**Locators and Form Automation with Reporting**

**Task**: Automate a multi-field form on a sample web application, generating a test report.

**Requirements**:

Use various locators (ID, CSS, XPath) to fill out the form (e.g., name, email, password).

Submit the form and validate the success message using assertions.

Produce an HTML or JUnit XML report to confirm test results.

**Data-Driven Testing (Selenium + Pipeline)**

**Task**: Read test data (e.g., from CSV or JSON) to drive multiple test scenarios in Selenium.

**Requirements**:

Implement a data-driven approach (e.g., a loop or parameterized test) for different user inputs.

Integrate this test suite into a CI/CD pipeline (Jenkins or GitLab).

Show pipeline logs verifying multiple test runs with different data sets.

**Login + Logout Flow in Parallel**

**Task**: Create two Selenium test classes: one for verifying login, another for logout. Run them in parallel within a pipeline.

**Requirements**:

Automate a basic login scenario, checking for a dashboard or welcome text.

Automate a logout scenario, verifying redirection to the login page.

Configure your CI/CD tool to run both tests in parallel stages.

**Explicit Waits and Dynamic Content**

**Task**: Automate a page where elements appear or change after loading, ensuring reliability.

**Requirements**:

Use explicit waits (e.g., WebDriverWait) for dynamic elements.

Validate that once the spinner or loading message disappears, you can interact with the final element.

Provide pass/fail logs or screenshots in a test report.

**Containerized Test Execution**

**Task**: Dockerize your Selenium test environment and run it in a CI/CD pipeline.

**Requirements**:

Create a Dockerfile that installs dependencies (e.g., Python + Selenium).

Build and run the container locally to confirm it can execute the tests.

Integrate this container image in your pipeline to run tests automatically.

**Testing on Multiple Browsers in CI**

**Task**: Validate cross-browser compatibility (e.g., Chrome + Firefox) in a pipeline.

**Requirements**:

Configure your test suite to run on at least two browsers.

Use Jenkins or GitLab CI with two stages: one for Chrome, one for Firefox.

Provide logs or screenshots verifying each stage’s pass/fail outcome.

**Handling Failed Tests with Notifications**

**Task**: In your CI/CD pipeline, send notifications (e.g., email or Slack) if Selenium tests fail.

**Requirements**:

Configure a pipeline step that checks the test exit code or parse the test results.

If tests fail, trigger a notification.

Document or screenshot a failing build’s notification.

**Page Object Model (POM) + CI**

**Task**: Implement a Page Object Model for a small web application test, then run it in a pipeline.

**Requirements**:

Create separate page classes (LoginPage, DashboardPage) with clear methods.

Write a test script that uses these page objects.

Integrate the test into your CI tool, verifying a successful or failing run.

**Monitoring Test Automation Performance**

**Task**: Track and log the runtime of each Selenium test in your CI/CD pipeline, identifying slow tests.

**Requirements**:

Add code to measure each test’s duration.

Store or display these durations in your pipeline’s report or logs.

Provide a short reflection on which tests took the longest and potential optimization steps.

**User Registration + Form Validation**

**Task**: Write Selenium tests for a user registration page that has client-side or server-side validation.

**Requirements**:

Input valid and invalid data, expecting different messages.

Use assertions to confirm success or error text.

Provide a final test report with pass/fail results.

**Integration with Cloud CI/CD (AWS or Azure)**

**Task**: Host your test code in a repository, set up a pipeline on AWS CodePipeline or Azure DevOps.

**Requirements**:

Configure the pipeline to clone your code and run Selenium tests.

Show that the pipeline fails if tests fail.

Provide screenshots/logs of a successful pipeline run.

**Data Entry and Assertions**

**Task**: Automate a multi-step form (e.g., shipping address + payment details), verifying each step with assertions.

**Requirements**:

Use Selenium to navigate through each step, checking that the correct summary or confirmation page appears.

Print test logs or generate an HTML report.

Optionally run these tests in a local CI environment.

**Scheduling Test Runs with Jenkins**

**Task**: Schedule daily test runs of your Selenium suite in Jenkins.

**Requirements**:

Create a Jenkins job that pulls the test code from Git.

Set a cron-like schedule (e.g., nightly).

Show logs verifying the scheduled build and test results.

**Form Submission with File Upload**

**Task**: Automate a scenario where the web app allows file uploads.

**Requirements**:

Use Selenium to locate the file upload input.

Upload a sample file, click submit, and assert the success message.

Provide screenshots or logs verifying the file was accepted.

**Parallel Browser Execution**

**Task**: Run Selenium tests in parallel on at least two browsers (Chrome + Firefox).

**Requirements**:

Split your tests into different classes or sets.

Configure a CI tool (or local test runner) to execute them simultaneously.

Show final results, highlighting any differences between browsers.

**Rollback + Rerun in Pipeline**

**Task**: If your pipeline fails due to a Selenium test, automatically revert the last code change and rerun.

**Requirements**:

Create a pipeline step that detects failure.

If tests fail, revert the commit (or mark it for manual rollback), then attempt a rerun.

Provide logs showing the pipeline logic and final outcome.

**Branch-Specific Test Suites**

**Task**: Configure a pipeline to run different Selenium test suites based on the branch name.

**Requirements**:

For feature/\* branches, run a quick smoke test.

For main, run the full regression test.

Document your pipeline configuration and show logs from each branch scenario.

**Advanced Element Locators + Waits**

**Task**: On a dynamic page (e.g., with AJAX calls), use complex XPath/CSS locators and explicit waits.

**Requirements**:

Identify elements that only appear after some delay.

Use WebDriverWait or similar mechanism.

Provide test logs showing the steps and final assertion that the dynamic content is correct.

**Pull Request Validation with Test Automation**

**Task**: For each new pull request, automatically run Selenium tests to validate the changes.

**Requirements**:

Use a CI tool (GitHub Actions, GitLab CI, or Jenkins) that triggers on pull requests.

Provide pass/fail checks on the PR.

Show that merging is blocked if tests fail, ensuring code quality.