End-to-End QA Automation for E-Commerce Web Applications

1. Project Summary

This project presents a comprehensive Quality Assurance (QA) strategy for three demos e-commerce web applications. The primary objective was to ensure the reliability, functionality, and robustness of critical user journeys through a hybrid testing approach. The project involved creating and executing manual test cases to establish a baseline, followed by developing a scalable Python-based automation framework using Selenium and PyTest to automate regression testing. In total, over 15 critical user flows were tested, validating core functionalities like authentication, session management, shopping cart operations, and the checkout process.

2. Objectives 🎯

- To perform thorough manual testing of the web applications to identify bugs and understand user workflows.
- To design and document comprehensive test scenarios, including positive (happy path) and negative (edge case) tests.
- To develop a reusable and maintainable QA automation framework using Python and Selenium.
- To automate the regression suite for critical features like Login, Registration, Add to Cart, and Checkout.
- To validate form handling, data integrity, and session behavior across the applications.

3. Technologies & Tools Used 💻

- Automation Language: Python 3.9
- Automation Framework: Selenium WebDriver
- Test Runner: PyTest
- Browser Driver Management: WebDriver Manager
- Version Control: Git & GitHub
- Manual Test Case Management: Google Sheets / Zephyr Scale (conceptual)
- Target Applications: Three demo e-commerce sites (e.g., saucedemo.com, demoblaze.com).

4. Project Methodology & Architecture

The project followed a structured, two-phase approach.

Phase 1: Manual Testing & Test Case Design

Before writing any code, a thorough manual testing cycle was conducted. This helped in deeply understanding the application's behavior and identifying areas prone to errors.

Test Case Design: For each feature, a set of test cases was written. Each case included a Test ID, Description, Steps to Reproduce, Expected Result, and Actual Result.

Execution: These test cases were executed manually to establish a quality baseline.

Example: Manual Test Cases for Login Functionality

Test ID	Test Scenario	Steps	Expected Result
TC-L-01	Positive - Valid Credentials	1. Navigate to login page. 2. Enter valid username. 3. Enter valid password. 4. Click Login.	User is redirected to the dashboard/home page.
TC-L-02	Negative - Invalid Password	1. Navigate to login page. 2. Enter valid username. 3. Enter invalid password. 4. Click Login.	An error message 'Invalid credentials' is displayed.
TC-L-03	Negative - Empty Credentials	1. Navigate to login page. 2. Leave username and password blank. 3. Click Login.	Error messages appear below both input fields.
TC-L-04	Negative - SQL Injection	1. Navigate to login page. 2. Enter ' OR '1'='1 in username. 3. Click Login.	The application should not log in and should handle the input safely.

5. Automation Script Implementation (Example: Login Test)

Step 1: Create the Page Object (pages/login_page.py)

```
from selenium.webdriver.common.by import By

class LoginPage:
    def __init__(self, driver):
        self.driver = driver
        self.username_input = (By.ID, 'user-name')
        self.password_input = (By.ID, 'password')
        self.login_button = (By.ID, 'login-button')
        self.error_message_label = (By.CSS_SELECTOR, 'h3[data-test="error"]')
```

```
    def enter_username(self, username):
        self.driver.find_element(*self.username_input).send_keys(username)

    def enter_password(self, password):
        self.driver.find_element(*self.password_input).send_keys(password)

    def click_login(self):
        self.driver.find_element(*self.login_button).click()

    def get_error_message(self):
        return self.driver.find_element(*self.error_message_label).text
```

Step 2: Create the Test Script (tests/test_login.py)

This script uses PyTest and the LoginPage object to run the actual tests.

Python

```
import pytest

from selenium import webdriver

from pages.login_page import LoginPage


# PyTest fixture to set up and tear down the browser for each test

@pytest.fixture()

def driver():

    driver = webdriver.Chrome()

    driver.implicitly_wait(10)

    driver.get("https://www.saucedemo.com")

    yield driver

    driver.quit()


# Test for a successful login

def test_valid_login(driver):

    login_page = LoginPage(driver)
```

```
    login_page.enter_username("standard_user")
    login_page.enter_password("secret_sauce")
    login_page.click_login()

    # Assertion: Check if the login was successful by verifying the new URL
    assert "inventory.html" in driver.current_url, "Login failed with valid credentials"

# Test for a failed login with an invalid password
def test_invalid_login(driver):
    login_page = LoginPage(driver)
    login_page.enter_username("standard_user")
    login_page.enter_password("wrong_password")
    login_page.click_login()

    # Assertion: Check if the correct error message is displayed
    error_msg = login_page.get_error_message()
    assert "Username and password do not match" in error_msg, "Error message for invalid login is incorrect"
```

---

6. Key Achievements & Learnings

- **Comprehensive Coverage:** Successfully designed and documented **100+ manual test cases** covering positive, negative, and UI scenarios for 15+ user flows.
- **Automation Success:** Automated the entire regression suite for core e-commerce workflows (Login, Registration, Search, Add to Cart, Checkout), reducing manual regression testing time by an estimated **80%**.
- **Technical Proficiency:** Gained hands-on experience in building a robust testing framework from scratch using **Python, Selenium, PyTest, and the Page Object Model**.

- **Bug Identification:** Identified and reported **12+ bugs** during the manual and automation cycles, ranging from UI inconsistencies to functional errors in form validation.

---

7. Challenges & Future Scope

- **Challenges:**
    - **Dynamic Elements:** Handling elements that take time to load (AJAX) was challenging. This was overcome by using Selenium's WebDriverWait for explicit waits instead of fixed time.sleep().
    - **Test Data Management:** Maintaining clean test data for different test environments was a key consideration.
- **Future Scope:**
    - **CI/CD Integration:** Integrate the automation suite with a CI/CD tool like **Jenkins** to trigger tests automatically on every new code commit.
    - **Cross-Browser Testing:** Expand the framework to run tests on other browsers like Firefox and Edge using Selenium Grid.
    - **API Testing:** Add an API testing layer using the requests library to test the backend independently of the UI.
    - **Reporting:** Implement Allure Report for generating detailed and interactive test reports.