Senior QC Automation Assessment – End-to-End Submission

Repository: https://github.com/YoussefAzabawi/flairstech-qc-automation
Author: Youssef Mohamed El-Azabawi - Senior QC Engineer Automation & Manual
Date: February 4, 2026

Part 1 – Test Automation Strategy

1.1 90-Day Roadmap (30/60/90)

Days 0–30 – Foundation & Quick Wins - Understand the product domain, critical user journeys, and release cadence. - Define the **test pyramid** across Web UI, API, and Mobile, and agree target coverage per layer. - Set up baseline frameworks: - Playwright TypeScript mono-repo for **Web UI + API**. - **Appium + Mocha** skeleton for Android using the Sauce Labs Sample App. - Implement smoke tests: - Web: happy-path **User Registration + Login**. - API: **Register + Login** endpoints (positive and basic negative cases). - Mobile: 1 happy-path workflow (login + simple navigation). - Integrate Web + API smoke suite into **GitHub Actions** (on PR and on push to main). - Define initial **coding standards**, review checklist, and branching strategy (tests required with features, POM usage, locator rules).

Days 31–60 – Expansion & Stabilization - Expand **API coverage** around critical microservices and contracts (auth, profile, catalog, orders). - Extend **Web POMs and flows** for high-value journeys (account, cart, checkout). - Establish **test data strategy**: - Environment-specific config via JSON (`config/env.dev.json`, `config/env.qa.json`) plus `TEST_ENV`. - JSON/CSV-driven test data (`data/users.json`, `data/apiUsers.csv`). - Stable test accounts and/or repeatable data seeding scripts. - Turn on and tune **parallel execution** and test sharding in CI. - Add **reporting stack**: - Playwright HTML report with screenshots, traces, and videos on failure (already configured). - CI artifacts to make reports easily accessible. - Start formal **flaky-test management**: - Tagging (e.g. `@flaky`), quarantine suite, and root-cause tracking.

Days 61–90 – Optimization & Governance - Refine **suite selection** by pipeline type: - PR: fast smoke + core API tests (few minutes). - Nightly: broader Web + API regression. - Pre-release: curated full regression (selected UI E2Es + full critical API + key mobile flows). - Increase framework resilience: - Stronger waits and retry policies; network mocking where appropriate. - Component-level POMs and reusable utilities to minimize duplication. - Expand **mobile coverage** only for high-value journeys; consider optional scheduled CI mobile jobs. - Formalize **governance**: - Definition of Done includes necessary automation. - Coding and test-design standards, review templates, and ownership per area/service. - Introduce **metrics & ROI** dashboards based on CI and defect data; review regularly with engineering/product.

1.2 Prioritization Across UI, API, Mobile, Microservices, Backend

- **Highest priority – API & microservices**
  - Focus first on core business capabilities (auth, identity, catalog, payments, orders).
  - These tests are fastest, most stable, and best suited as CI gates.
- **Next – Web UI flows on critical paths**
  - Registration, login, checkout, and key dashboards.
  - UI tests are kept thin; business rules are validated at API level.
- **Mobile**
  - Prioritize high-traffic or unique mobile journeys (login, simple forms, critical navigation).
  - Maintain a small, stable set of high-value tests due to device/emulator cost.
- **Backend validations**
  - Prefer API-level state checks; use read-only DB checks or test hooks only when necessary.

1.3 What to Automate vs Not Automate, and When

Automate: - Repeatable regression flows on critical journeys and high-risk functionality. - Microservice contract tests and cross-service integrations. - Cheap, automatable non-functional checks (e.g. basic performance smoke, security headers).

Avoid or Defer: - Very rarely used flows and rapidly changing UIs. - Purely visual aspects better suited for exploratory/manual testing. - Large, brittle end-to-end chains; prefer **layered tests** (API + smaller UI checks).

When: - Apply **shift-left**: define automation approach during story refinement. - Implement feature-level automation within the same sprint, or at most one sprint behind. - Add tests only when they are valuable, maintainable, and fit the agreed test pyramid.

1.4 Embedding Automation into CI/CD (GitHub Actions)

- **PR pipeline:**
  - Lint/unit tests (if present).
  - **Web + API smoke suite** using Playwright, running in parallel.
  - Fail the PR on red; attach Playwright HTML report as CI artifact.
- **Nightly pipeline:**
  - Broader Web + API regression suite (tag-based selection).
  - Optional mobile suite (Appium) on emulator or device farm.

- **Pre-release pipeline:**
  - Full curated regression: key UI E2Es + full critical API + key mobile paths.
- **Gating:**
  - Merges blocked when **smoke/core suites** fail.
  - Extended suites can be non-blocking but visible and monitored.

1.5 Managing Flaky Tests & Enforcing Reliability

- Treat flakiness as a **defect**, not "noise".
- Use tags (`@flaky`) and a **quarantine suite**; flaky tests are isolated and prioritized for repair.
- Track flaky failure rate over time as a first-class metric.
- Technical practices:
  - Use Playwright's **auto-waiting** and robust locators (data-test attributes).
  - Avoid dependency on unstable external systems; prefer controlled test environments, mocks, and data seeding.
  - Ensure tests are **data-isolated and idempotent** for safe parallel execution.

Part 2 – Framework Architecture & Tooling

2.1 Frameworks I Have Built (Short Description)

In my previous roles, I've built several automation frameworks from scratch, and I've learned what works and what doesn't. The frameworks that lasted were the ones with clear separation of concerns and minimal coupling.

My typical architecture includes: - **Test layer:** spec files separated by domain and layer (Web, API, Mobile) so it's easy to find and maintain tests. - **Abstraction layer:** Page Objects / Screen Objects, API clients, and reusable UI components—this is where most of the maintenance happens, so keeping it clean matters. - **Core layer:** shared utilities (config, waits, assertions, logging) and test data handlers that everyone can reuse. - **Integration layer:** runners (Playwright/Mocha), reporters, and CI/CD configuration.

Patterns I've found most effective: - Page Object Model (POM) for UI tests—it's simple and everyone understands it. - Centralized configuration (JSON + env vars) so switching environments is trivial. - Reusable test data builders and fixture utilities to avoid copy-paste code.

2.2 Proposed Architecture (Implemented in This Repo)

Layers: - **Web UI (Playwright TS):** - POMs in `web/pages`. - Utilities (config loader, logger, test data helper) in `web/utils`. - Specs in `web/tests`.

- **API Testing (Playwright API module):** - Shared `ApiClient` wrapper in `api/client`. - JSON schemas for validation in `api/schemas`. - Specs in `api/tests`. - **Mobile (Appium + Mocha):** - Android capabilities in `mobile/config/capabilities.json`. - Screen Objects (POM) in `mobile/pageobjects`. - Specs in `mobile/test`. - **Core shared resources:** - Environment configs in `config/`. - JSON/CSV test data in `data/`. - CI workflow in `.github/workflows/ci.yml`.

Key characteristics: - Separation of concerns between: - Test intent (specs). - Implementation details (POMs/API client). - Cross-cutting concerns (config, data, logging). - Reuse of config and test data across Web + API + Mobile where appropriate.

2.3 Tooling Choices

- **UI Automation:** Playwright (TypeScript) – modern, fast, auto-waiting, parallel-friendly, strong debugging tools (traces, screenshots, videos).
- **API Automation:** Playwright's `APIRequestContext` – same runner/config/fixtures as UI tests, no extra runner needed.
- **Mobile Automation:** Appium + Mocha – standard for Android/iOS, integrates smoothly with JavaScript ecosystem.
- **Reporting & Logging:**
  - Playwright **HTML reporter** for Web + API tests (screenshots, traces, videos on failure).
  - Console-based logger helper for Web to show scoped messages.
  - Mocha spec reporter for Mobile (extendable to Mochawesome or Allure if required).
- **Parallel/Distributed Execution:**
  - Playwright `workers` and sharding (CLI) for fast Web + API runs.
  - Mobile can be distributed across devices/emulators or device farm providers.
- **Test Data Management:**
  - Environment JSON (`env.dev.json`, `env.qa.json`) + `TEST_ENV` variable.
  - JSON/CSV-driven data (`users.json` for Web/API, `apiUsers.csv` for login scenarios).
  - Helpers for unique data (e.g. unique emails) to support parallel runs.

2.4 Example Components (From the Implementation)

Reusable utility helper – test data loader / email builder (TypeScript) - `web/utils/testData.ts`: - Loads user fixture from `data/users.json`. - Builds unique e-mails using a timestamp to avoid collisions.

Page Object / UI abstraction - `web/pages/HomePage.ts` and `web/pages/SignupLoginPage.ts`: - Expose high-level actions (`goto`, `clickSignupLogin`, `signupNewUser`, `fillAccountDetails`, `login`, `assertLoggedIn`). - Hide all locators and DOM interaction details.

Custom wrapper – API Client - `api/client/ApiClient.ts`: - Wraps Playwright's `APIRequestContext` for `createAccount` and `verifyLogin`. - Centralizes base URL, basic status validation, and message schema expectations.

Part 3 – Metrics, Governance & ROI

3.1 Metrics (Ranked by Importance)

1. **Build/CI stability:** % of green builds on main branches.
2. **Flaky test rate:** % of failures caused by non-code/environment issues.
3. **Mean Time to Detect (MTTD)** and **Mean Time to Repair (MTTR)** defects.
4. **Suite duration:** runtime per pipeline type (PR, nightly, full regression).
5. **Automation coverage:**
    - % of critical user journeys covered by automated tests.
    - % of critical APIs/microservices covered by automated tests.
6. **Defect escape rate:** number of production defects vs total found in earlier stages.
7. **Test reliability:** pass rate of the "stable" suites over time.
8. **Team engagement:** % of PRs that include tests, review comments about automation, adherence to standards.

3.2 Calculating & Communicating Automation ROI

Inputs: - Manual execution time for a given regression suite (hours per run). - Frequency of runs (per sprint/release). - Automated execution time and ongoing maintenance cost.

Approximate formula: - (ROI = \dfrac{(ManualTime - AutomatedTime) * RunsPerPeriod - MaintenanceCost}{MaintenanceCost})

Communication: - Show **hours saved per sprint/release**, and approximate cost saving or extra capacity. - Show reduced **feedback time** (e.g. several days → minutes) and lower defect escape rate. - Use CI statistics and bug tracker reports to produce simple graphs or tables for stakeholders.

3.3 Strategy Evolution with Product & Architecture

- **New features/microservices:**
    - Start by adding **API contract tests**, then add UI/mobile tests for user-visible impact.
    - Keep UI tests minimal and focused on cross-service journeys.
- **Architectural changes (e.g. migrations, re-platforming):**
    - Add characterization tests at API level to capture current behavior before change.

  - Use feature flags and env configs to compare old vs new services.
- **Sprint flow changes:**
  - Move towards **continuous testing**: small, fast suites on each PR; richer regressions nightly.
  - Ensure each story's Definition of Done includes appropriate automation impact and implementation.

Part 4 – Scenario-Based Deep Technical Challenge

Scenario: 300+ Playwright/Appium UI tests and 200+ API tests already in CI. Pipelines take 90+ minutes, tests fail for non-code reasons, flaky tests are skipped, and merge queues are blocked.

4.1 Architectural & Environmental Improvements

- **Environment reliability:**
  - Introduce consistent **test data seeding/reset** so each run starts from a known baseline.
  - Make APIs used by tests idempotent and deterministic (where possible).
  - Reduce dependence on unstable external systems; mock or stub third parties where practical.
- **Test suite organization:**
  - Split tests into tiers:
    * Tier 0: smoke – very fast, always on PR.
    * Tier 1: core regression – nightly and before release.
    * Tier 2: extended/experimental – scheduled or on demand.
  - Use tags or naming conventions to control which suites run in which pipelines.
- **Infrastructure for speed:**
  - Increase **Playwright workers** and use test sharding across CI agents.
  - Cache Node modules and Playwright browsers to avoid re-downloading.
  - For mobile, reuse Appium sessions per file when safe, or use device farms for parallelism.

4.2 Framework Refactors for Stability & Runtime

- **Stabilize POMs and locators:**
  - Use data-test attributes and robust CSS selectors; avoid brittle XPath or text-only locators.
  - Standardize on Playwright's `expect` and auto-waiting patterns.
- **Shorter, more focused tests:**

  - Replace long "journey of everything" flows with **layered coverage**: API checks + smaller UI checks.
  - Extract reusable flows (login, navigation, common actions) into helpers or POM methods.
- **Runtime optimization:**
  - Use fixtures / `beforeAll` for expensive setup shared across tests.
  - Reuse authenticated state (Playwright storage state) instead of logging in in every test.
  - Run headless in CI and tune timeouts down where appropriate.

4.3 Test Data, Isolation, Seeding, Environment Strategy

- **Data isolation:**
  - Use unique identifiers (timestamps, UUIDs) to prevent data collisions in parallel runs.
  - Prefer tests that create and clean up their own data.
- **Seeding strategy:**
  - Central test data seeding step (script or CI stage) to create a known baseline before suites run.
  - Clear rollback or cleanup strategy between runs.
- **Environment strategy:**
  - Dedicated **automation environments** separate from manual QA and staging.
  - Minimal shared mutable state; rely on APIs for safe state inspection and cleanup.

4.4 Governance, Code Review, Coaching

Governance & standards: I've found that automation code reviews need to be stricter than application code reviews, because bad tests can slow down the whole team. I introduce a **test review checklist** for every automation PR: - Locator robustness and proper waits (no brittle XPath or text-only selectors). - Data isolation and idempotency (can this test run in parallel safely?). - Assertions clarity and correct test scope (is this testing the right thing?).

I enforce test tagging (`@smoke`, `@regression`, `@flaky`) and naming conventions—it makes suite selection trivial. I also require justification for adding UI E2E tests where API/component tests would suffice; this keeps the suite fast and maintainable.

Team coaching: I run short sessions every 2-3 weeks on Playwright/Appium best practices and common pitfalls I've seen. I pair with engineers on complex scenarios and on refactoring flaky tests—there's no better way to learn than hands-on. I document standards and examples in a `CONTRIBUTING.md` so new team members can onboard quickly.

Part 5 – Practical Hands-On Implementation (Repo Overview)

5.1 Web UI – Playwright TypeScript

- **Site:** `https://www.automationexercise.com`
- **Tech:** Playwright + TypeScript with POM.
- **Key elements:**
  - POMs:
    - `HomePage` – navigation and access to Signup/Login.
    - `SignupLoginPage` – registration form, account details, login, logged-in assertions.
  - Config-driven data:
    - `data/users.json` for user profile and address fields.
    - Utility `buildUniqueEmail()` to generate unique emails per run.
  - Fixtures and settings:
    - `playwright.config.ts`:
      - `retries: process.env.CI ? 2 : 0`
      - `screenshot: 'only-on-failure'`
      - `trace` and `video: 'retain-on-failure'`
      - Parallel `workers` on CI.
  - Flows automated:
    - Full **User Registration** flow using only UI.
    - **Login** flow for a user pre-created via API (UI/API cross verification).

5.2 API Testing – Playwright API Module

- **Base:** `https://automationexercise.com/api`
- **Endpoints covered:**
  - `createAccount` – Register.
  - `verifyLogin` – Login (positive + negative).
- **Implementation details:**
  - Shared `ApiClient` wrapper around `APIRequestContext` with `createAccount` and `verifyLogin`.
  - Environment-based configuration with `env.dev.json`, `env.qa.json` and utility `getEnvConfig()`.
  - JSON schema validation using Ajv:
    - `loginSuccessSchema` / `loginErrorSchema`.
  - CSV-driven test data:
    - `data/apiUsers.csv`: scenarios for valid login, missing email, invalid credentials.
  - Tests:
    - Positive register and login.
    - Negative cases (missing required fields, invalid credentials).
    - Assertion of:
      - HTTP behavior.

        · Payload structure via schemas.
        · `responseCode` and `message` fields for success and errors.

5.3 Mobile – Appium + Mocha

- **Sample App:** Sauce Labs Sample App (Android).
- **Structure:**
  - `mobile/config/capabilities.json` – Appium capabilities for Android emulator.
  - `mobile/pageobjects/` – Screen Objects (POM):
    - `LoginScreen` – username/password/login.
    - `HomeScreen` – asserts that the post-login view is displayed.
  - `mobile/test/login.flow.spec.js` – workflow:
    - Start Appium session.
    - Login with `standard_user` / `secret_sauce`.
    - Assert that the home/cart screen is displayed and activity is valid.
- **Assertions:**
  - Uses meaningful, stable UI locators and checks (visibility, activity name).

5.4 CI/CD – GitHub Actions

- Workflow file: `.github/workflows/ci.yml`.
- Triggers: on `push` and `pull_request` to main branches.
- Steps:
  - Checkout code.
  - Setup Node.js with npm caching.
  - Install dependencies (`npm ci`).
  - Install Playwright browsers (`npx playwright install --with-deps`).
  - Run combined Web + API tests (`npm test`).
  - Upload Playwright HTML report as CI artifact.

5.5 How to Run (Summary)

- **Install dependencies (root):**
  - `npm install`
- **Run Web UI tests:**
  - `npx playwright test web/tests`
- **Run API tests:**
  - `npx playwright test api/tests`
- **Run Web + API together (CI entry point):**
  - `npm test`
- **Run Mobile tests (after configuring capabilities and starting Appium):**
  - `cd mobile`
  - `npm install`

– `npm run test:mobile`

Final Notes

The framework is complete and ready for review. All Web + API tests are passing, the CI/CD pipeline is configured, and the mobile framework follows the same patterns (targeting Android as specified in the assignment; can be extended to iOS with XCUITest when a Mac environment is available).

Repository: https://github.com/YoussefAzabawi/flairstech-qc-automation

This implementation reflects how I approach automation in real production environments: prioritizing maintainability, speed, and team adoption over complex abstractions. The framework is designed to scale and evolve with the product.