**SELENIUM AUTOMATION TESTING**

**CAPSTONE PROJECT - JPETSTORE**

**Project Name:** Automation Testing of JPetStore E-Commerce Website

**Introduction**

The Automation Testing Capstone Project focused on the website marked as significant milestone in the participants' learning journey. The project provided an opportunity to apply knowledge and skills acquired during the automation testing training, incorporating Selenium, TestNG, Cucumber, Apache POI, a hybrid framework, and advanced reporting tools like Allure/Extent. The objective is to create a comprehensive automation testing solution, fostering practical experience and showcasing participants' abilities to potential employers.

**Problem Statement**

JPetStore, being an online shopping portal, requires thorough testing to ensure its functionalities work seamlessly. The objective is to **automate major user workflows**, including registration, login, searching for products, adding to cart, and checkout processes, to ensure **faster execution** and **reliable results**.

**Goal**

The goal of this project is to automate the end-to-end user workflows of the JPetStore application, including user registration, login, product search, cart management, and the checkout process. It aims to develop a robust, scalable, and maintainable hybrid automation framework by leveraging Selenium WebDriver, TestNG, BDD-Cucumber, and Apache POI for data-driven testing. Additionally, the project focuses on generating advanced reports using Extent and Allure, ensuring cross-browser compatibility, and providing comprehensive documentation for easy execution and future maintenance.

**Tools and Technologies Used**

* **Selenium WebDriver**
* **TestNG**
* **Cucumber (BDD Framework)**
* **Apache POI**
* **Maven**
* **Extent Reports / Allure Reports**
* **Java**
* **Page Object Model (POM)**
* **Rest Assured (Optional for API Testing)**
* **Git / GitHub**
* **Eclipse**
* **Jenkins**

**Objective**

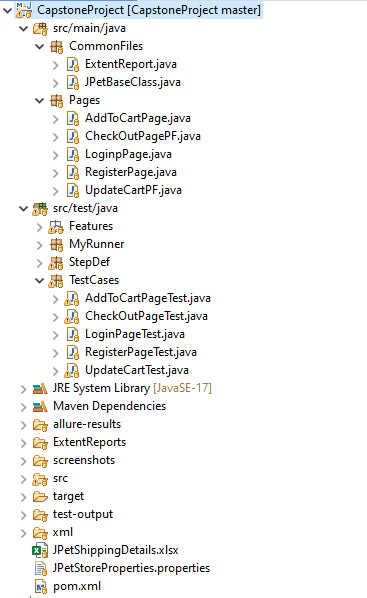
The main objective of this capstone project is to design and implement a **robust, scalable, and maintainable automation testing framework** for the **JPetStore E-Commerce Application**. The goal is to automate critical user workflows, ensuring the application's functionality, reliability, and performance are thoroughly tested, while reducing manual testing efforts and increasing test efficiency.

This project showcases practical skills in **Selenium WebDriver, TestNG, BDD-Cucumber, Apache POI**, and advanced reporting tools like **Extent** and **Allure**, simulating real-world automation scenarios faced by professional SDETs.

**Project Test Cases**

* User Registration
* User Login
* Pet Details
* Add to Cart
* Update Cart
* Payment/Checkout

**Project Structure**

* I have created Maven Project with the name of Capstone Project.
* I have created the Package with the name of **Common Files** containing 2 classes which is common to all.
* I have created the another package with the name of **Pages** containing 5 different pages.
* Feature Package where I have implemented the feature file.
* StepDef Package
* My Runner Package
* Then I have created the **TestCases** package containing 5 testcases.

1. **Maven project 🡪 Name: CapstoneProject**

To begin the automation testing for the JPetStore website, a **Maven** project was created to efficiently manage dependencies and ensure a structured test framework. Maven simplifies the project setup by handling required libraries such as **Selenium WebDriver, TestNG, Cucumber, Apache POI**, and reporting tools like **Extent Reports and Allure.**

1. **Package 1 🡪 Name: CommonFiles**

After setting up the Maven project, the **CommonFiles** package was created to hold all the reusable components and utilities that are shared across different test cases and page classes. This package promotes reusability, reduces code duplication, and maintains consistency throughout the automation framework. It includes the following key components:

* 1. **ExtentReport.java**
* This class is responsible for setting up and configuring the Extent Reports. It helps in generating detailed, structured, and interactive HTML reports that provide a comprehensive view of test execution status, logs, and screenshots of failed step.
  1. **JPetBaseClass.java**
* Acts as a base class for all the test classes.
* It manages the setup and teardown of WebDriver instances.
* It initializes browser configurations, loads property files, and helps in managing preconditions required before executing any test cases.
* It provides common methods like launching browsers, closing browsers, and initializing page objects.

1. **Packages 2 🡪 Name: Pages**

After setting up the CommonFiles, the **Pages** package was created following the Page Object Model (POM) design pattern. This package contains classes that represent the different web pages and their elements on the JPetStore website. Each class is responsible for encapsulating the page's locators and the actions that can be performed on those elements.

The goal of this package is to improve code readability, reusability, and maintainability by separating the page structure and interaction logic from the actual test scripts.

**3.1 RegisterPage.java**

* Handles the elements and actions required for user registration.
* Includes methods for filling out the registration form and submitting it.

**3.2 LoginPage.java**

* Contains locators for login-related elements (username, password, sign-in button).
* Provides methods to interact with these elements, like entering username/password and clicking the sign-in button.

**3.3 AddToCartPage.java**

* Manages the add-to-cart functionality by identifying product elements and adding them to the cart.

**3.4 UpdateCart.java**

* Allows updating the cart by modifying quantities or removing products.

**3.5 CheckOutPage.java**

* Represents the checkout page and includes methods for entering shipping/payment details and placing the order.

**4.Package 3 🡪 Name: TestCases**

The **TestCases** package is designed to implement automated test cases for the critical functionalities of the JPetStore application. It focuses on verifying the correctness of essential workflows such as user login, registration, searching for products, adding products to the cart, updating cart items, and performing the checkout process.

**4.1 RegisterPageTest.java**

* Automates the registration process by filling out user details and submitting the registration form.
* Validates the successful creation of a new user account.

**4.2 LoginPageTest.java**

* Automates the login functionality using valid and invalid credentials.
* Verifies whether the user is successfully logged in or receives appropriate error messages.

**4.3 AddToCartPageTest.java**

* Tests the ability to search for products, view product details, and add items to the shopping cart.
* Verifies whether items are correctly added to the cart.

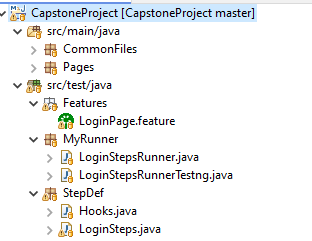
**4.4 UpdateCartTest.java**

* Automates updating the quantity of products in the cart.
* Verifies whether cart updates (quantity changes/removal) are correctly reflected.

**4.5 CheckOutPageTest.java**

* Tests the complete checkout process, including confirming cart items, entering payment/shipping details, and placing the order.
* Validates whether the checkout is successful and the order confirmation message is displayed.

1. **Cucumber Framework for Login Page**

****

**fig: Screenshot of the cucumber framework**

**5.1 Package 4 🡪 Name: Feature**

* The feature file is where we write the test scenarios in plain English using the **Given-When-Then** format. For the login functionality, I created **LoginPage.feature.** It describes steps like opening the JPetStore login page, entering the username and password, clicking the login button, and verifying if the user is logged in successfully.

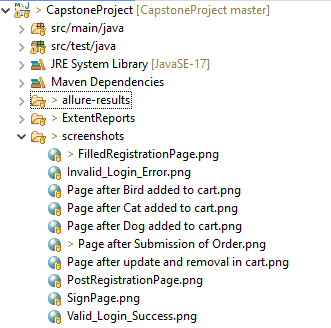
**5.2 Package 5 🡪 Name: StepDef**

* The step definition file connects each line in the feature file to actual code. In my project, I created **LoginSteps.java** in the StepDef package. It has the code that interacts with the webpage elements like entering text, clicking buttons, and checking results. This is where Selenium WebDriver actions are written.

**5.3 Package 4 🡪 Name: MyRunner**

* The runner class runs the feature files. I used LoginStepsRunnerTestng.java as the runner class. It tells Cucumber where the feature files and step definition files are located. It also helps in running the tests through **TestNG** and generating reports.

**6.Folder: Name 🡪 Screenshots**

****

**Fig: Screenshot of the “Screenshots” folder**

I have created the Screenshots folder to capture the screen shots.

**6.1 RegisterPage.png**

* Captures the **user registration page** either before the user fills out the form or after successful form completion.

**6.2 Error Page after completion of Registration.png**

* This screenshot captures the **error message** that appears when a user tries to register with **invalid data** or **incomplete information.**

**6.3 SignPage.png**

* Captures the **login (sign-in) page** of the JPetStore application, where users enter their credentials.

**6.4 Page after Bird added to cart.png**

* Captures the **shopping cart page** after successfully adding a **Bird** item from the catalog to the cart.

**6.5 Page after Cat added to cart.png**

* Captures the cart page after the user adds a Cat item to their shopping cart.
  1. **Page after Dog added to cart.png**
* Captures the cart page after the user **updates item quantities** or **removes items** from their cart.

**6.7 Page after updating of cart.png**

* Captures the cart page after the user **updates item quantities** or **removes items** from their cart.

**6.8 Page after Submission of Order.png**

* Captures the confirmation page that appears once a user has successfully placed an order.

**7. Folder: Name 🡪 xml**

**I have created the** xml **folder** in my CapstoneProject directory to organize and store all the **TestNG configuration files**. This folder helps in managing test suite files in one place, making the project structure clean and easy to maintain.

**7.1 File Name 🡪 testing.xml**

**7.1.1 Run Multiple Test Cases Together**

* I can add different test classes like LoginPageTest, RegisterPageTest, AddToCartPageTest, UpdateCartPage, CheckOutPage in one place.
* All these test classes will run in the sequence defined in the XML file.
  + 1. **Parameterization**
* I can pass parameters like the browser type (Chrome, Firefox) through testng.xml and execute the same tests on different browsers.

**7.1.3 Parallel Execution**

* It allows me to run tests parallelly across multiple threads or browsers to save time.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE suite SYSTEM "https://testng.org/testng-1.0.dtd">

<suite name="Suite">

<test thread-count="5" name="Test">

<parameter name="browser" value="firefox"></parameter>

<!-- <parameter name="url" value="url-value"></parameter>-->

<classes>

<class name="TestCases.RegisterPageTest"/>

<class name="MyRunner.LoginStepsRunnerTestng"/>

<class name="TestCases.AddToCartPageTest"/>

<class name="TestCases.UpdateCartTest"/>

<class name="TestCases.CheckOutPageTest"/>

</classes>

</test> <!-- Test -->

</suite> <!-- Suite -->

**8. File: Name 🡪 JPetStoreProperties.properties**

* I have created a JPetStoreProperties.properties file to store all the essential configuration and test data required for the automation framework.
* It includes application URLs, valid user credentials, registration details, and shipping information used across different test scenarios. This centralized file makes it easy to manage and update test data without changing the test scripts.

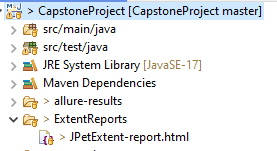
**9. Excel File: Name 🡪 JPetShippingDetails.xlsx**

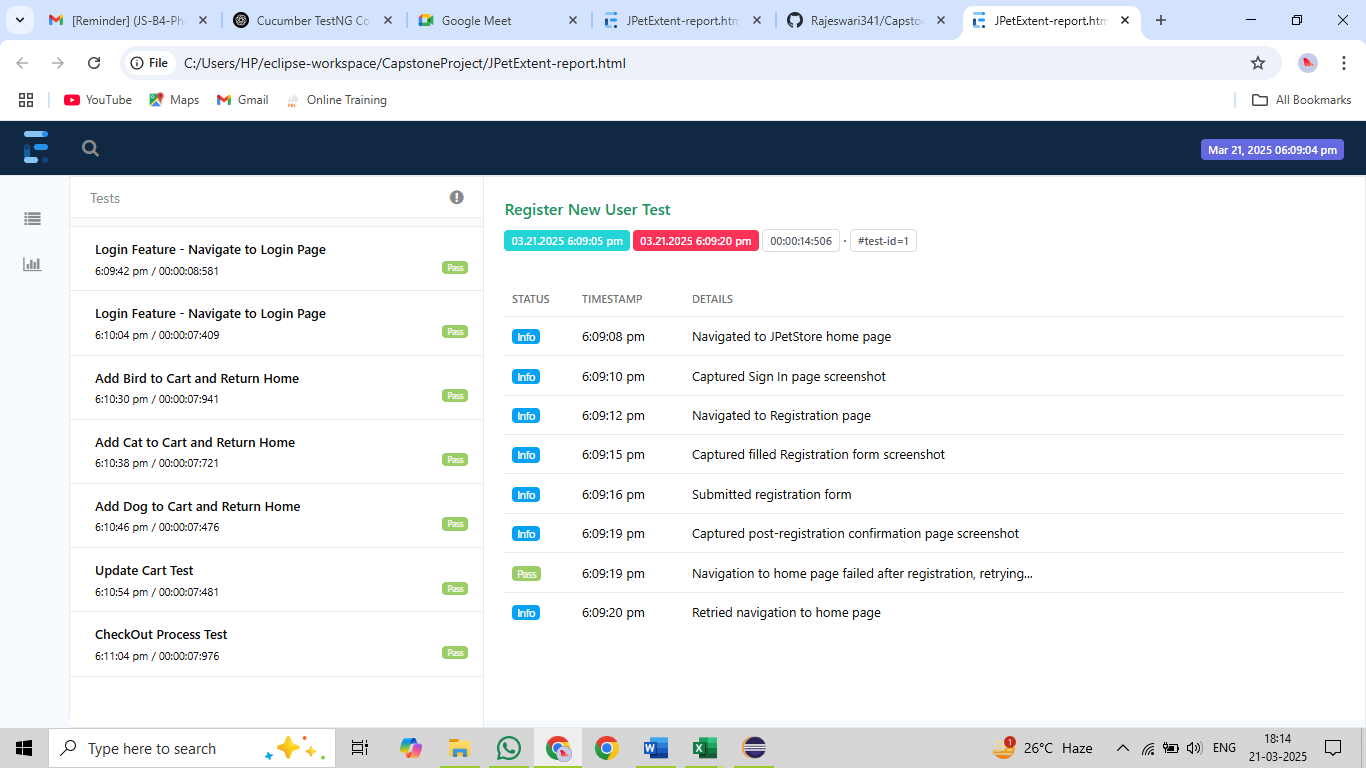
* In the Checkout Process of the JPetStore Selenium Automation Framework, JPetShippingDetails.xlsx provides shipping address details required for placing an order. These details simulate the information a real customer would provide while ordering a product.
* The Shipping Details are externally maintained in the Excel file to support data-driventesting and allow for easy updates and maintenance.

### **9. JPetExtent-report.html**

* 1. **Folder: Name** 🡪 **ExtentReports**

I have created the ExtentReports folder contains the **HTML reports** generated after executing the automated test cases. These reports are produced using the **ExtentReports** library, which provides detailed, interactive, and visually rich test execution results.

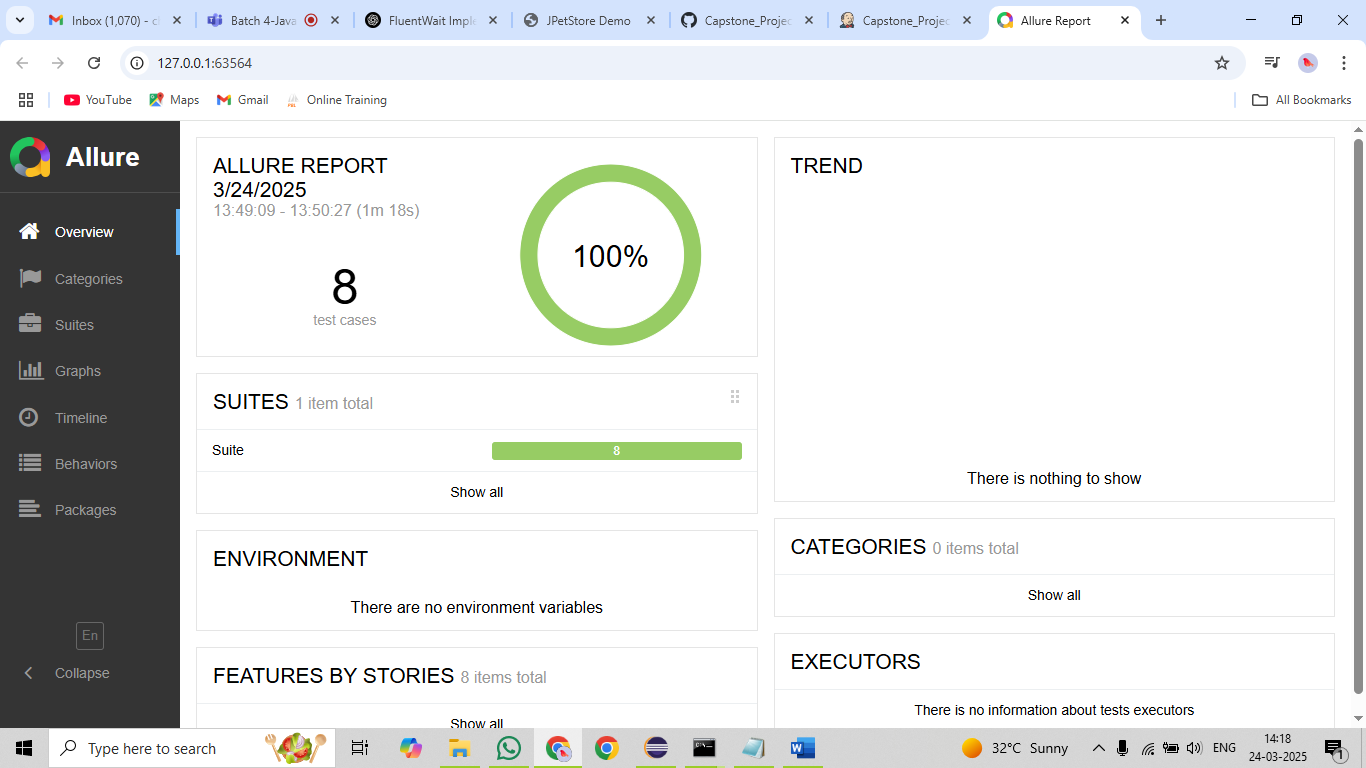
* **JPETExtent-report.html**:  
  This is the main Extent report file. It includes:
  + Summary of all test cases (Pass/Fail/Skip).
  + Logs and steps captured during test execution.
  + Screenshots attached (if implemented) for better analysis.
  + ****Detailed information like execution time, browser used, and step-wise results.

** Fig: Screenshot of the Extent Report**

**10.Allure Report**

The allure-results folder contains **raw test execution data files** used to generate Allure Reports. These files are created after running your automated test cases with **Allure reporting integration** in your Maven-based Selenium TestNG framework.  
🡪 These files describe the **structure of test suites** and **test cases.** They represent how tests are organized and linked together in the report.  
🡪 These files hold **actual test results,** including:

* + Status (Passed/Failed/Skipped)
  + Execution time
  + Error messages (if any)
  + Links to attachments like screenshots or logs

****

**Fig: Screenshot of the Allure Report**

**11. pom.xml**

The pom.xml file is the **backbone of my Maven project**.

* Manages all the **dependencies** required for the project like Selenium, TestNG, Cucumber, ExtentReports, etc.
* Handles **build lifecycle, compilation**, and **execution** tasks.
* Defines **plugins** (if any), for example, Surefire for running TestNG suite.
* It helps me **automatically download and manage** library versions without adding jar files manually.
* I can easily **share** my project, and others just need to run **mvn** install to get everything working.
* It ensures **consistent builds,** making the project scalable and easy to maintain.

## **12. GitHub**

## **12.1 Steps to Push Eclipse Project into GitHub**

### **Step 1: Initialize Git Repository in Eclipse**

1. Right-click on your project (e.g., CapstoneProject) in the **Project Explorer**.
2. Navigate to:  
   Team 🡪 Share Project...
3. In the dialog box, select **Git** and click **Next.**
4. If you have an existing Git repository:
   * Select it from the list.  
     **OR**
   * Click **Create** to make a new repository inside the project directory.
5. Click **Finish.**

### **Step 2: Create a GitHub Repository**

1. Go to [GitHub](https://github.com).
2. Click on **New Repository.**
3. Enter the repository name (for example, CapstoneProject).
4. Click **Create Repository.**  
   we can now see the **remote repository URL** (e.g., https://github.com/yourusername/CapstoneProject.git

### **Step 3: Add GitHub Remote Repository in Eclipse**

1. In Eclipse, go to **Git Repositories** view:  
   Window 🡪 Show View 🡪 Other... 🡪 Git 🡪 Git Repositories
2. Right-click on **Remotes** under your local repo 🡪 Create Remote.
3. Name it origin 🡪 Click **OK.**
4. In **Configure Push:**
   * Click **Change...** next to URI.
   * Enter the **GitHub remote URL** (https://github.com/yourusername/CapstoneProject.git).
   * Enter your **GitHub username** and **personal access token (PAT)** (instead of a password).
   * Click **Next**, then **Finish.**

### **Step 4: Add Files to Git & Commit Changes**

1. Right-click on your project 🡪 Team 🡪 Commit....
2. Select the files to be committed or click **Select All.**
3. Write a **commit message** (e.g., "Initial Commit").
4. Click **Commit and Push....**

### **Step 5: Push Project to GitHub**

1. After clicking **Commit and Push**, the **Push Wizard** opens.
2. Ensure the **branch** is master or main.
3. Click **Next,** then **Finish.**
4. Eclipse pushes the project to GitHub.

## **13.Jenkins**

## **13.1 Integrating Jenkins with a Local Project (Direct File Selection, Without Git)**

### **Step 1: Create a Freestyle Project in Jenkins**

1. Go to **Jenkins Dashboard** → Click **New Item.**
2. Enter a project name (e.g., CapstoneProject).
3. Select **Freestyle Project** → Click **OK.**

### **Step 2: Add an Advanced**

1. Click on Advanced.
2. Click on the use custom workspace.
3. Go to the eclipse and copy the path of the project.
4. Paste in the place of Directory.
5. Under **Source Code Management**, select **None.**

**Step 3: Add a Build Step for Your Local Project**

1. Under **Build** → Click **Add Build Step** → Select **Invoke top-level Maven targets.**
2. Select MAVEN\_HOME and the select the enter “test” in the Goals field.

### **Step 4: Post-build Actions**

* we can publish reports like testing, html.

### **Step 5: Save and Build**

1. Click **Save.**
2. Go to the job → Click **Build Now.**
3. Check **Console Output** for logs.

## **13.2** **Integrating Jenkins with Git (Local Git / GitHub)**

### **Step 1: Create a Freestyle Project for Git**

1. Jenkins Dashboard → **New Item.**
2. Enter a job name → Select **Freestyle Project → OK.**

### **Step 2: Configure Source Code Management**

1. Under **Source Code Management,** select **Git.**
2. **Repository URL** (for local repo):

* file:///C:/Users/HP/eclipse-workspace/CapstoneProject/.git

1. Add **Credentials** (GitHub username and PAT).

**Step 3: Add a Build Step for Your Local Project**

1. Under **Build** → Click **Add Build Step** → Select **Invoke top-level Maven targets.**
2. Select MAVEN\_HOME and the select the enter “test” in the Goals field**.**

**Step 7: Add Post-build Actions (Optional)**

* we can publish reports like testing, html.

### **Step 8: Save and Build**

1. Save the job.
2. Click **Build Now** to trigger manually.
3. Check **Console Output** to verify the process.

**14. Synchronization**

I have used three types of synchronization methods in my project: Thread.sleep(), Implicit Wait, and Explicit Wait.

* **Thread.sleep():** It pauses the execution for a fixed amount of time, specified in milliseconds, but it’s not recommended as it makes the script wait unnecessarily even if the element is ready.
* **Implicit Wait:** It tells WebDriver to wait for a certain amount of time when trying to find an element if it’s not immediately available; it's applied globally to the driver instance.
* **Explicit Wait:** It allows you to wait for specific conditions to occur (like visibility or clickability of an element) before proceeding, making it more efficient and reliable than Thread.sleep().

**15. Conclusion**

In this project, I have successfully implemented automation testing for the **JPetStore** web application using Selenium WebDriver, TestNG, Cucumber, and Maven. The framework follows the Page Object Model (POM) design pattern for better code reusability and maintainability. Different testing scenarios, such as user registration, login, add to cart, checkout, and update cart functionalities, were automated and validated. Synchronization techniques like Thread.sleep(), Implicit Wait, and Explicit Wait were applied to handle dynamic web elements efficiently. Extent Reports were used to generate detailed test execution reports, and screenshots were captured for better analysis. This project demonstrates the capability to build an efficient and scalable automation framework for e-commerce applications.