Building a Selenium Framework with Object Chaining: A Step-by-Step Guide

In this post, we'll walk through the concepts of **Object Chaining** in **Selenium Java**, building a **simple test automation framework**, and handling **WebDriver interactions** across different web pages. This guide is perfect for **beginners** who are new to Selenium, and it will also help you structure your framework in a modular way for better maintainability.

---

1. What is Object Chaining in Selenium?

Object Chaining refers to calling multiple methods on the same object in a single statement. In Selenium, this can be particularly useful for interacting with web elements in a fluent way. Instead of writing multiple statements for interacting with elements, we can chain method calls, making the code more concise and readable.

Use Case: Let's consider a login process. We can chain actions like entering a username, password, and clicking the login button in a single line.

Code Example:

```
public class LoginPage {
    private WebDriver driver;

    // Locators for the elements on the login page
    private By usernameField = By.id("username");
    private By passwordField = By.id("password");
    private By loginButton = By.id("loginButton");

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    // Method to enter username
    public LoginPage enterUsername(String username) {
        driver.findElement(usernameField).sendKeys(username); // Locate and
input the username
        return this; // Return current instance for chaining
    }

    // Method to enter password
```

```
    public LoginPage enterPassword(String password) {
        driver.findElement(passwordField).sendKeys(password); // Locate and
input the password
        return this; // Return current instance for chaining
    }

    // Method to click login button
    public HomePage clickLoginButton() {
        driver.findElement(loginButton).click(); // Click the login button
        return new HomePage(driver); // Navigate to HomePage after login
    }

    // Method to log in with username and password and navigate to HomePage
    public HomePage login(String username, String password) {
        return enterUsername(username) // Enter the username
            .enterPassword(password) // Enter the password
            .clickLoginButton(); // Click login and return the HomePage
    }
}
```

Code Explanation:

- **Line 1:** Declaring the LoginPage class.
- **Line 2:** Declaring a private WebDriver variable that will interact with the web browser.
- **Lines 4-6:** Defining By locators for the username, password fields, and the login button.
- **Line 8:** Constructor for the LoginPage class that initializes the WebDriver object.
- **Lines 11-13:** Method enterUsername inputs the username into the username field and returns the LoginPage object for method chaining.
- **Lines 15-17:** Method enterPassword inputs the password into the password field and returns the LoginPage object.
- **Lines 19-21:** Method clickLoginButton clicks the login button and returns the HomePage object, indicating a successful login.
- **Lines 23-25:** login method chains the three actions (enter username, enter password, and click login) and navigates to the HomePage.

2. Handling WebDriver Interactions

In addition to element interactions, Selenium also requires handling various complex web interactions such as **popups**, **dropdowns**, and others.

Handling Popups:

Popups can be handled using the Alert interface in Selenium. Here's an example of

```
handling a JavaScript alert popup:
public class AlertPage {
    private WebDriver driver;

    public AlertPage(WebDriver driver) {
        this.driver = driver;
    }

    // Method to accept a JavaScript alert popup
    public void acceptAlert() {
        driver.switchTo().alert().accept(); // Switch to the alert and
accept it
    }

    // Method to dismiss a JavaScript alert popup
    public void dismissAlert() {
        driver.switchTo().alert().dismiss(); // Switch to the alert and
dismiss it
    }
}
```

Code Explanation:

- **Line 1:** Declaring the AlertPage class.
- **Line 2:** Declaring a private WebDriver variable to interact with the browser.
- **Lines 4-6:** Constructor that initializes the WebDriver.
- **Line 9:** acceptAlert method that switches to the active alert and clicks "Accept".
- **Line 12:** dismissAlert method that switches to the active alert and clicks "Dismiss".

Handling Dropdowns:

Dropdowns are typically handled using the Select class in Selenium. Here's an example of selecting an option from a dropdown:

```
import org.openqa.selenium.support.ui.Select;

public class DropdownPage {
    private WebDriver driver;
```

```
    private By dropdown = By.id("dropdown");

    public DropdownPage(WebDriver driver) {
        this.driver = driver;
    }

    // Method to select an option from dropdown by visible text
    public void selectOption(String optionText) {
        Select select = new Select(driver.findElement(dropdown)); // Create
a Select object
        select.selectByVisibleText(optionText); // Select the option by its
visible text
    }
}
```

Code Explanation:

- **Line 1:** Importing the Select class to handle dropdowns.
- **Line 3:** Declaring the DropdownPage class.
- **Line 4:** Declaring the WebDriver instance to interact with the browser.
- **Line 5:** Locator for the dropdown element.
- **Lines 8-10:** Constructor that initializes the WebDriver.
- **Lines 12-14:** selectOption method that locates the dropdown and selects an option based on visible text.

---

3. Building a Simple Selenium Framework

Let's now move towards building a **small framework** using Selenium. Our framework will follow the **Page Object Model (POM)** pattern, where each page of the application is represented by a separate Java class, and tests are organized in a structured manner.

Project Structure:

```
selenium-framework/
├── src/
│   ├── main/
│   │   └── java/
│   │       └── framework/
│   │           ├── base/
│   │           │   └── BaseTest.java
│   │           ├── pages/
│   │           │   └── LoginPage.java
│   │           │   └── HomePage.java
│   │           ├── utils/
│   │               └── WebDriverUtils.java
│   └── test/
│       └── java/
│           └── test/
│               └── LoginTest.java
└── pom.xml
```

BaseTest Class:

The BaseTest class will handle **WebDriver initialization** and **teardown**.

```
package framework.base;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import io.github.bonigarcia.wdm.WebDriverManager;

public class BaseTest {
    protected WebDriver driver;
```

```
    // Initialize WebDriver and navigate to a URL
    public void setup(String url) {
        WebDriverManager.chromedriver().setup(); // Setup ChromeDriver with
WebDriverManager
        driver = new ChromeDriver(); // Initialize ChromeDriver
        driver.get(url); // Open the URL in the browser
    }

    // Close WebDriver after the test
    public void teardown() {
        if (driver != null) {
            driver.quit(); // Close the browser
        }
    }
}
```

Code Explanation:

- **Line 1:** Declaring the BaseTest class.
- **Line 2:** Importing necessary Selenium and WebDriverManager classes.
- **Line 6:** Declaring the driver instance that will interact with the browser.
- **Line 9:** setup method to initialize the WebDriver and navigate to the provided URL.
- **Line 10:** WebDriverManager automatically manages the ChromeDriver version for you.
- **Line 12:** teardown method to quit the browser session after tests are completed.

LoginPage Class (Page Object):

Represents the login page where we enter credentials and perform the login action.

```
package framework.pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class LoginPage {

    private WebDriver driver;

    private By usernameField = By.id("username");
```

```
    private By passwordField = By.id("password");
    private By loginButton = By.id("loginButton");

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public LoginPage enterUsername(String username) {
        driver.findElement(usernameField).sendKeys(username); // Enter the
username in the field
        return this; // Return current instance for chaining
    }

    public LoginPage enterPassword(String password) {
        driver.findElement(passwordField).sendKeys(password); // Enter the
password in the field
        return this; // Return current instance for chaining
    }

    public HomePage clickLoginButton() {
        driver.findElement(loginButton).click(); // Click the login button
        return new HomePage(driver); // Navigate to HomePage after login
    }

    public HomePage login(String username, String password) {
        return enterUsername(username) // Enter the username
            .enterPassword(password) // Enter the password
            .clickLoginButton(); // Click login and return the HomePage
    }
}
```

Code Explanation:

- **Lines 1-3:** Define the LoginPage class and declare WebDriver and locators.
- **Line 6:** Constructor to initialize the WebDriver instance.
- **Lines 9-14:** Methods for entering the username, password, and clicking the login button.
- **Line 17:** login method chains the actions for a quick login process and returns the HomePage.

Test Class:

Finally, the **LoginTest** class will use the LoginPage and HomePage objects to test the login functionality.

```
package test;

import framework.base.BaseTest;
import framework.pages.LoginPage;
import framework.pages.HomePage;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class LoginTest extends BaseTest {

    private LoginPage loginPage;
    private HomePage homePage;

    @BeforeEach
    public void setUp() {
        super.setup("https://example.com/login"); // Set up WebDriver and
open the login page
        loginPage = new LoginPage(driver); // Initialize the LoginPage
object
    }

    @Test
    public void testLogin() {
        homePage = loginPage.login("testUser", "password123"); // Perform
login
        String greeting = homePage.getUserGreeting(); // Get the greeting
message from HomePage
        System.out.println("Greeting: " + greeting); // Print the greeting
    }

    @AfterEach
    public void tearDown() {
        super.teardown(); // Close the browser session after the test
    }
}
```

Code Explanation:

- **Lines 1-3:** Import necessary classes for the test.
- **Lines 6-8:** @BeforeEach sets up the WebDriver, opens the login page, and initializes the LoginPage object.
- **Lines 10-13:** @Test method that performs the login action and retrieves the greeting from the HomePage.
- **Lines 15-17:** @AfterEach ensures that the browser session is closed after the test.

---

4. Using the Framework for Different Web Pages

To extend this framework to different pages, you can create more **Page Object Classes** for other parts of your application (e.g., ProfilePage, SettingsPage). You can use the same WebDriver instance to navigate between pages using methods that return the appropriate page object after performing actions.

For example, if you have a **ProfilePage** after logging in:

```
package framework.pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class ProfilePage {

    private WebDriver driver;
    private By profileName = By.id("profileName");

    public ProfilePage(WebDriver driver) {
        this.driver = driver;
    }

    public String getProfileName() {
        return driver.findElement(profileName).getText(); // Retrieve
profile name text
    }
}
```

You can modify the LoginPage class to return ProfilePage after a successful login:

```
public ProfilePage login(String username, String password) {
    return enterUsername(username)
        .enterPassword(password)
        .clickLoginButton();
}
```

This approach allows you to easily handle navigation between different pages, keeping your tests **modular** and **scalable**.

Conclusion

This post provides a **complete guide** on setting up a **Selenium framework** with **object chaining**, **WebDriver interactions**, and **Page Object Model**. It also explains how to extend the framework to work across multiple pages in your application. This approach ensures **scalability** and **maintainability** as your test suite grows.

Start building your automation framework with these principles, and as you advance, you'll be able to add more complex interactions and handle various web elements seamlessly.

Feel free to share this post and help others get started with Selenium test automation!