI pipeline.
4. **Automatic Retries:** Configure the test runner to retry failed tests once to account for transient network blips, while still flagging them for investigation if they fail consistently.

—---

Part 2: Test Framework Strategy and Design

Framework Hierarchy and Organization

The following structure is engineered to meet the demands of a multi-tenant B2B SaaS environment. It emphasizes modularity and separation of concerns to ensure that web, mobile, and API tests remain maintainable as the application scales.

Directory Layout

Below is the file structure given :

```
automation_root/
│
├──→ Settings/
│      ├──→ env-profiles.json         # defn for Dev, staging & Prodn.
│      ├──→ execution-caps.json       # local & Browserstack. driver config
│      ├──→ tenant-registry.json      # multi-tenant routing & metadata
│      └──→ identity-roles.json       # credn mapped to specific permissio
│
├──→ Components/
│      ├──→ base-interaction.py       # Global UI methods & Synch
│      ├──→ login-view.py
│      ├──→ project-dashboard.py
│      └──→ handheld/                 # optimized views for mobile native
│              ├──→ handheld-base.py
│              └──→ handheld-login.py
│
├──→ api-engine/
│      ├──→ request_manager.py        # Session handling & std. headers
│      └──→ routes/
│              ├──→ project-endpoints.py
│              └──→ account-endpoints.py
│
├──→ Suites/
│      ├──→ web-desktop/
│      ├──→ mobile-app/
│      └──→ backend-api/
│
├──→ support/
│      ├──→ reporting-bridge.py       # Connector Allure/TestRail
│      ├──→ synthetic data.py         # Dynamic data generation
│ ~~de~~ └──→ cloud-tunnel.py         # Browserstack Connectivity.
│
├──→ dependency_injection/
│      ├──→ ui_hooks.py               # playwright browser
│      └──→ api-hooks.py              # Authentication & session setup.
│
├──→ pytest.ini
│
└──→ conftest.py
```

Core Logic Implementation

Page Object Foundation (base_interaction.py)

```
class BaseInteraction:
    def __init__(self, page, settings):
        self.page = page
        self.settings = settings
    def element_sync(self, locator, time_limit=12000):
        """Standardized wait to ensure UI readiness."""
        self.page.wait_for_selector(locator, state="visible", timeout=time_limit)
    def log_visual_state(self, event_name):
        """Captures screenshots for audit trails on failure."""
        self.page.screenshot(path=f"logs/visuals/{event_name}.png")
    def route_to_tenant(self, company_key):
        """Handles dynamic subdomain routing for SaaS isolation."""
        self.page.goto(f"https://{company_key}.workflowpro.com")
```

API Service Client (request_manager.py)

```
class RequestManager:
    def __init__(self, base_url, auth_token=None):
        self.base_url = base_url
        self.http = requests.Session()
        if auth_token:
            self.http.headers.update({"Authorization": f"Bearer {auth_token}"})
    def inject_tenant_context(self, tenant_uuid):
        """Ensures API calls are scoped to the correct company data."""

        self.http.headers.update({"X-Tenant-Context": tenant_uuid}

    def post_resource(self, path, body):

        """Generic POST method with error handling wrapper."""

        return self.http.post(f"{self.base_url}/{path}", json=body)
```

Discovery and Requirement Clarification

To finalize this architecture, the following operational questions need to be addressed:

Data Integrity and Lifecycle

- **Setup Strategy:** Should we rely on real-time API calls for test prerequisite data, or should we utilize database seeding for faster execution?
- **Resource Cleanup:** Is there a preference for "soft deletes" during teardown, or should we target a dedicated ephemeral environment that resets daily?
- **Collision Prevention:** How will we manage data state when multiple parallel threads target the same tenant simultaneously?

Observability and Reporting

- **Stakeholder Needs:** Do we require executive-level dashboards (like Allure) in addition to developer-centric console logs?
- **Evidence Management:** How long should we retain failure artifacts like videos and traces in our CI/CD storage?
- **Failure Alerts:** Should regressions trigger immediate notifications via Slack or Microsoft Teams?

Execution Environment

- **BrowserStack Utilization:** What is our current parallel thread limit? This determines how we optimize our test distribution.
- **Pipeline Integration:** Should the suite run as a "gatekeeper" on every PR, or as a scheduled regression at specific intervals?

Strategic Design Justification

This framework is purpose-built for the complexities of B2B SaaS:

1. **Isolation Verification**: By externalizing tenant and role data, we can verify that users in "Company A" cannot access data in "Company B" simply by altering configurations.

2. **Unified Platform Logic**: A single codebase manages Desktop, Mobile Web, and API validation, reducing the maintenance burden and preventing logic fragmentation.
3. **Decoupled Architecture**: Separation of configuration from execution allows for rapid environment switching (e.g., pointing local tests to a Staging URL) without modifying source code.
4. **Resilience by Design**: Using Base Classes for UI interaction ensures that all wait logic and synchronization are handled globally, significantly reducing flakiness.

Part 3: API + UI Integration Strategy

This test employs a **"Hybrid Validation"** approach. We use the API for high-speed state setup (creating the project) and then use the UI to verify the end-user experience across platforms. This maximizes coverage while minimizing the execution time spent on slow UI setups.

Implementation: Project Lifecycle & Isolation Test

```
import pytest
from playwright.sync_api import expect

def test_project_creation_cross_platform_flow(api_client, dashboard_page, mobile_page, security_context):
    """
    Validates end-to-end project lifecycle: API Creation -> Web UI -> Mobile UI -> Security Isolation.
    """
    # Unique data generation to prevent collision in parallel runs
    project_name = f"QA_Project_{generate_uuid()}"
    tenant_a = security_context.get_tenant("CompanyA")
    tenant_b = security_context.get_tenant("CompanyB")

    # --- 1. API LAYER: State Injection ---
    # We create the project via API to ensure a clean, predictable state
    payload = {
        "name": project_name,
        "description": "Integration Test Project",
        "team_members": ["admin@companya.com"]
    }

    response = api_client.post("/api/v1/projects", payload, tenant_id=tenant_a.id)
    assert response.status_code == 201, f"Failed to create project via API: {response.text}"
    project_id = response.json().get("id")

    try:
        # --- 2. WEB UI: Desktop Validation ---
        dashboard_page.route_to_tenant(tenant_a.slug)
        dashboard_page.login(tenant_a.admin_user)

        # Handling dynamic loading: verify_project_exists handles polling internally
        dashboard_page.verify_project_exists(project_name)
        expect(dashboard_page.project_card(project_name)).to_be_visible()

        # --- 3. MOBILE: BrowserStack Validation ---
        # Using a separate driver context configured for mobile viewport/User-Agent
        mobile_page.route_to_tenant(tenant_a.slug)
        mobile_page.login(tenant_a.admin_user)

        # Verify responsiveness: check if the project card stacks correctly or uses mobile UI
        mobile_page.toggle_hamburger_menu()
        expect(mobile_page.project_list_item(project_name)).to_be_visible()

        # --- 4. SECURITY: Tenant Isolation check ---
        # Login as a different company to ensure the project is NOT leaked
        dashboard_page.logout()
        dashboard_page.route_to_tenant(tenant_b.slug)
        dashboard_page.login(tenant_b.admin_user)

        # Explicitly verify the project is absent for Tenant B
        dashboard_page.search_project(project_name)
        expect(dashboard_page.empty_state_message).to_be_visible()
        expect(dashboard_page.project_card(project_name)).not_to_be_attached()

    finally:
        # --- CLEANUP: Resource Disposal ---
        # Always attempt to delete the test project regardless of test outcome
        api_client.delete(f"/api/v1/projects/{project_id}", tenant_id=tenant_a.id)
```

Strategy and Technical Decisions

1. Data Management & Clean-up

- **API for Setup/Teardown:** Using the API for POST and DELETE ensures tests are atomic. We don't rely on the UI to "clean up" (which might fail if the UI is broken).
- **Dynamic Naming:** Using UUIDs in project names allows this test to run in **parallel threads** without multiple instances of the test trying to manage the same project.

2. Handling Edge Cases

- **Network Resilience:** The api_client includes a retry decorator for 503 or 504 errors common in CI/CD.
- **Slow Loading/Hydration:**Instead of time.sleep(), we use Playwright's Auto-waiting. For example, expect(locator).to_be_visible() polls for the element to be both present and "stable" (not moving due to CSS transitions).

- **Mobile Responsiveness:** On BrowserStack, we verify that interaction triggers (like the Hamburger menu) are used, as standard sidebars often disappear on mobile screen widths.

3. Cross-Platform Validation

- **BrowserStack Capabilities:** The mobile_page fixture utilizes BrowserStack's real device cloud (e.g., iPhone 15, Pixel 8). We pass the browserstack.isMobile = true capability to ensure the app serves the mobile-optimized version.

4. Security (Tenant Isolation)

- **The "Negative" Test:** Isolation isn't proven just by seeing data in Company A; it's proven by its absence in Company B. The test explicitly logs into a second tenant to confirm a "404 Not Found" or empty search result state.

Assumptions Made

1. **Authentication:** Assumed a security_context utility provides valid tokens and company-specific credentials.
2. **API Access:** Assumed the API is reachable from the same network where Playwright is running (or via a BrowserStack Local tunnel).
3. **Mobile Setup:** Assumed the application is responsive (Web) rather than a Native App, allowing Playwright to drive the mobile browser via viewport emulation or real device connectivity.