Playwright WordPress E-commerce Website Automation Test

Author

Manan Ahmed Broti
https://ahmedmanan.com
https://github.com/AhmedManan
https://www.linkedin.com/in/ahmedmanan

December 24, 2025
version 1.0

Contents

1 Introduction

This documentation details the architecture, implementation, and deployment of a robust End-to-End automation framework designed to ensure the quality and reliability of some WP plugins, among them one plugin which uses data sourced from Google Sheets. The core purpose of this document is to provide a comprehensive guide to the established testing environment. The framework utilizes Python as the primary language, with Playwright for high-speed, reliable browser automation, and Pytest as the powerful test runner. The structure follows strictly to the Page Object Model (POM) pattern, separating test logic from UI locators to maximize code reusability and significantly reduce maintenance overhead.

Finally, the entire testing lifecycle is integrated into a GitHub Actions CI/CD workflow, allowing tests to be executed automatically upon code changes, with detailed reports including HTML and Allure.

2 Project Setup

2.1 Prerequisites

Before running the playwright tests, ensure you have the following installed on your system:

- Python (Installed in your device)
- Node (Installed in your device)
- Java (Installed in your device)
- A Code Editor (PyCharm is recommended)

2.2 Installation

- Clone this repository to your local machine.
- Install all prerequisites

To run the project in your local system, you need to install all the libraries listed in requirements.txt.

To install all the libraries at once, go to your project root directory and open terminal. Use the below command:

Listing 1: Terminal Session

```
python -m pip install -r requirements.txt
```

You can also install the libraries manually.

Next step, install the browsers Playwright needs:

Listing 2: Terminal Session

```
playwright install
```

2.3 Setting Up Environment

A .env file is a plain text file used to store environment variables for an application, especially during local development or testing. It follows a simple key-value format, making it easy to manage configuration settings. To setup the project you need to create a .env file using the .env.example file provided in the project repository.

Listing 3: Default .env file used in the project

```
BASE_URL= 'Your Website URL'
ADMIN_USERNAME= 'Your Username'
ADMIN_PASSWORD= 'Your Password'


# Default values you may change
TABLE_URL= 'Google spreadsheet url'
TABLE_NAME='Project Table Name'
TABLE_DESCRIPTION='This table is created from google sheet to
   perform automation test on WP Plugin'
```

3 Run Tests & Report Generation

3.1 Running Tests with Pytest

Pytest is a popular and powerful testing framework for Python. To run tests using Pytest, you typically execute the pytest command from your terminal in the root directory of your project.

The simplest way to run your tests is to call the pytest command with no arguments:

Listing 4: Basic Pytest command

```
pytest
```

3.2 Running Specific Tests

You can specify a file path or directory path after the pytest command. Example:

Listing 5: Basic Pytest command to run specific test

```
pytest tests/test_cases/test_01_login.py
```

3.3 Generating Report With Pytest

3.3.1 Generating HTML Report

HTML reports are excellent for visually reviewing test results. The most popular plugin for this is pytest-html.

First, install the plugin from terminal:

Listing 6: Pytest command to instal pytest-html

```
pip install pytest-html
```

Run your tests using the –html flag to specify the output path for the report. You can also use the –self-contained-html flag to ensure all CSS and images are embedded in the single file, making it easy to share.

Listing 7: Pytest command to genarate self contained html report

```
pytest --html=test_reports/index.html --self-contained-html
```

- –html=path Specifies the output path and filename for the HTML report.
- –self-contained-html Embeds all assets (CSS, JS) into the HTML file so you only have one file to share.

After the tests run, you can find the index.html file in your report/ directory. Open it directly in any web browser to see a detailed summary of passing, failing, and skipped tests.

3.3.2 Generating Allure Report

Generating an Allure Report provides a rich, interactive, and visually appealing summary of your test execution results. Here is how to set up and generate an Allure Report, specifically using Pytest (Python) as the example framework.

First, install it from terminal:

Listing 8: Pytest command install allure-pytest

```
pip install allure-pytest
```

Use the allure generate command to process the raw results into an HTML report structure.

Listing 9: Pytest command to grnerate allure-results

```
pytest --alluredir=test_reports/allure-results
```

Use the allure generate command to process the raw results into an HTML report structure.

Listing 10: Pytest command to grnerate allure-report

```
allure generate test_reports/allure-results -o test_reports/
   allure-report
```

The easiest way to view the report locally is to use the allure serve command, which starts a local web server and opens the report in your default browser.

Listing 11: Pytest command to grnerate allure server

```
allure serve test_reports/allure-results
```

3.3.3 Generate Tracing

Run the following command in your terminal, which will open the viewer in your default browser:

Listing 12: Pytest command to grnerate trace.zip

```
pytest --tracing=retain-on-failure
```

retain-on-failure Generate a trace, but only save the file if the test fails.

4 Project Structure

I used Page Object Model as my project structure to perform all test in this project.The project is organized into the following key directories and files:

4.1 test_cases

Contains all executable test scripts (test_*.py). These files focus solely on test logic and assertions and drive the automation by calling methods from the Page Objects.

4.2 conftest.py

Centralizes resource management by defining Pytest fixtures. This file safely loads environment configuration (secrets, URLs) and initializes/manages Playwright's core components (browser and page), ensuring test isolation and consistent setup/teardown.

4.3 pages/

Contains all Page Object Model (POM) classes. Each class holds the locators and high-level interaction methods (page actions) for a specific application page, abstracting UI details away from the tests.

4.4 utils/

Stores utility scripts for common tasks, such as data handling and configuration processing (e.g., methods for reading and processing CSV data).

4.5 test_data/

Stores all external data files used by the tests, including CSV files and other structured input for data-driven testing.

4.6 test_reports/

The output directory for all generated test results, including HTML reports, JUnit XML reports, and the necessary files for Allure reports.

4.7 screenshots/

Contains all automatically captured screenshots taken by the framework, typically saved upon test failure for quick debugging.

4.8 requirements.txt

Lists all necessary Python libraries (e.g., pytest, playwright, allure-pytest) required to install and run the entire project

4.9 documents/

All project documents. Project pdf files will be available in this section

5 Allure Reports

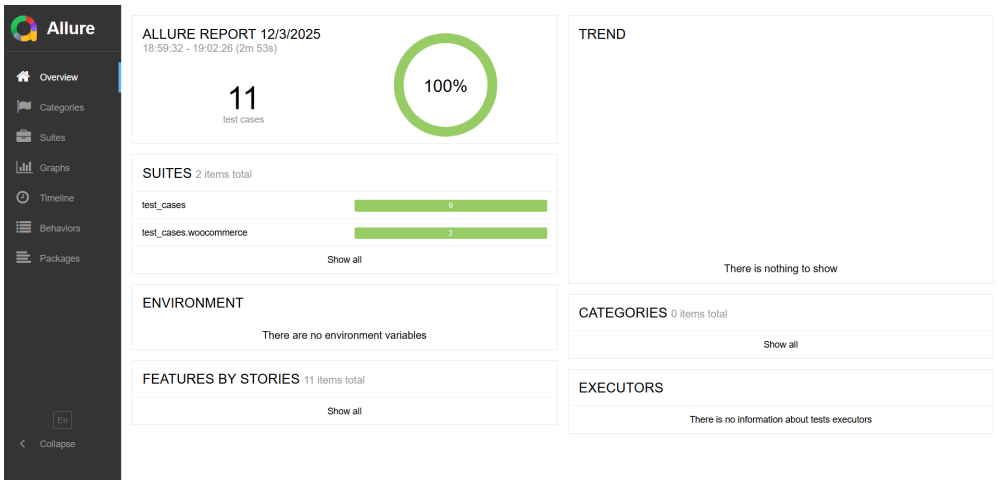

Figure 1: Allure Report

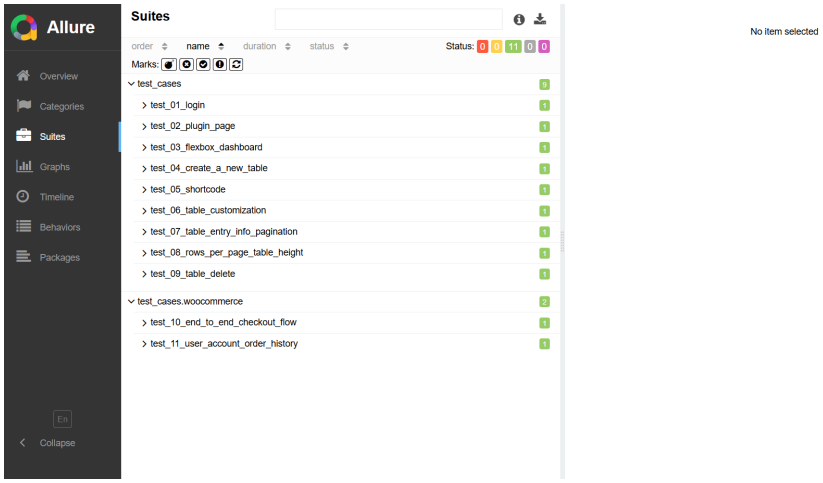

Figure 2: All Tests

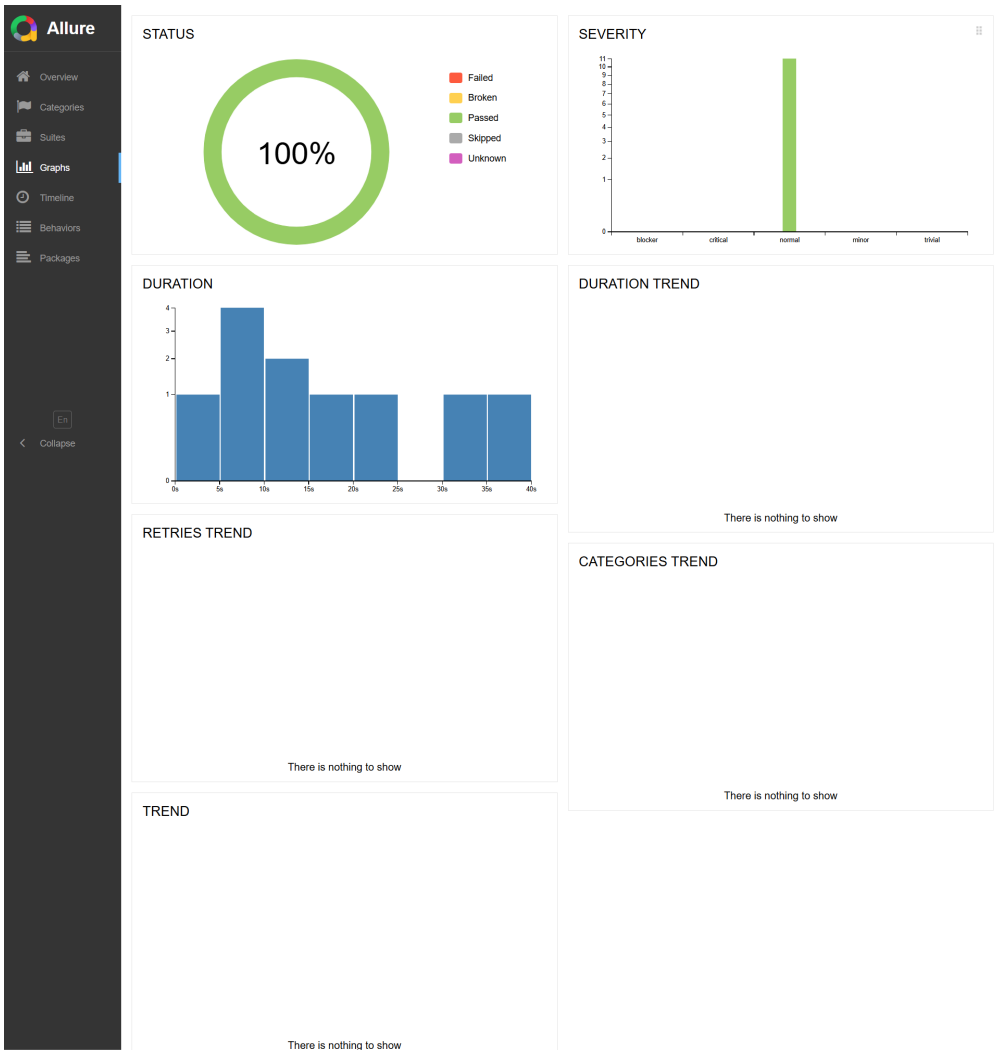

Figure 3: Allure Report Graph

5.1 Part A — WordPress

5.1.1 Test Case 1: Verify WordPress Login Functionality

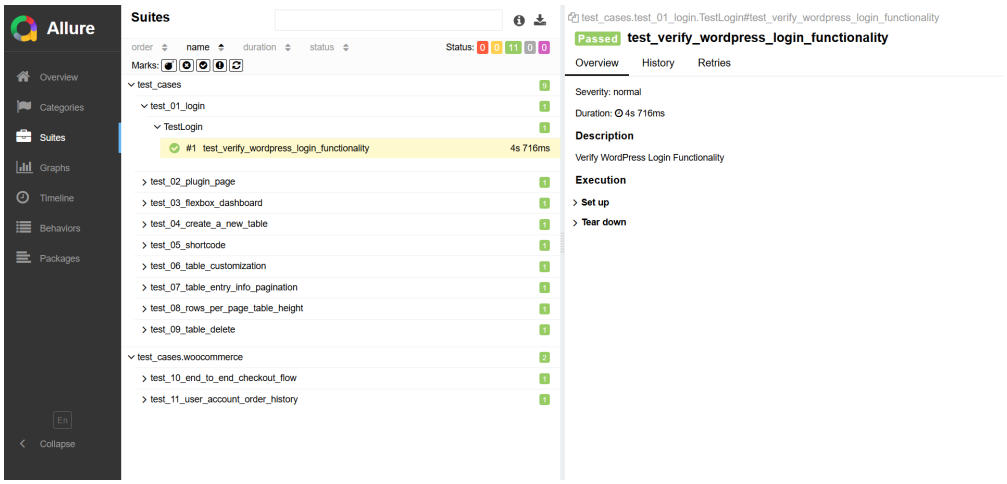

Figure 4: Verify WordPress Login Functionality

5.2 Part B — WooCommerce Test Scenarios

5.2.1 Scenario 1: End-to-End Checkout Flow

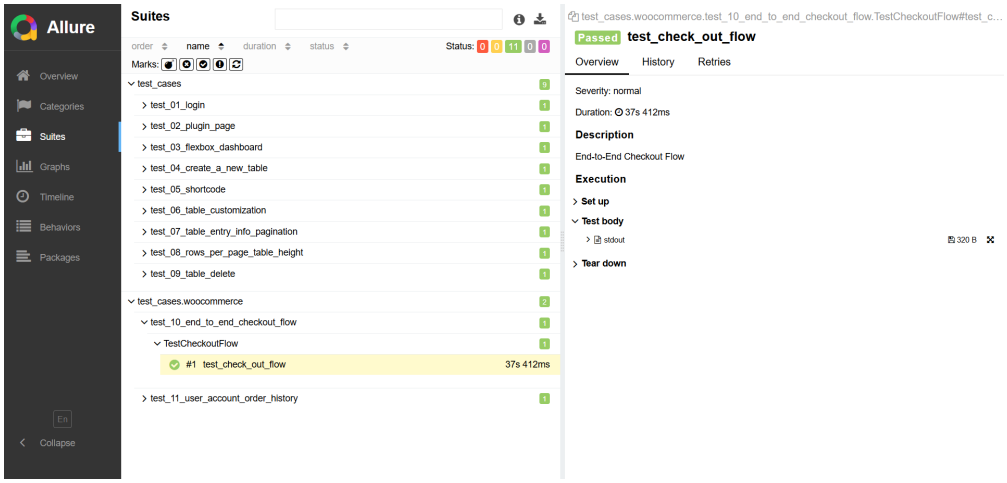

Figure 5: End-to-End Checkout Flow

5.2.2 Scenario 2: User Account Order History

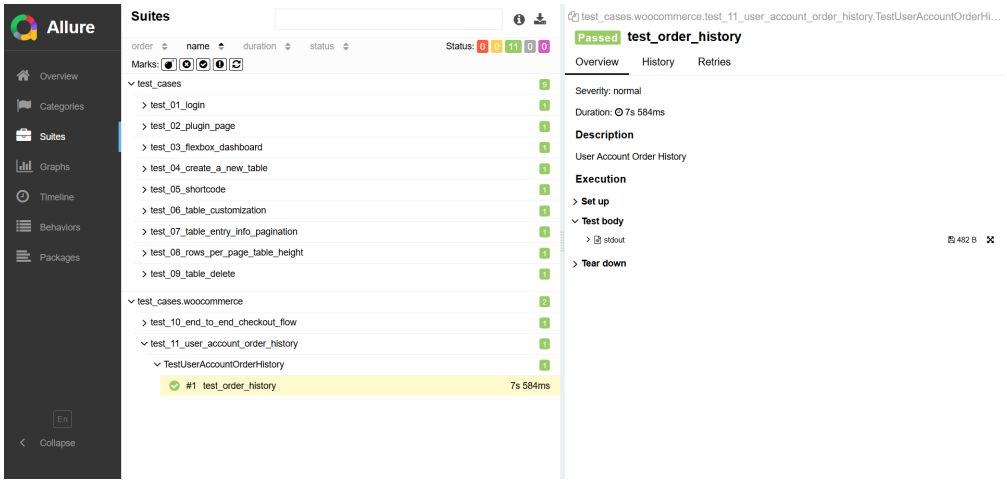

Figure 6: User Account Order History

List of Figures