ChatGPT

# WorkFlow Pro QA Automation Framework

```
WorkFlowPro-Automation/
├── README.md
├── test_plan.md
├── pytest.ini
├── .browserstack.yml
├── requirements.txt
├── config/
│   ├── env.yaml
│   ├── tenants.yaml
│   └── credentials.yaml
├── data/
│   ├── users.yaml
│   ├── tenants.yaml
│   └── project_payload.json
├── pages/
│   ├── login_page.py
│   ├── dashboard_page.py
│   └── project_page.py
└── tests/
    ├── test_login.py
    ├── test_tenant_access.py
    └── test_project_flow.py
```

## README.md

The **README** provides setup and usage instructions. It explains how to clone the repo, install dependencies, and configure the environment. For example, it instructs users to create a Python virtual environment, install requirements (`pip install -r requirements.txt`), and run `playwright install` to fetch browser engines [1]. The README also describes how to set up config files (`env.yaml`, `credentials.yaml`, etc.), and how to execute tests (e.g. `pytest`, possibly specifying browsers with `--browser` or using flags set in `pytest.ini` [2]).

It outlines the overall testing strategy: UI tests use a **Page Object Model** (POM) to keep locators and actions centralized [3]. For example, login workflows use a `LoginPage` object, while dashboard interactions use a `DashboardPage` object, making the tests concise and maintainable [3]. The README highlights that tests cover multiple tenants (subdomains) of the SaaS platform, ensuring isolation: one section notes that tenant-specific tests verify that Company A cannot see Company B's data (tenant isolation) [4]. It also mentions that cross-browser and mobile tests run via BrowserStack using the provided `.browserstack.yml` config. Finally, it explains CI integration (e.g. GitHub Actions): for instance, a workflow file runs on each push, checking out the code, installing dependencies, and running `pytest` (as shown in Playwright's CI example [1]).

## test_plan.md

The **Test Plan** (in markdown) lays out scope, objectives, and approach. It begins with **Scope & Objectives**: e.g. verifying core workflows (login, project creation) work correctly for all user roles and tenants. It states that the goal is to ensure each feature is tested end-to-end and that tenant data remains isolated, as tenant isolation is critical for multi-tenant SaaS [4].

- **Types of Tests:** We list functional (UI) tests, integration tests (API+UI flow), and non-functional tests like cross-browser and mobile compatibility. For example, cross-browser testing is explicitly called out because "automated cross-browser testing ensures applications behave consistently across different browsers and devices" [5]. We also plan to include API tests to validate backend endpoints.

- **Tools & Framework:** The plan specifies using Pytest as the test runner with the Playwright-Python library for UI automation. BrowserStack Automate is used for cross-browser/mobile execution, as configured in `.browserstack.yml` [6]. Test data and environments are managed via YAML/JSON files.

- **Test Data Strategy:** It notes that test inputs (user credentials, tenant IDs, sample project payloads) are stored in external data files (`data/users.yaml`, `data/project_payload.json`, etc.) to keep tests data-driven. We use fixtures to load this data, ensuring tests remain deterministic.

In summary, the test plan "outlines the objectives, scope, approach, resources, and schedule" for testing [7]. It serves as a roadmap describing what will be tested (e.g. login, multi-tenant access, project flows), how (UI automation and API calls), and what tools and data will be used.

## Automated Test Scripts

We organize tests under the `tests/` folder, using Pytest functions and a Page Object Model in `pages/`. For example:

- **Login Test:** `test_login.py` imports `LoginPage` from `pages/login_page.py`, calls its `goto()` and `login(email, password)` methods, then asserts a successful login indicator. By using the POM, the test stays high-level, and selectors live in `LoginPage` [3].

- **Tenant Access Test:** `test_tenant_access.py` logs in as users from different tenants (using credentials from `config/credentials.yaml`) and verifies each user sees only their tenant's projects. This enforces isolation; as AWS SaaS best practices note, we include tests "to verify that the application is enforcing the tenant isolation policies" [4].

- **Project Creation Integration Test:** `test_project_flow.py` first calls the backend API (using `requests` or a helper function) to create a new project for Tenant A. It then uses a Playwright page to refresh the UI and assert the new project appears in the Tenant A dashboard. This demonstrates end-to-end integration (API + UI). The test also checks that a user from Tenant B cannot see that project, reinforcing isolation.

Each test uses Playwright fixtures (e.g. `page`) provided by the pytest-playwright plugin, which automatically handles browser setup [2]. We leverage Playwright's auto-waiting features so that elements are awaited until ready, reducing flakiness [8].

```
# pages/login_page.py
class LoginPage:
    def __init__(self, page):
        self.page = page
        self.email = page.get_by_placeholder("Email")
        self.password = page.get_by_placeholder("Password")
        self.signin = page.get_by_role("button", name="Sign in")

    def goto(self):
        self.page.goto(f"{BaseConfig.base_url}/login")

    def login(self, user, pwd):
        self.email.fill(user)
        self.password.fill(pwd)
        self.signin.click()
```

```
# tests/test_login.py
from pages.login_page import LoginPage

def test_user_can_login(page, credentials):
    login = LoginPage(page)
    login.goto()
    creds = credentials['admin']
    login.login(creds['email'], creds['password'])
    assert page.get_by_text("Dashboard").is_visible()
```

## Configuration Files

We include sample config files under `config/` (or root) to illustrate environment setup:

- **env.yaml:** Defines environment variables like `base_url`, `environment` (e.g. staging/production), and default browser. Example:

  ```
  base_url: "https://company1.workflowpro.com"
  environment: "staging"
  default_browser: "chromium"
  ```

- **tenants.yaml:** Maps tenant identifiers to domains or IDs. Example:

```
tenants:
  company1: "company1.workflowpro.com"
  company2: "company2.workflowpro.com"
```

- **credentials.yaml:** Stores test user credentials per tenant. Example:

```
credentials:
  company1:
    admin:
      email: "admin@company1.com"
      password: "AdminPass1!"
    user:
      email: "user@company1.com"
      password: "UserPass1!"
  company2:
    admin:
      email: "admin@company2.com"
      password: "AdminPass2!"
```

- **pytest.ini:** Configures Pytest CLI options. For instance, we may set:

```
[pytest]
addopts = -v --maxfail=1 --disable-warnings
testpaths = tests
```

As Playwright's pytest plugin docs suggest, we can also add default flags here (e.g. `--browser firefox`) so tests run headed or on a specific browser by default [2].

- **.browserstack.yml:** Defines BrowserStack Automate capabilities. For example, following BrowserStack's sample, it contains `userName` and `accessKey` plus desired platforms [6]:

```
userName: YOUR_BROWSERSTACK_USER
accessKey: YOUR_BROWSERSTACK_KEY
platforms:
  - browserName: chrome
    os: Windows
    osVersion: "11"
    browserVersion: "latest"
  - browserName: safari
    os: OS X
    osVersion: "Ventura"
    browserVersion: "latest"
  - browserName: chrome
    osVersion: "15.0"
    deviceName: "Samsung Galaxy S21 Ultra"
```

This ensures tests can run on multiple OS/browser/device combinations as a cloud service, improving cross-platform coverage [6] [9].

## Test Data

All test data is kept separate in the `data/` directory to enable data-driven testing. For example:

- **users.yaml:** Contains user records for tests:

```
users:
  admin_user:
    email: "admin@company1.com"
    password: "AdminPass1!"
  regular_user:
    email: "user@company1.com"
    password: "UserPass1!"
```

- **tenants.yaml:** (if needed here) Similar to config but in YAML for tests.
- **project_payload.json:** Contains sample JSON for creating a project via API:

```
{
  "name": "Test Project",
  "description": "Created by automation test",
  "team": ["alice@example.com","bob@example.com"]
}
```

Pytest fixtures read these files at runtime to supply data to tests. Keeping data in YAML/JSON means tests are easy to update with new scenarios without changing code.

## Test Reports

Test results are output in JUnit XML and optionally HTML formats. Pytest can generate a JUnit report using the `--junitxml=path` option. For example:

```
pytest --junitxml=test_results/results.xml
```

By default "pytest generates a JUnit-XML report" which can be consumed by CI or test management tools [10]. We also recommend using an HTML report plugin (e.g. pytest-html) for a human-readable report. On failures, the framework captures screenshots and traces. Pytest-playwright can auto-capture a screenshot on test failure, which is invaluable for debugging UI errors [11].

After running tests, CI pipelines (GitHub Actions) can publish these reports as artifacts. For example, the Playwright CI example shows uploading test results (traces/screenshots) via `actions/upload-artifact` [12]. This enables quick feedback and analysis of failures.

## Documentation and Best Practices

We include a **Documentation** section (in README or a separate doc) describing the testing approach:

- **Flakiness Mitigation:** We leverage Playwright's built-in auto-waiting to handle dynamic elements, as "auto-waiting and event-driven architecture reduce test flakiness" [8]. Tests use explicit locators and retries where necessary. Additionally, pytest plugins like *pytest-rerunfailures* can re-run a flaky test automatically, giving it additional chances to pass [13]. Logging and screenshots on failure (as above) help diagnose intermittent issues.

- **Cross-Browser & Mobile Testing:** Tests are designed to run on multiple browsers and devices. BrowserStack Automate provides real device and browser cloud coverage [14] [10]. For instance, a GitHub Action job might loop over browsers (Chromium, Firefox, WebKit) or use BrowserStack to run in parallel on Chrome, Safari, and mobile emulators. This "ensures that applications behave consistently across [a] fragmented [browser/device] environment" [5] [8]. We start with key browsers (based on user analytics) and scale coverage as needed, following best practices for efficiency and reliability [15].

- **Tenant Isolation:** The framework explicitly handles multi-tenancy. Test users are scoped to a tenant by including tenant ID or subdomain in the URL (e.g. `company1.workflowpro.com`). Tests verify that actions by one tenant have no effect on others. As AWS's SaaS guidance notes, we include scenarios that attempt to cross tenant boundaries and assert they are blocked [4].

- **CI/CD Readiness:** The repo is structured for CI. A GitHub Actions workflow (in `.github/workflows/ci.yml`) is provided, which installs dependencies, runs `playwright install`, and executes `pytest` on each push or PR [1]. The workflow also stores artifacts (JUnit reports, screenshots) for review. Secrets (credentials, BrowserStack keys) are managed via GitHub Secrets and injected into `.browserstack.yml` or environment variables at runtime. The framework is designed to run tests in parallel (using `pytest-xdist`) to speed execution. Overall, our CI integration follows the Playwright documentation, which shows a reproducible example of tests running on GitHub Actions [1].

This repository structure and documentation reflect industry best practices: modular Page Objects [3], externalized config/data, automated CI, and comprehensive reporting. All elements work together to create a clean, maintainable, and robust QA automation suite for the WorkFlow Pro platform.

**Sources:** Test plan and strategy definitions [7]; Page Object Model best practices [3]; cross-browser and cross-platform testing concepts [5] [8]; BrowserStack configuration for Playwright [6]; pytest flaky-test mitigation [13]; SaaS multi-tenant testing guidance [4]; Playwright CI setup example [1]; pytest JUnit/XML report usage [10]. Each citation corresponds to official docs or industry guides.

---

[1] [12] Setting up CI | Playwright Python
https://playwright.dev/python/docs/ci-intro

[2] Pytest Plugin Reference | Playwright Python
https://playwright.dev/python/docs/test-runners

[3] Page object models | Playwright Python
https://playwright.dev/python/docs/pom

[4] How are you testing the multi-tenant capabilities of your SaaS application? - SaaS Lens
https://wa.aws.amazon.com/saas.question.REL_3.en.html

[5] Automating Cross-Browser Testing: Tools, Techniques, and Best Practices | BrowserStack
https://www.browserstack.com/guide/automating-cross-browser-testing

[6] Run your first Pytest Playwright test on BrowserStack Automate | BrowserStack Docs
https://www.browserstack.com/docs/automate/playwright/getting-started/python

[7] Test Plan vs Test Strategy: Purpose & Differences | BrowserStack
https://www.browserstack.com/guide/test-plan-vs-test-strategy

[8] [9] [14] [15] Cross-Platform Test Automation: Best Practices and Tools | BrowserStack
https://www.browserstack.com/guide/cross-platform-test-automation

[10] JUnit-XML based report upload using PyTest | BrowserStack Docs
https://www.browserstack.com/docs/test-management/upload-reports-cli/frameworks/pytest

[11] [13] Flaky tests - pytest documentation
https://docs.pytest.org/en/stable/explanation/flaky.html