# Automation Framework using Selenium, Cucumber & Java

Step 1: Create a Maven project in an IDE (Intellij / Eclipse).

Step 2: Add the necessary dependencies.

Here is a list of dependencies commonly required for an Automation Framework using Selenium, Cucumber, and Java:

1. **Selenium WebDriver**:
   * org.seleniumhq.selenium:selenium-java
2. **Cucumber**:
   * io.cucumber:cucumber-core
   * io.cucumber:cucumber-java
   * io.cucumber:cucumber-junit
   * io.cucumber:cucumber-html
   * io.cucumber:cucumber-jvm-deps
   * io.cucumber:cucumber-reporting
   * io.cucumber:gherkin
3. **JUnit** (for running tests):
   * junit:junit
4. **Hamcrest** (for assertions):
   * org.hamcrest:hamcrest-core
5. **Cobertura** (for code coverage, optional):
   * net.sourceforge.cobertura:cobertura

These dependencies can be added to your pom.xml if you are using Maven. Here's an example pom.xml configuration:

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>SCJ\_Framework</groupId>

<artifactId>SCJ\_Framework</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.2.5</version>

<configuration>

<testFailureIgnore>true</testFailureIgnore>

</configuration>

</plugin>

<plugin>

<groupId>net.masterthought</groupId>

<artifactId>maven-cucumber-reporting</artifactId>

<version>5.7.6</version>

<executions>

<execution>

<id>execution</id>

<phase>verify</phase>

<goals>

<goal>generate</goal>

</goals>

<configuration>

<projectName>SCJ\_Framework</projectName>

<outputDirectory>${project.build.directory}/cucumber-report-html</outputDirectory>

<cucumberOutput>${project.build.directory}/cucumber.json</cucumberOutput>

</configuration>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>17</source>

<target>17</target>

</configuration>

</plugin>

</plugins>

</build>

<dependencies>

<dependency>

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

<artifactId>selenium-java</artifactId>

<version>4.14.1</version>

</dependency>

<dependency>

<groupId>net.masterthought</groupId>

<artifactId>maven-cucumber-reporting</artifactId>

<version>5.7.6</version>

</dependency>

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-picocontainer</artifactId>

<version>1.2.6</version>

</dependency>

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-java</artifactId>

<version>1.2.6</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>info.cukes</groupId>

<artifactId>cucumber-junit</artifactId>

<version>1.2.6</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.16.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.pageobject</groupId>

<artifactId>core\_2.12.0-M5</artifactId>

<version>0.1.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-api</artifactId>

<version>3.0.0-beta2</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>3.0.0-beta2</version>

</dependency>

<!-Hamcrest -->

<dependency>

<groupId>org.hamcrest</groupId>

<artifactId>hamcrest-core</artifactId>

<version>2.2</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>gherkin</artifactId>

<version>28.0.0</version>

</dependency>

<!-https://mvnrepository.com/artifact/io.cucumber/cucumber-core -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-core</artifactId>

<version>7.18.0</version>

</dependency>

<!-https://mvnrepository.com/artifact/io.cucumber/cucumber-java -->

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-java</artifactId>

<version>7.18.0</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-junit</artifactId>

<version>7.13.0</version>

</dependency>

<dependency>

<groupId>io.cucumber</groupId>

<artifactId>cucumber-html</artifactId>

<version>0.2.7</version>

</dependency>

</dependencies>

</project>

Step 3: Update the pom.xml from Maven

Step 4: Create folder structure as below;

Project Name

* Features (Folder)
* pageObjects (Package)
* stepDefinitions
* utilities (Package)
* Drivers (Folder)
* Target (Reports)
* pom.xml

Step 5: Copy drivers to Drivers folder (Ex; Chrome Driver)

Step 6: Automating Login test case

* 1. Create a **Login.feature** under features folder.
  2. Create a page object class for LoginPage.java under **pageObjects** package.
  3. Create stepsDef.java under definitions under **stepDefinitions** package.
  4. Create **TestRunner.java** under **myRunner** package.

# Wait in Selenium:

In Selenium, there are three main types of waits that you can use to manage synchronization between your test scripts and the web application under test: implicit waits, explicit waits, and fluent waits. Each of these waits has its own use cases and advantages. Here’s an overview of each type:

## 1. Implicit Waits:

Implicit Wait is used to tell the WebDriver to wait for a certain amount of time before throwing a NoSuchElementException. It applies to all elements in the test script.

**Usage:** When you need to wait for elements to be present before interacting with them, but don’t want to specify conditions for each individual element.

Example:

WebDriver driver = new ChromeDriver();

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

driver.get("http://example.com");

WebElement element = driver.findElement(By.id("exampleId")); // Waits up to 10 seconds

## 2. Explicit Waits:

Explicit Wait is used to tell the WebDriver to wait for certain conditions to be met before proceeding with the next step. This type of wait is more flexible and can be applied to specific elements.

**Usage:** When you need to wait for specific conditions (e.g., element to be clickable, element to be visible).

Example:

WebDriver driver = new ChromeDriver();

driver.get("http://example.com");

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));

WebElement element = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("exampleId"))); // Waits up to 10 seconds

element.click();

## 3. Fluent Waits

Fluent Wait is a more advanced type of explicit wait that allows you to define the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition. You can also specify what exceptions to ignore.

**Usage:** When you need finer control over the wait behavior, such as polling intervals or ignoring certain exceptions.

Example:

WebDriver driver = new ChromeDriver();

driver.get("http://example.com");

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

.withTimeout(Duration.ofSeconds(30)) // Maximum time to wait

.pollingEvery(Duration.ofSeconds(5)) // Frequency to check the condition

.ignoring(NoSuchElementException.class); // Exceptions to ignore

WebElement element = wait.until(driver -> driver.findElement(By.id("exampleId")));

element.click();

**When to Use Each Type of Wait?**

**Implicit Wait:** Use it when you want to apply a default waiting time for all elements in the script. It’s simple and easy to use but can lead to longer wait times than necessary.

**Explicit Wait:** Use it when you need to wait for specific conditions for certain elements. It provides more control and precision compared to implicit waits.

**Fluent Wait:** Use it when you need advanced control over the wait behavior, such as customizing polling intervals or ignoring certain exceptions. It is the most flexible type of wait.

**Best Practices:**

Avoid using ***Thread.sleep()*** for waits as it leads to inefficient and brittle tests.

Use implicit waits sparingly and prefer explicit waits for better control.

Use fluent waits when dealing with complex synchronization issues that require custom polling intervals or exception handling.

By choosing the appropriate type of wait based on your specific needs, you can create more reliable and efficient Selenium test scripts.

Alternatively, we can use WebDriver Wait to wait on purpose for user defined time.

**WebDriverWait** is a part of Selenium WebDriver's Explicit Wait mechanism. It allows you to wait for a certain condition to occur before proceeding with the next step in the automation script. This is useful for handling dynamic web elements and conditions that can change over time, such as loading times for web elements.

**Key Points about WebDriverWait:**

*1. Explicit Wait:* Unlike Implicit Wait, which is applied globally, WebDriverWait is applied to specific conditions.

*2. Conditions:* You can wait for various conditions such as element visibility, element clickability, presence of an element, etc.

*3. Timeout:* You specify a maximum amount of time to wait for the condition to be met.

**Example Usage of WebDriverWait**

Here’s an example to demonstrate how WebDriverWait can be used in a Selenium test script:

Scenario: Imagine you want to test a login functionality where you need to wait for the username and password fields to be visible before entering the credentials, and then wait for the login button to be clickable before clicking it.

First, you need to import the necessary Selenium libraries:

*import org.openqa.selenium.WebDriver;*

*import org.openqa.selenium.WebElement;*

*import org.openqa.selenium.By;*

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

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

*import java.time.Duration;*

Then, you can create your step definitions:

*public class LoginSteps {*

*private WebDriver driver;*

*private WebDriverWait wait;*

*@Given("User is on the login page")*

*public void userIsOnTheLoginPage() {*

*driver = new ChromeDriver();*

*driver.get("http://example.com/login");*

*wait = new WebDriverWait(driver, Duration.ofSeconds(10));*

*}*

*@When("User enters username {string} and password {string}")*

*public void userEntersUsernameAndPassword(String username, String password) {*

*WebElement usernameField = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("username")));*

*usernameField.sendKeys(username);*

*WebElement passwordField = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("password")));*

*passwordField.sendKeys(password);*

*}*

*@When("User clicks the login button")*

*public void userClicksTheLoginButton() {*

*WebElement loginButton = wait.until(ExpectedConditions.elementToBeClickable(By.id("loginButton")));*

*loginButton.click();*

*}*

*@Then("User should see the homepage")*

*public void userShouldSeeTheHomepage() {*

*WebElement homePageElement = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("homePageElement")));*

*assertTrue("User is not on the homepage", homePageElement.isDisplayed());*

*driver.quit();*

*} }*

**Explanation:**

**1. Initialization:**

The WebDriver and WebDriverWait instances are initialized in the @Given step. WebDriverWait is set to wait for up to 10 seconds for the specified conditions.

**2. Waiting for Elements:**

Username and Password Fields: The script waits for the username and password fields to be visible using ExpectedConditions.visibilityOfElementLocated().

Login Button: The script waits for the login button to be clickable using ExpectedConditions.elementToBeClickable().

**3. Assertion:**

The script waits for an element on the homepage to be visible to confirm that the login was successful.

**Common ‘ExpectedConditions’ Methods:**

**visibilityOfElementLocated(By locator):** Waits until the specified element is visible.

**elementToBeClickable(By locator):** Waits until the specified element is clickable.

**presenceOfElementLocated(By locator):** Waits until the specified element is present in the DOM.

**textToBePresentInElement(By locator, String text):** Waits until the specified text is present in the element.

**alertIsPresent():** Waits until an alert is present.

By using WebDriverWait, you can make your test scripts more robust and reliable, handling dynamic content and varying load times effectively.

## What to do if reports are not generating?

The key part here is the `plugin` option, which specifies `json:target/cucumber.json`. This should generate a JSON report in the `target` directory.

Given that your configuration seems correct, there are a few things you should check and verify to resolve the issue:

1. \*\*Ensure Tests are Running:\*\*

- Make sure that the Cucumber tests are being executed. If `dryRun` is set to `true`, the tests will not actually run, and thus no JSON report will be generated. Set `dryRun` to `false` or remove it for actual test execution:

@CucumberOptions(

dryRun = false, // or remove this line

features = {"src/Features/Login.feature"},

monochrome = true,

glue = {"stepDef"},

tags = "@run",

plugin = {"pretty" ,

"html:target/cucumber-html-report",

"json:target/cucumber.json",

"junit:target/hc-cucumber.xml",

"rerun:target/hc-rerun.txt"}

)

2. \*\*Check File Path:\*\*

- After running your tests, verify that the JSON file is indeed created at `target/cucumber.json`. If the file is not there, there might be an issue with the test execution or the path configuration.

3. \*\*Verify Maven Surefire/Failsafe Plugin Configuration:\*\*

- If you are running the tests with Maven, ensure that the `maven-surefire-plugin` or `maven-failsafe-plugin` is configured correctly in your `pom.xml`:

```xml

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<version>3.0.0-M5</version> <!-- or the latest version -->

<configuration>

<includes>

<include>\*\*/AppTest.java</include>

</includes>

</configuration>

</plugin>

4. \*\*Run Clean Build:\*\*

- Run a clean build to ensure there are no stale files or artifacts interfering with the process:

mvn clean test

5. \*\*Check for Errors in Test Output:\*\*

- Review the console output when you run the tests to ensure there are no errors or issues preventing the JSON file from being generated.

6. \*\*Check Directory Permissions:\*\*

- Ensure that the `target` directory is writable and that there are no permission issues preventing the JSON file from being created.

If all these checks are in place and the issue persists, please provide the output of your test run and any relevant portions of your project configuration for further diagnosis.