**Selenium**

Swapni Vaidya   
octo /nov2022

**Q1)** Write a code for Mouse Hover, click on Enter using action class, JavaScript Executot code for Hidden Element (click and send-keys), switching window using window handles, and Type of waits in Selenium (all with Syntax).

import org.openqa.selenium.By;

import org.openqa.selenium.JavascriptExecutor;

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.chrome.ChromeDriver;

import org.openqa.selenium.interactions.Actions;

import org.openqa.selenium.support.ui.ExpectedConditions;

import org.openqa.selenium.support.ui.WebDriverWait;

public class Main {

public static void main(String[] args) {

// Set the driver path

System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");

// Create a new instance of the Chrome driver

WebDriver driver = new ChromeDriver();

// Navigate to a URL

driver.get("https://www.example.com");

// Mouse Hover

Actions actions = new Actions(driver);

WebElement element = driver.findElement(By.xpath("//a[@href='/hover']"));

actions.moveToElement(element).build().perform();

// Click on Enter

actions.sendKeys(Keys.RETURN).build().perform();

// Hidden Element (click and send-keys)

JavascriptExecutor js = (JavascriptExecutor) driver;

WebElement hiddenElement = driver.findElement(By.id("hidden"));

js.executeScript("arguments[0].click();", hiddenElement);

hiddenElement.sendKeys("text");

// Switching window using window handles

String parentWindowHandle = driver.getWindowHandle();

driver.findElement(By.xpath("//a[@href='/newWindow']")).click();

for (String windowHandle : driver.getWindowHandles()) {

if (!windowHandle.equals(parentWindowHandle)) {

driver.switchTo().window(windowHandle);

}

}

// Type of waits in Selenium

// Implicit Wait

driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);

// Explicit Wait

WebDriverWait wait = new WebDriverWait(driver, 10);

WebElement dynamicElement = wait.until(ExpectedConditions.presenceOfElementLocated(By.id("dynamic")));

dynamicElement.click();

// Fluent Wait

Wait<WebDriver> fluentWait = new FluentWait<>(driver)

.withTimeout(Duration.ofSeconds(30))

.pollingEvery(Duration.ofSeconds(5))

.ignoring(NoSuchElementException.class);

WebElement fluentElement = fluentWait.until(driver -> driver.findElement(By.id("fluent")));

fluentElement.click();

// Close the driver

driver.quit();

}

}

Q2) Write a code for Read and Write Excel.

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import org.apache.poi.ss.usermodel.Cell;

import org.apache.poi.ss.usermodel.Row;

import org.apache.poi.ss.usermodel.Sheet;

import org.apache.poi.ss.usermodel.Workbook;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

public class Main {

public static void main(String[] args) throws IOException {

// Read from an Excel file

FileInputStream inputStream = new FileInputStream("path/to/excel.xlsx");

Workbook workbook = new XSSFWorkbook(inputStream);

Sheet sheet = workbook.getSheetAt(0);

for (Row row : sheet) {

for (Cell cell : row) {

System.out.print(cell.getStringCellValue() + " ");

}

System.out.println();

}

inputStream.close();

// Write to an Excel file

FileOutputStream outputStream = new FileOutputStream("path/to/excel.xlsx");

Sheet sheet = workbook.createSheet();

Row row = sheet.createRow(0);

Cell cell = row.createCell(0);

cell.setCellValue("Hello World");

workbook.write(outputStream);

outputStream.close();

}

}

You need to have Apache POI library added to your classpath to be able to run this code.

Q3) In TestNg: what is a priority, hook, Assertion, Type Of assertion, and Annotation (all with Syntax)?

Priority: In TestNG, we can assign a priority to each test method. The priority determines the order in which the test methods are executed. The lower the priority number, the higher the priority of the test method.

@Test(priority = n)

public void testMethod() {

// code to run the test

}

where 'n' is an integer that determines the priority of the test method.

Hook: Hooks are methods that are executed before and after each test method. TestNG provides the following hooks.

@BeforeMethod: This method is executed before each test method.

@AfterMethod: This method is executed after each test method.

@BeforeMethod

public void beforeMethod() {

// code to run before each test method

}

@AfterMethod

public void afterMethod() {

// code to run after each test method

}

Assertion: Assertions are used to verify the expected behavior of the code. In TestNG, we can use the org.testng.Assert class to perform assertions.

import org.testng.Assert;

@Test

public void testMethod() {

// code to run the test

Assert.assertEquals(actualValue, expectedValue);

}

Type of Assertions: There are various types of assertions in TestNG. Some common assertions are:

assertEquals(actual, expected): This assertion verifies that the actual value is equal to the expected value.

assertTrue(condition): This assertion verifies that the condition is true.

assertFalse(condition): This assertion verifies that the condition is false.

Annotation: Annotation is a special type of tag in TestNG that provides metadata information about the test methods. TestNG supports a variety of annotations, including:

@Test: This annotation marks a method as a test method.

@BeforeSuite: This method is executed before the entire suite.

@AfterSuite: This method is executed after the entire suite.

@BeforeTest: This method is executed before each <test> tag in the testng.xml file.

@AfterTest: This method is executed after each <test> tag in the testng.xml file

import org.testng.annotations.Test;

@Test

public void testMethod() {

// code to run the test

}

Q4) In Framework: how to pass data using data provider, Type of TestNg Listerner, Retry logic on test case failure(all with Syntax).

Data Provider: In TestNG, we can pass data to test methods using the @DataProvider annotation. The data provider method returns an object array that contains the data for the test method.

import org.testng.annotations.DataProvider;

import org.testng.annotations.Test;

@DataProvider

public Object[][] dataProviderMethod() {

return new Object[][] {

{ "data1", "data2" },

{ "data3", "data4" }

};

}

@Test(dataProvider = "dataProviderMethod")

public void testMethod(String data1, String data2) {

// code to run the test with data

}

Type of TestNG Listeners: TestNG listeners are used to extend the functionality of TestNG. TestNG supports the following types of listeners:

ITestListener: This listener interface is used to listen to the events that occur during the test run.

IInvokedMethodListener: This listener interface is used to listen to the events that occur during the test method invocations.

IAnnotationTransformer: This listener interface is used to listen to the events that occur during the processing of annotations.

import org.testng.ITestListener;

import org.testng.ITestResult;

public class TestListener implements ITestListener {

@Override

public void onTestStart(ITestResult result) {

// code to run when a test starts

}

@Override

public void onTestSuccess(ITestResult result) {

// code to run when a test is successful

}

// ... other listener methods

}

Retry Logic on Test Case Failure: In TestNG, we can retry a test method if it fails. To implement the retry logic, we can use the retryAnalyzer attribute of the @Test annotation.

import org.testng.annotations.Test;

@Test(retryAnalyzer = RetryAnalyzer.class)

public void testMethod() {

// code to run the test

}

import org.testng.IRetryAnalyzer;

import org.testng.ITestResult;

public class RetryAnalyzer implements IRetryAnalyzer {

private int retryCount = 0;

private int maxRetryCount = 3;

@Override

public boolean retry(ITestResult result) {

if (retryCount < maxRetryCount) {

retryCount++;

return true;

}

return false;

}

}

Q5) Create a small framework using pom cucumber and perform login functionality

First, create a Maven project in an IDE such as Eclipse.

Add the necessary dependencies in the pom.xml file:

<dependencies>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>6.0.0</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>6.0.0</version>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

</dependency>

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>3.14.0</version>

</dependency>

</dependencies>

**Create the following directories:**

src/test/java/stepDefinitions: This is where you will define your Cucumber step definitions

src/test/java/pageObjects: This is where you will define your Page Object Model classes

src/test/resources/features: This is where you will store your feature files

for example:

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.WebElement;

import org.openqa.selenium.support.FindBy;

import org.openqa.selenium.support.PageFactory;

public class LoginPage {

WebDriver driver;

@FindBy(id = "username")

private WebElement username;

@FindBy(id = "password")

private WebElement password;

@FindBy(id = "submit")

private WebElement submit;

public LoginPage(WebDriver driver) {

this.driver = driver;

PageFactory.initElements(driver, this);

}

public void enterUsername(String name) {

username.sendKeys(name);

}

public void enterPassword(String pass) {

password.sendKeys(pass);

}

public void clickSubmit() {

submit.click();

}

}

Create a feature file for the login functionality in the features directory.

Feature: Login functionality

Scenario: Successful login

Given I navigate to the login page

When I enter the username and password

Then I should be logged in successfully

Create the step definition class for the feature file,

import org.openqa.selenium.WebDriver;

import org.openqa.selenium.chrome.ChromeDriver;

import cucumber.api.java.en.Given;

import cucumber.api.java.en.Then;

import cucumber.api.java.