**Selenium Java Automation Testing Project Documentation**

**Capstone Project – Demo Web Shop**

**1. Project Overview**

**Project Name:** Capstone Project – Demo Web Shop Automation

**Project URL:** https://demowebshop.tricentis.com/

**Problem Statement:**

This s a comprehensive e-commerce platform that allows users to log in, search products, add to cart and filter the product. Users can add products to their cart, edit orders, and remove items from the cart. The checkout process involves clicking the Check Out button, entering shipping details, and payment through card. Given the critical nature of these functionalities, any failure in the system can lead to significant user dissatisfaction and potential revenue loss. Applying automated testing solution to maintain the highest standards of performance and user experience.

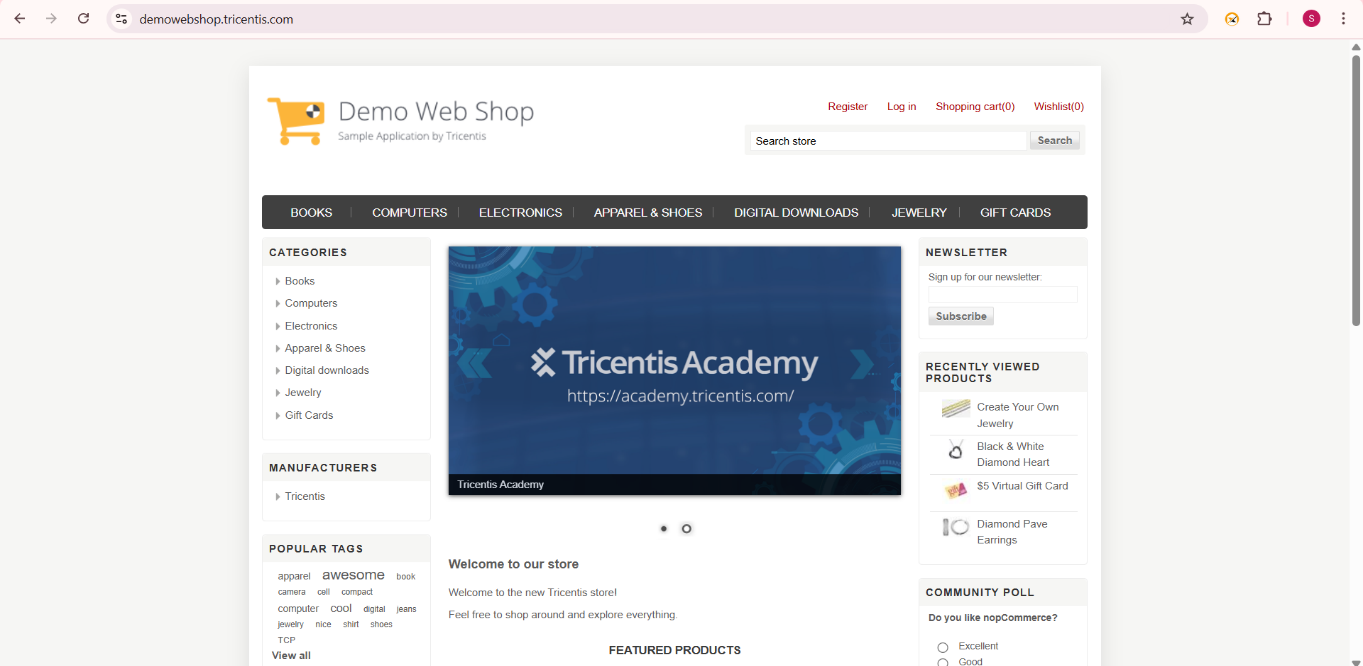
**Objective:**

The objective of this project is to implement automated testing for the Online Shopping site using Selenium, TestNG. This automation will cover UI testing to ensure that the website functions correctly by using POM or Page Factory, providing a reliable and user-friendly experience for customers.

**Goal:**

Write few test cases (both positive and negative) and sample defects from the template provided. Automate the scenarios/test cases written using the framework mentioned in the Objective section.

**Website Interface:**

****

**Tools & Technologies Used:**

* **Programming Language:** Java
* **Testing Framework:** TestNG
* **Behavior-Driven Development (BDD):** Cucumber
* **Automation Tool:** Selenium WebDriver
* **Build Tool:** Maven
* **Reporting:** Extent Reports
* **IDE:** Eclipse
* **Data Management:** Excel file (for test data) and properties file (for configuration)

**2. Project Objective**

The objective of this project is to automate the testing of key functionalities of an e-commerce Demo Web Shop website. The project covers the following test cases:

* **User Registration**
* **User Login with valid and invalid Credentials**
* **Search for product and then add to cart**
* **Adding multiple Jewelry’s to Cart**
* **Validating Successful Purchase (Checkout)**
* **Update and remove product from cart**
* **Logging out after valid login**

**3. Project Structure**

The project is organized into multiple packages and classes for better maintainability and readability.

**3.1. Package Structure**

src/test/java

|── base # Base class for WebDriver initialization

|── feature # Cucumber feature files

|── stepdef # Cucumber step definitions

|── hooks # Cucumber hook

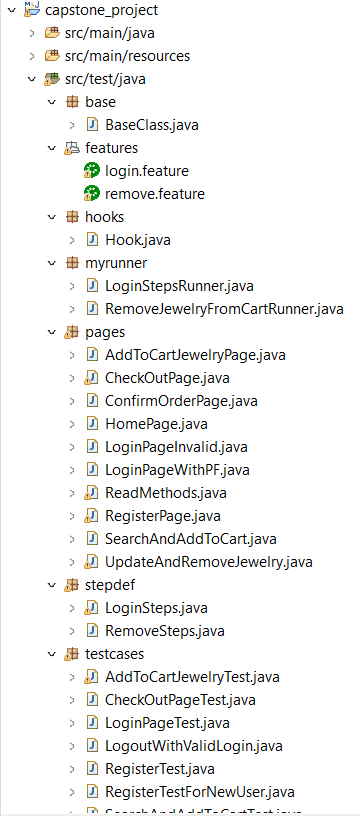
|── myrunner # Cucumber runner for executing tests

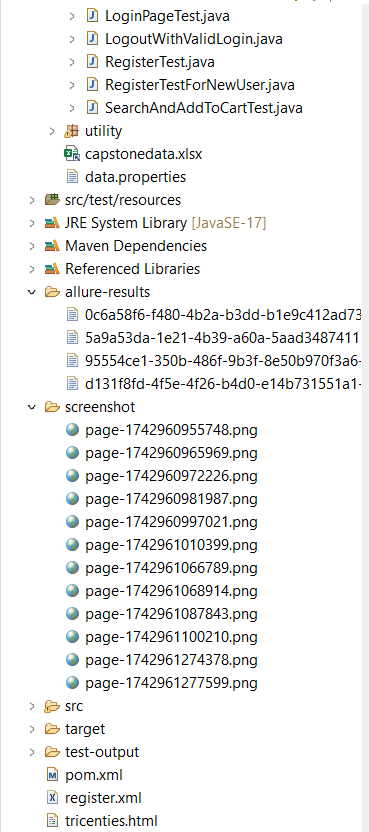
|── pages # Page Object Model (POM) classes for UI interactions

|── testcases # Test classes for different scenarios

|── utility # Utility classes (Excel reader, Extent Report generator)

|── resources # Data files (Excel, properties)

****

****

**4. Key Components**

**4.1. Base Class**

* The BaseClass.java handles the WebDriver initialization and browser configurations.

1. Browser Initialization (invokeBrowser)

* Accepts a browser name as input (chrome, firefox, or edge).
* Launches the specified browser using WebDriver and maximizes the window.
* Displays an error message for invalid browser names.

2. Screenshot Capture (screenshot)

* Takes a screenshot of the current page using TakesScreenshot.
* Saves the screenshot in the ./screenshot/ folder with a timestamped filename.

3. Allure Reports

1.@BeforeSuite – Cleaning Previous Allure Results:

* The clearAllureResults() method, annotated with @BeforeSuite, ensures that old Allure result files are deleted before starting a new test execution.
* It iterates through the allure-results folder and removes all existing files, ensuring that the new test execution produces a clean and accurate report without outdated or conflicting data.
* This prevents the report from displaying stale or duplicate test results.

2.@AfterSuite – Generating the Allure Report:

* The generateAllureReport() method, annotated with @AfterSuite, automatically triggers the Allure report generation after all tests have been executed.
* It uses ProcessBuilder to execute the Maven command mvn allure:report, which generates the Allure report based on the test execution results.
* The method waits for the report to be generated and verifies if the report was successfully created by checking the allure-results directory.
* If the report is successfully generated, it displays the report path in the console; otherwise, it logs an error message.

**4.2. Utility Classes**

* **ExcelReader.java:**
  + Used for reading test data from an Excel file.
  + Data is fetched based on row and column indices.
* **ExtentReport.java:**

The ExtentReport.java file is responsible for generating detailed HTML reports for the test execution.

* **Report Initialiation:**
  + Uses ExtentReports and ExtentSparkReporter to generate reports.
  + Sets the report theme to **DARK** for better readability.
* **Report Creation:**
  + Creates a new HTML report named tricenties.html.
  + Adds test cases dynamically to the report using createTest().
* **Report Attachment:**
  + Attaches the sparkReporter to the ExtentReports instance.
  + Allows tracking of test execution status with visual logs.

**5. Test Cases**

**5.1. User Registration Test**

**Class:** RegisterTest.java  
**Description:** Automates the registration process using Selenium WebDriver ,testing and Page Object Model.  
**Steps:**

1. **Reading Properties File:**

* The @BeforeTest method reads the **URL** from the data.properties file using FileInputStream and initializes the **ExtentReport** instance.

2.**Launching the Browser:**

* The @Test method uses the @Parameters annotation to run on different browsers.
* It invokes the browser, navigates to the URL, and applies **implicit wait** for stable execution.

3.**Registering a User:**

* Uses the RegisterPage class to interact with the **registration page** elements.
* Reads **user details** (first name, last name, email, password, and confirm password) from an Excel file using the ExcelReader utility.
* Fills the form and clicks the **register button**.

4. **Logging Execution Status:**

* The test logs execution status using **ExtentReport**, marking the test as **PASS** if registration is successful.

5.**Closing the Browser:**

* The @AfterTest method closes the browser using driver.quit() and flushes the Extent Report data.

**5.1.2 Register with new User**

**Class:** RegisterTestForNewUser.java

**Description:** Automates The registration process for new user .

**Steps:**

* Read the URL from a data.properties file.
* Used parameterization to run tests on multiple browsers.
* Filled the registration form with data retrieved from an Excel file.
* Verified the success message after registration.
* Captured screenshots in case of test failures.
* Logged test execution using ExtentReport and generated a final report.

**5.2. User Login Test**

**5.2.1 User Login Test for valid Credentials**

**Class:** LoginPageTest.java  
**Description:** Automates the login process.  
**Steps:**

1. **Reading Properties File:**

* The @BeforeTest method reads data from the data.properties file using FileInputStream.
* It extracts the **URL, email, and password** for the test and initializes the **ExtentReport** instance.

2. **Launching the Browser:**

* The @Test method uses the @Parameters annotation to run the test on different browsers.
* The invokeBrowser() method opens the specified browser and navigates to the URL.
* It applies **implicit wait** for smooth execution.

3.**Performing Login Actions:**

* Uses LoginPageWithPF (Page Factory) to interact with the **login page** elements.
* Enters the **email and password**, clicks the "Remember Me" checkbox, and logs in.
* ExtentReport logs the **login details** and updates the report status.

4. **Validating and Logging Test Status:**

* Uses ExtentReport.createTest() to log the execution status as **PASS**.
* Sleeps for 2 seconds to allow the actions to complete.

5. **Closing the Browser:**

* The @AfterTest method closes the browser using driver.quit() and flushes the Extent Report data.
* This ensures proper cleanup after the test execution.

**5.2.2 User Login Test for invalid Credentials**

**Description:** Automates login test case for invalid Credentials

**Feature File:**

* You created a login.feature file that defines the test scenario in Gherkin syntax.
* The scenario includes steps to:
* Open the browser.
* Navigate to the Demo Web Shop URL.
* Enter invalid credentials (email and password).
* Click the login button.
* Verify that an error message is displayed.
* You used Examples to specify different sets of invalid credentials in a data table.

**Step Definitions:**

* You created LoginSteps.java to define the actual steps linked to the feature file.
* Implemented methods for:
* Opening the browser (invokeBrowser("chrome")).
* Navigating to the URL.
* Entering email and password.
* Clicking the login button.
* Verifying the error message.
* Added implicit waits for stable execution.
* Used Extent Reports for logging test data and results.

**Hooks:**

1.Test Setup and Execution:

* Used the @Before annotation to initialize the Extent Report and launch the Chrome browser before each test scenario using the invokeBrowser("chrome") method.

2.Test Cleanup and Reporting:

* Used the @After annotation to close the browser and flush the Extent Report after each test scenario, ensuring the test execution details are saved.

**Page Object Model (POM):**

* You implemented LoginPageInvalid.java to follow the Page Object Model (POM) design pattern.
* Defined WebElements using By.xpath() locators for:
* Email input field.
* Password input field.
* Login button.
* Error message.
* Created methods for interacting with these elements:
* enterEmailId() → Returns the email input field.
* enterPwd() → Returns the password input field.
* clickOnLoginBtn() → Clicks the login button.
* getErrorMsg() → Fetches the displayed error message.
* Added reusable methods for interacting with the login page.

**Runner Class:**

* You created the LoginStepsRunner.java class to run the Cucumber test.
* Used the @RunWith(Cucumber.class) annotation.
* Included CucumberOptions to specify:
* The location of the feature file.
* The step definitions package.
* Used tag i.e @login
* Added plugin = {“pretty”} for readable output.
* Enabled monochrome = true for better console readability.
* Extended the AbstractTestNGCucumberTests class to run the tests with TestNG.

**5.3**.**Search for product and then add to cart**

**Class:** SearchAndAddToCartTest.java

**Description:** Automates the search for a product and after searching for the product then adding that product to cart.

**Steps:**

1.Property File Reading and Initialization:

* Read the application URL from the data.properties file using FileInputStream and Properties, and initialized the ReadMethods class and Extent Report in the @BeforeTest method.

2.Browser Launch and Navigation:

* Launched the browser based on the parameter provided (browser) and navigated to the URL with implicit wait settings.

3.Product Search and Filtering:

* Performed product search by interacting with the search bar, applying filters (category, manufacturer, price range), and executing the search.

4.Add to Cart and Screenshot:

* Sorted and displayed products, scrolled down using JavascriptExecutor, added jewelry to the cart, and captured a screenshot.

5.Test Cleanup and Reporting:

* Flushed the Extent Report and closed the browser in the @AfterTest method.

**5.4. Adding multiple products to Cart Test**

**Class:** AddToCartJewelryTest.java  
**Description:** The AddToCartJewelryTest.java file contains the test cases to automate the "Add to Cart" functionality for multiple jewelry products using Selenium and TestNG.  
**Steps:**

**1**. Property File Reading and Initialization:

* Read the application URL from the data.properties file using FileInputStream and Properties, and initialized the Extent Report in the @BeforeTest method.

2. Browser Launch and Navigation:

* Launched the browser based on the browser parameter, navigated to the URL, and applied implicit wait settings.

3. Product Selection and Adding to Cart:

* Selected jewelry products by applying filters (sort by name, display per page), added one product to the cart, and took a screenshot.

4. Adding a Second Product:

* Added a second product by selecting material, length, pendant type, and quantity, then added it to the cart and captured a screenshot.

5. Validation and Reporting:

* Verified the success message using Assert, logged the test result in Extent Report, and closed the browser in the @AfterTest method.

**5.5. Validating Successful Purchase (Checkout)**

**Class:** CheckOutPageTest.java

**Description:** Automates the Checkout page by logging in Demo web shop website and added product to cart and then proceed with checkout.

**Steps:**

1.Property File Reading and Initialization:

* Read the application URL, coupon, gift card, pincode, city, address, and phone number from the data.properties file, initialized the ReadMethods class, and started the Extent Report in the @BeforeTest method.

2. Browser Launch and Login:

* Launched the browser based on the browser parameter, navigated to the URL, and performed the login using the LoginMethod() from the ReadMethods class.

3. Product Selection and Checkout:

* Added jewelry to the cart, applied a coupon and gift card, selected shipping details, and proceeded with the checkout process while capturing logs in the Extent Report.

4. Order Confirmation:

* Entered billing and shipping details, selected payment options, confirmed the order, and captured screenshots of the confirmation and order details.

5.Test Cleanup and Reporting:

* Saved the Extent Report, captured the order details in PDF format, and closed the browser in the @AfterTest method.

**5.6. Update and remove product from cart**

**Class:** RemoveJewelryFromCartRunner.java

**Description**: Automates Update and Remove the product which is present in cart by using BDD-Cucumber.

**Steps:**

1.Feature Definition and Scenario Execution:

* Created a Cucumber feature file remove.feature to automate the removal of a product from the cart, using scenario outlines with parameters for URL, email, and password.

2.Step Definition Implementation:

* Added step definitions in the RemoveSteps class to perform login, add jewelry to the cart, update the cart, and remove the product while utilizing the ReadMethods and UpdateAndRemoveJewelry classes.

3.Test Execution and Reporting:

* Integrated Extent Report to log the test execution status, including a screenshot upon successful product removal.

4.Hooks for Setup and Teardown:

* Used @Before and @After hooks to initialize the Extent Report, launch the browser before the test, and close the browser while flushing the report after the test.

5.Test Runner with Cucumber and TestNG:

* Created a TestNG runner class RemoveJewelryFromCartRunner with Cucumber options specifying the feature file, glue code, plugins for reporting, and the @remove tag to execute the test.

**5.7. Logging out after valid login**

**Class:** LogoutWithValidLogin.java

**Description:** Automates the Logout Functionality.

**Steps:**

1.Test Setup and Configuration:

* Loaded the URL, email, and password from the data.properties file and initialized the ReadMethods class for test execution.

2. Browser Invocation and Navigation:

* Launched the browser using TestNG parameters, navigated to the provided URL, and applied implicit wait for stability.

3.Login with Valid Credentials:

* Used the LoginMethod from the ReadMethods class to log in with valid credentials and verified the login process.

4.Logout Functionality:

* Automated the logout process by interacting with the HomePage class, capturing a screenshot upon successful logout.

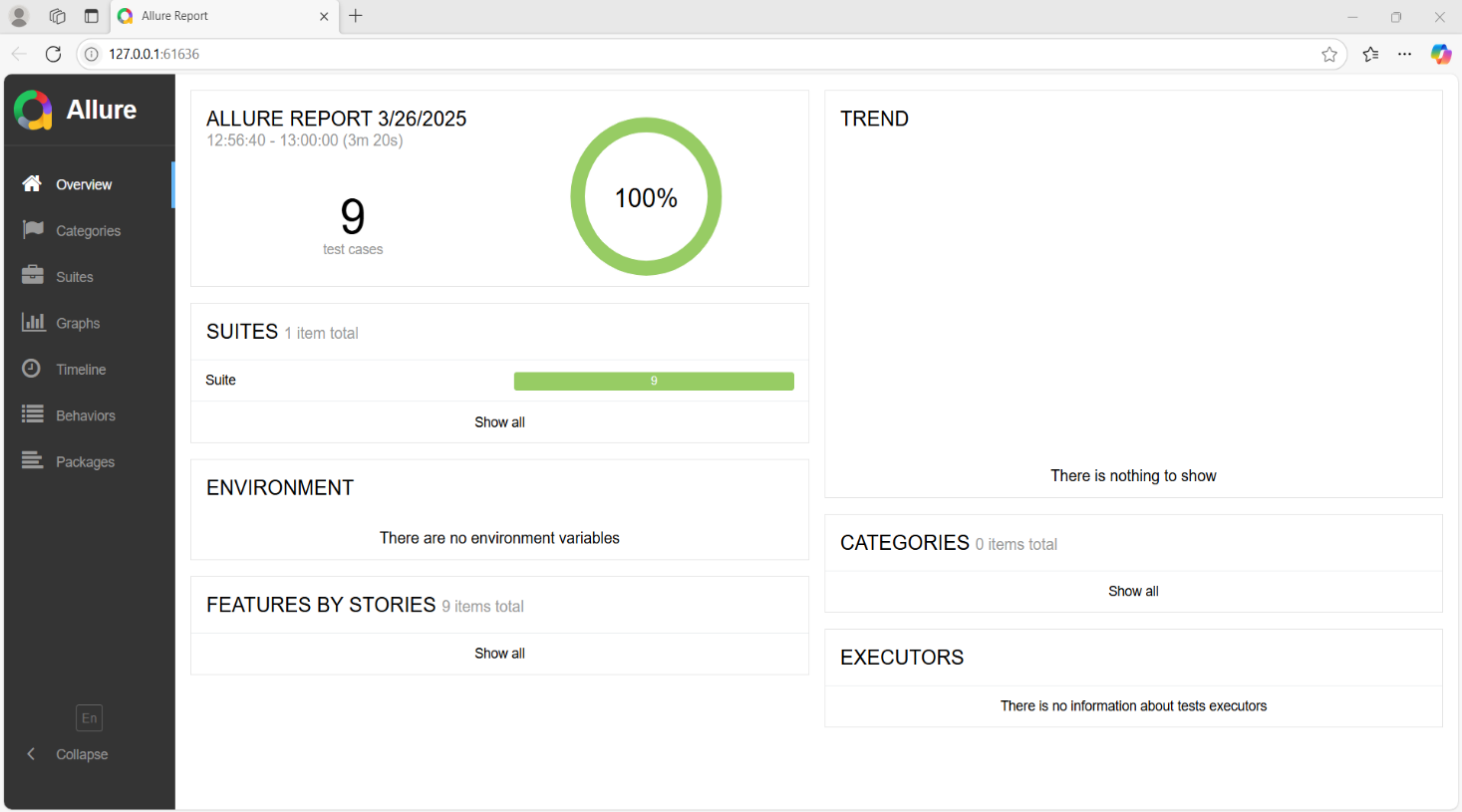
5.Test Execution and Reporting:

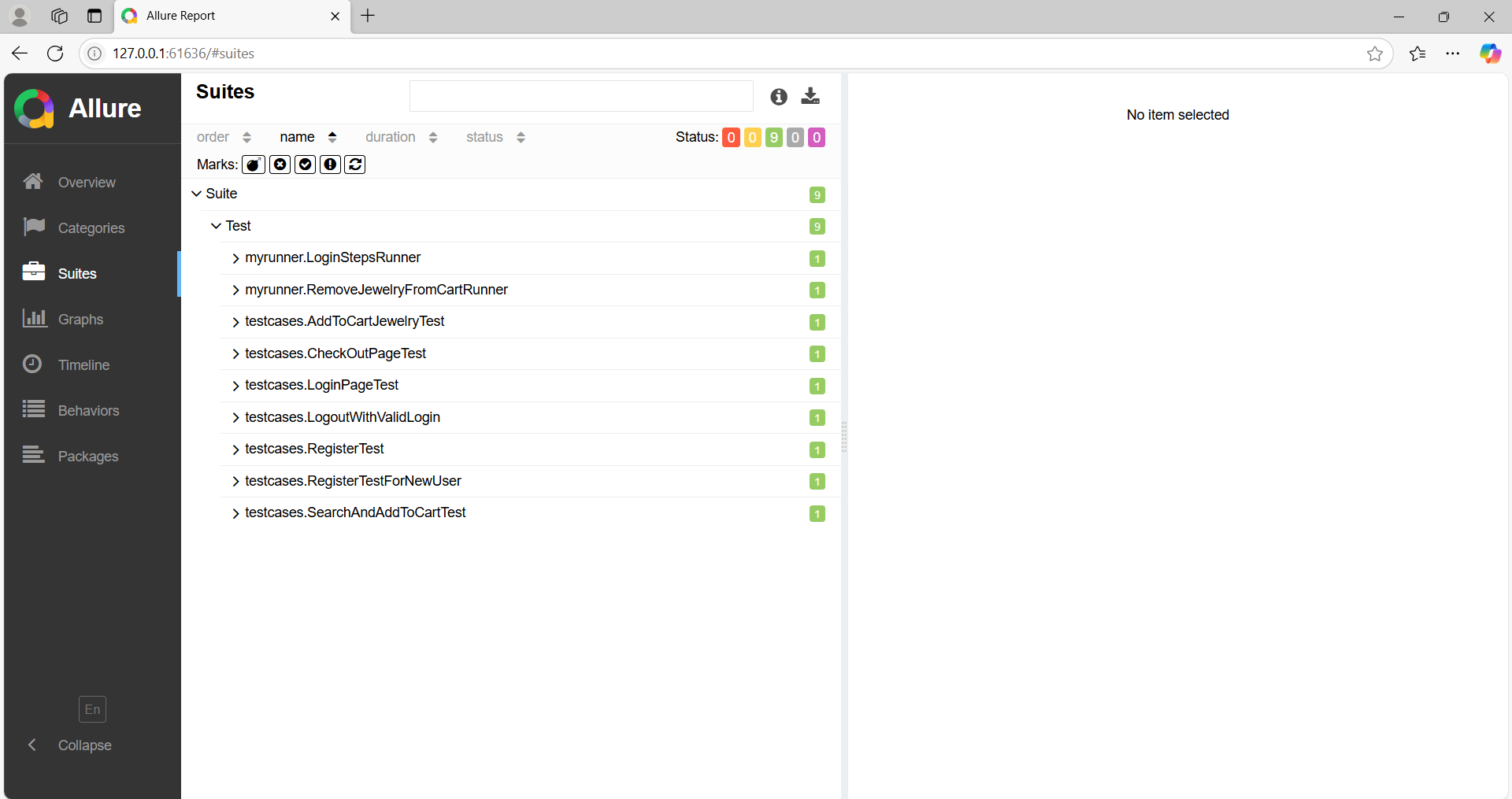
* Integrated Extent Report to log the test status, and ensured the browser session is properly closed using driver.quit() after test execution.

**6. Reporting**

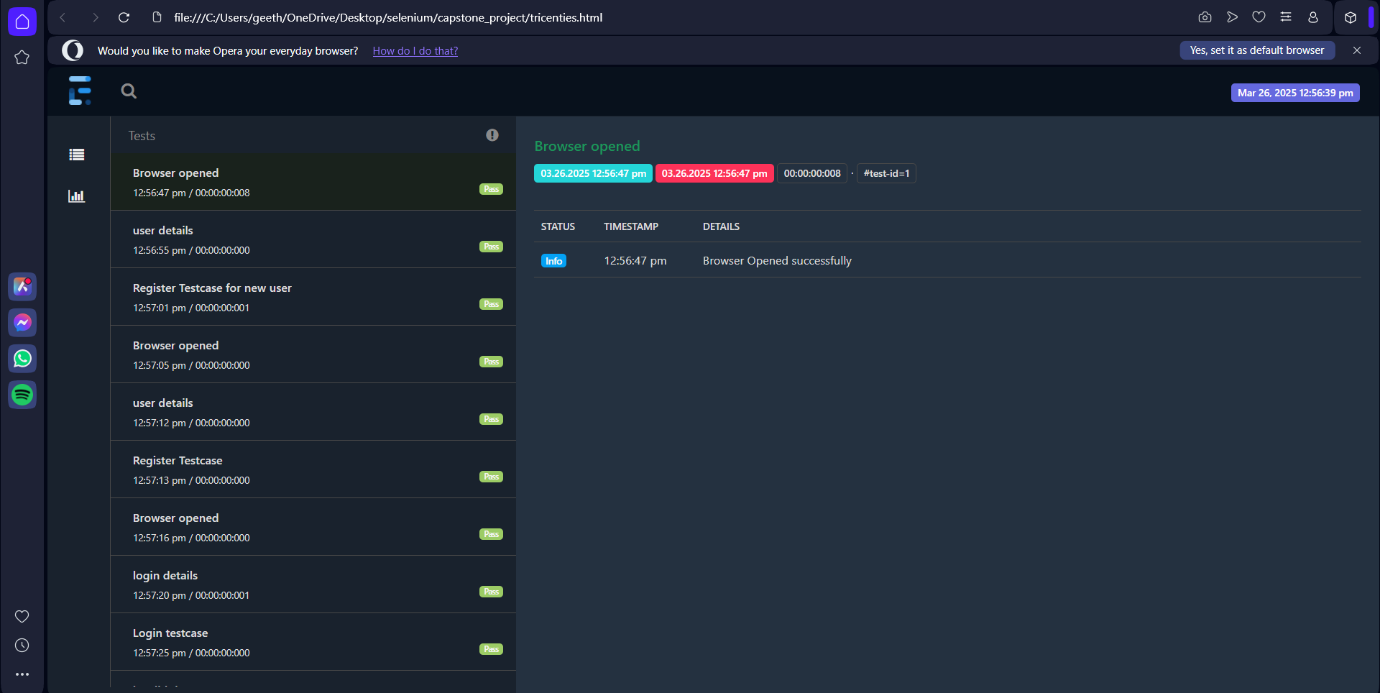
* **Extent Reports**:
* The project uses Extent Reports to generate detailed and customizable test reports.
* Each test execution generates a new report with the pass/fail status, execution time.
* **Allure Reports:**
* The combination of @BeforeSuite and @AfterSuite ensures that the Allure reports are always **fresh and accurate**, reflecting only the current test execution.
* This setup **automates the report generation** process, making it seamless and efficient without manual intervention.
* The generated Allure report provides a **detailed and visually appealing representation** of the test execution, including steps, status, execution time, and error details.

**Allure Report:**

****

****

**Extent Report:**

****

**7. Execution Flow**

1. **Execution :**
   * The project uses TestNG for test execution, with parameters for browser configuration.
   * Execute with .xml file.
2. **Reports:**
   * After execution, Extent Reports are generated in the test-output folder.
   * Reports include details of each test case, execution time.
   * Allure Report is genereated in allure-results folder.

**8. Challenges and Solutions**

* **Challenge:** Code Reusability and Modularity
  + Problem: Repeated code for common actions like login, adding products, and cart operations.
  + Solution:
    - Created a ReadMethods utility class with reusable methods.
    - Applied Page Object Model (POM) to modularize test scripts, improving readability and maintainability.
* **Challenge:** Synchronization Issues:
  + Handled using Thread.sleep() and driver.manage().timeouts().implicitlyWait(), Explicit wait, and fluent wait for stability.
  + Added **implicit and explicit waits** to ensure elements are fully loaded before interaction.
* **Challenge:** Proper Test Teardown and Resource Management

**Problem:** Browser sessions not closing properly.

**Solution:**

* + Added @AfterTest method to **close and quit the driver** after each test execution.
  + Included null checks before quitting to avoid NullPointerException.

**9. Conclusion**

The **Demo Web Shop Automation Project** successfully demonstrates end-to-end testing of an e-commerce platform using **Selenium, TestNG, and Cucumber**. It effectively validates critical functionalities such as **user registration, login, product search, cart management, checkout, and logout**, ensuring the platform's reliability and performance. The use of **Page Object Model (POM)** and **reusable utility classes** enhances code maintainability and readability. Additionally, **Extent Reports and Allure Reports** provide detailed insights into test execution, aiding in efficient debugging and validation. The project's modular structure, comprehensive test coverage, and robust reporting mechanism make it scalable and easy to maintain, highlighting best practices in automation testing.