Playwright + Pytest Automation Framework

Complete Theory Guide (Start to End Flow)

1. Project Overview

This project is a Python-based test automation framework built using Playwright and Pytest. The goal of the framework is to provide scalable, maintainable, and reusable UI automation. Instead of writing raw scripts, the project follows structured automation design principles like Page Object Model (POM), reusable fixtures, and clear test separation.

2. Overall Architecture

The framework separates responsibilities into different layers: 1) Tests Layer – contains actual test cases. 2) Pages (POM) Layer – contains page logic and UI interactions. 3) Base Page – reusable browser actions like open URL. 4) Fixtures/Configuration – manages browser setup and teardown. 5) Reports and Logs – execution tracking. This separation makes debugging and maintenance easier.

3. How Test Creation Works

Tests are written using Pytest. Each test imports required page objects and fixtures. The test describes WHAT to validate, not HOW to interact with UI. Example flow: - Pytest starts execution. - Fixtures initialize browser and page. - Test calls POM methods. - Assertions validate expected results.

4. Page Object Model (POM) Concept

POM is a design pattern where each web page is represented as a class. The class contains: - Locators - Page-specific actions - Helper functions Advantages: - Avoid duplicated selectors. - Easier maintenance. - Improved readability.

5. Base Page Responsibility

Base page acts as a parent class. Common browser actions like opening URLs are implemented here. Other page classes inherit from base page to reuse functionality.

6. Pytest Role in Framework

Pytest is the test runner. Responsibilities: - Discover tests automatically. - Execute tests. - Provide fixtures. - Generate reports. - Handle assertions. Pytest controls execution flow from start to end.

7. Fixtures Concept

Fixtures provide reusable setup and teardown logic. Example responsibilities: - Launch browser. - Create new page context. - Provide page object to test. - Close browser after execution. Fixtures reduce repetitive code.

8. Test Execution Flow (Start to End)

1) User runs pytest command. 2) Pytest scans project for tests. 3) Fixtures initialize browser and Playwright session. 4) Test requests page object. 5) Page object uses BasePage methods. 6) Browser performs actions. 7) Assertions validate results. 8) Reports generated after completion.

9. Interaction Between Files

Test file -> calls Page Object methods. Page Object -> uses BasePage utilities. Fixtures -> create and inject browser/page instance. Pytest -> manages execution lifecycle. Each component has a clear responsibility.

10. Logging and Reporting

Logging helps debugging failures. Reports provide summary of passed/failed tests. These tools help track execution history.

11. Best Practices Used

- Separation of concerns. - Reusable page methods. - Clear assertions. - Centralized browser setup. - Modular project structure.

12. Topics Covered So Far

- Playwright basics - Pytest framework - Page Object Model - Fixtures usage - Browser automation flow - Test structuring - Debugging failing tests - GitHub project setup - Automation best practices

13. Suggested Next Topics

- Advanced locators and waits - Parallel execution - CI/CD integration - Data-driven testing - API + UI combined testing - Performance analysis - Advanced reporting tools