**Table of Contents**

1. [Overview](#overview)
2. [Package Structure](#package-structure)
   * [Config Package](#config-package)
   * [Drivers Package](#drivers-package)
   * [Pages Package](#pages-package)
   * [TestResults Package](#testresults-package)
   * [Testsuites Package](#testsuites-package)
   * [Utils Package](#utils-package)
   * [WebDriver Package](#webdriver-package)
3. [Core Files](#core-files)
   * [Conftest.py](#conftestpy)
   * [Pytest.ini](#pytestini)
   * [Requirements.txt](#requirementstxt)
4. [Project Pre-requisites](#project-pre-requisites)
5. [Executing Tests](#executing-tests)

**Overview**

The **Generic Automation Framework** is a modular and reusable framework designed to simplify and standardize automation testing across projects. This framework includes packages for configuration, utilities, drivers, page objects, test results, and test suites, along with helper classes like LaunchBrowser and WebDriverHelper. The framework supports modular test creation, cross-browser support, test data management, and integration with CI/CD pipelines, enhancing both maintainability and scalability.

**Package Structure**

**Config Package**

The config package contains test configurations and data initialization files:

* **TestConfig.py**: Holds configuration data like environment URLs and other test-related constants.
* **Test Data (e.g., test\_data.json)**: Contains reusable test data, such as login credentials:

"login\_data": [

{

"username": "Admin",

"password": "admin123"

}

]

**Drivers Package**

The drivers package contains platform-specific browser driver executables:

* **Driver Files**: Chrome, Firefox, Safari, and other browser drivers for Windows, macOS, and Linux.
* **Path Management**: Automatically detects and uses the correct driver based on the OS and browser being tested.

**Pages Package**

The pages package contains page object model (POM) classes:

* **Page Classes**: Each class represents a webpage, containing methods that map to actions and elements on the page.
* **Structure**:
  + **New UI Functionality**: Add a method in the relevant page class.
  + **New UI Feature**: Create a new page class under pages with methods for each action on that page.

Example:

# HrmAdmin.py

class HrmAdminPage:

def open\_admin\_tab(self):

# Method to interact with Admin tab

**TestResults Package**

The testResults package contains test execution results, reports, and logs:

* **Allure Reporting**: Used for reporting test results with detailed logs, screenshots, and execution paths.
* **Setup Allure**:
  1. Install Java and NodeJS.
  2. Run npm install -g allure-commandline.
  3. Use pytest with –alluredir:

pytest test\_xyz.py --alluredir=path\_to\_save\_reports

* 1. View report with:

allure serve path\_to\_report

**Testsuites Package**

The testsuites package organizes tests for various features:

* **Existing Feature**: Add test methods in the relevant test suite file.
* **New Feature**: Create a new test suite file and add tests with appropriate titles and markers.
* **Subfolders**:
  + **Regression**: Contains scripts for UI automation testing.
  + **API**: Contains scripts for API testing and database connections using unittest. mock.

**Utils Package**

The utils package provides helper utilities for data handling, database connections, and API interactions:

* **DB Utils**: Reusable functions to connect to and query databases.
* **FileHandler Utils**: Reads and writes data to/from files.
* **Service API Utils**: Provides methods for GET, POST, PUT, and DELETE API requests.
* **SSH Utils**: Provides methods to connect to remote machines via SSH.
* **PyAuto Utils**: Common methods to handle OS-level operations like file popups.

**WebDriver Package**

The WebDriver package contains reusable methods for launching and managing browsers and Selenium WebDriver commands:

* **LaunchBrowser Class**: Handles browser setup and initialization.
* **WebDriverHelper Class**: Contains over 50 reusable methods for browser interactions like clicking, typing, scrolling, and capturing screenshots.

**Core Files**

**Conftest.py**

conftest.py is used to import plugins or shared fixtures, and to define setup and teardown operations. This file:

* Manages global variables for tests.
* Contains commonly used fixtures that can be shared across all tests.
* Supports custom hooks like setup and teardown.

**Pytest.ini**

pytest.ini defines command-line options and configuration settings for pytest:

* **Execution Parameters**: Manages test execution parameters, including marker definitions and logging levels.
* **TestRail Integration**: Uncomment parameters to enable results upload to TestRail. Modify values based on requirements.

**Requirements.txt**

requirements.txt lists the Python packages and versions required for the project:

* Install dependencies with:

pip install -r requirements.txt

**Project Pre-requisites**

1. **Install Python**: Add Python to the environment variables.
2. **Install PyCharm**: (Optional) Use PyCharm for an IDE optimized for Python projects.
3. **Install Git**: Use Git for version control and clone the project repository into PyCharm.
4. **Install Dependencies**: Run:

pip install -r requirements.txt

**Executing Tests**

**Running a Specific Test**

1. **Navigate to Project Directory**:

cd path\_to\_project\_directory

1. **Run a Specific Test File**:

pytest -s -v tests/test\_file.py -k test\_method\_name

Example:

pytest -s -v tests/test\_login.py -k test\_login\_with\_valid\_credentials

**Running Tests by Marker**

Markers allow you to categorize tests, making it easier to run a specific group:

1. **Run Tests by Marker**:

pytest -s -v -m marker\_name

Example:

pytest -s -v -m regression

**Generating Reports**

1. **Allure Report Generation**:

pytest --alluredir=report\_path

1. **Viewing Allure Report**:

allure serve report\_path