SESSION UNDERSTANDING DOCUMENT ON AUTOMATION TESTING

TABLE OF CONTENTS:

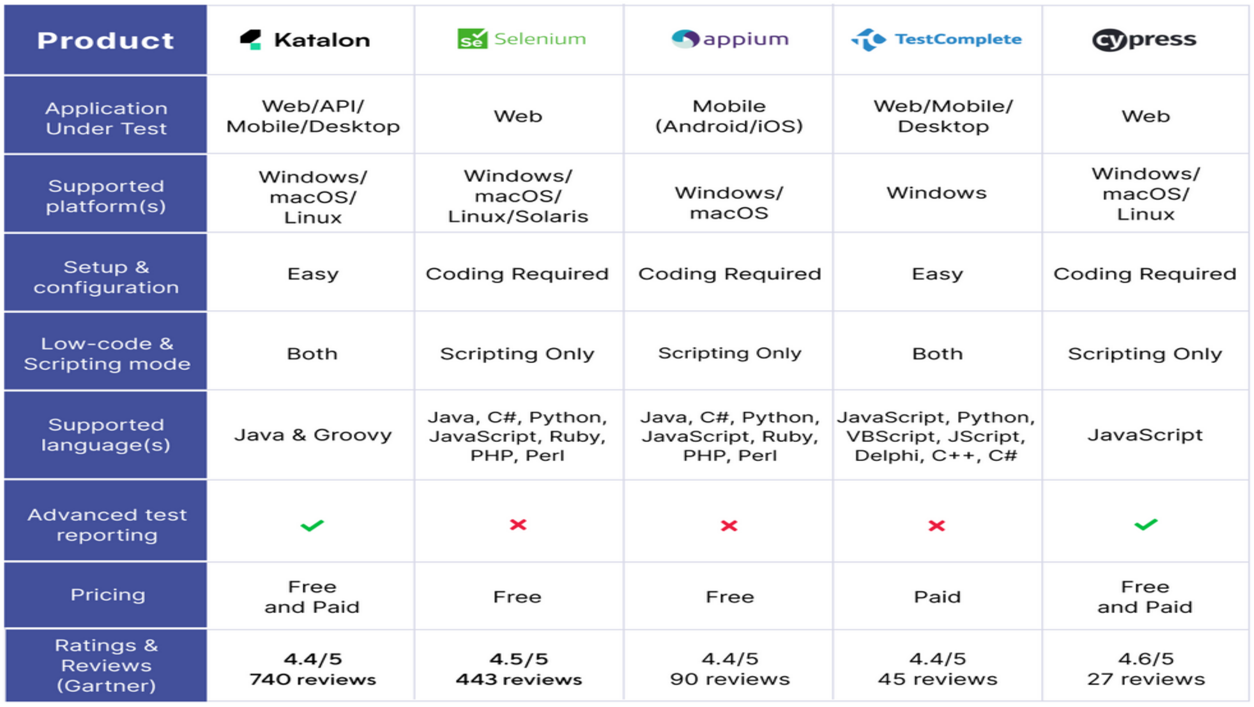
Automation Testing:

Automation testing is a software testing technique where specialized tools and scripts are used to perform predefined actions on a software application to verify its functionality. It involves creating and executing test cases automatically, without requiring manual intervention for each test step. It aims to automate repetitive, time-consuming, and labor-intensive manual testing tasks to enhance efficiency, accuracy, and effectiveness in software testing.

Advantages of automation testing

- It saves time and cost in testing and provides an increment in the efficiency of testing.
- Automation testing improves the accuracy of testing
- With automation, more cycles can be achieved
- It also ensures consistency in testing
- It's test scripts can be reusable
- Ability to cover the test application features widely
- Automation testing results are reliable
- In this testing, human intervention is not required
- Speedily executes the testing process frequently and thoroughly

Most used Automation Tools:



Product	Katalon	Selenium	appium	TestComplete	cypress
Application Under Test	Web/API/ Mobile/Desktop	Web	Mobile (Android/iOS)	Web/Mobile/ Desktop	Web
Supported platform(s)	Windows/ macOS/ Linux	Windows/ macOS/ Linux/Solaris	Windows/ macOS	Windows	Windows/ macOS/ Linux
Setup & configuration	Easy	Coding Required	Coding Required	Easy	Coding Required
Low-code & Scripting mode	Both	Scripting Only	Scripting Only	Both	Scripting Only
Supported language(s)	Java & Groovy	Java, C#, Python, JavaScript, Ruby, PHP, Perl	Java, C#, Python, JavaScript, Ruby, PHP, Perl	JavaScript, Python, VBScript, JScript, Delphi, C++, C#	JavaScript
Advanced test reporting	✓	✗	✗	✗	✓
Pricing	Free and Paid	Free	Free	Paid	Free and Paid
Ratings & Reviews (Gartner)	4.4/5 740 reviews	4.5/5 443 reviews	4.4/5 90 reviews	4.4/5 45 reviews	4.6/5 27 reviews

Selenium:

Selenium is an open-source suite of tools and libraries that is used for browser automation. Selenium us used to:

It allows users to test their websites functionally **on different browsers**.

Importance of Testing in Selenium

Manual testing ca n be time-consuming and prone to human errors. Selenium Automation allows tests to be executed quickly and accurately, reducing the likelihood of human mistakes and ensuring consistent test results.

Language Support: Selenium allows you to create test scripts in different languages like Ruby, Java, PHP, Perl, Python, JavaScript, and C#, among others.

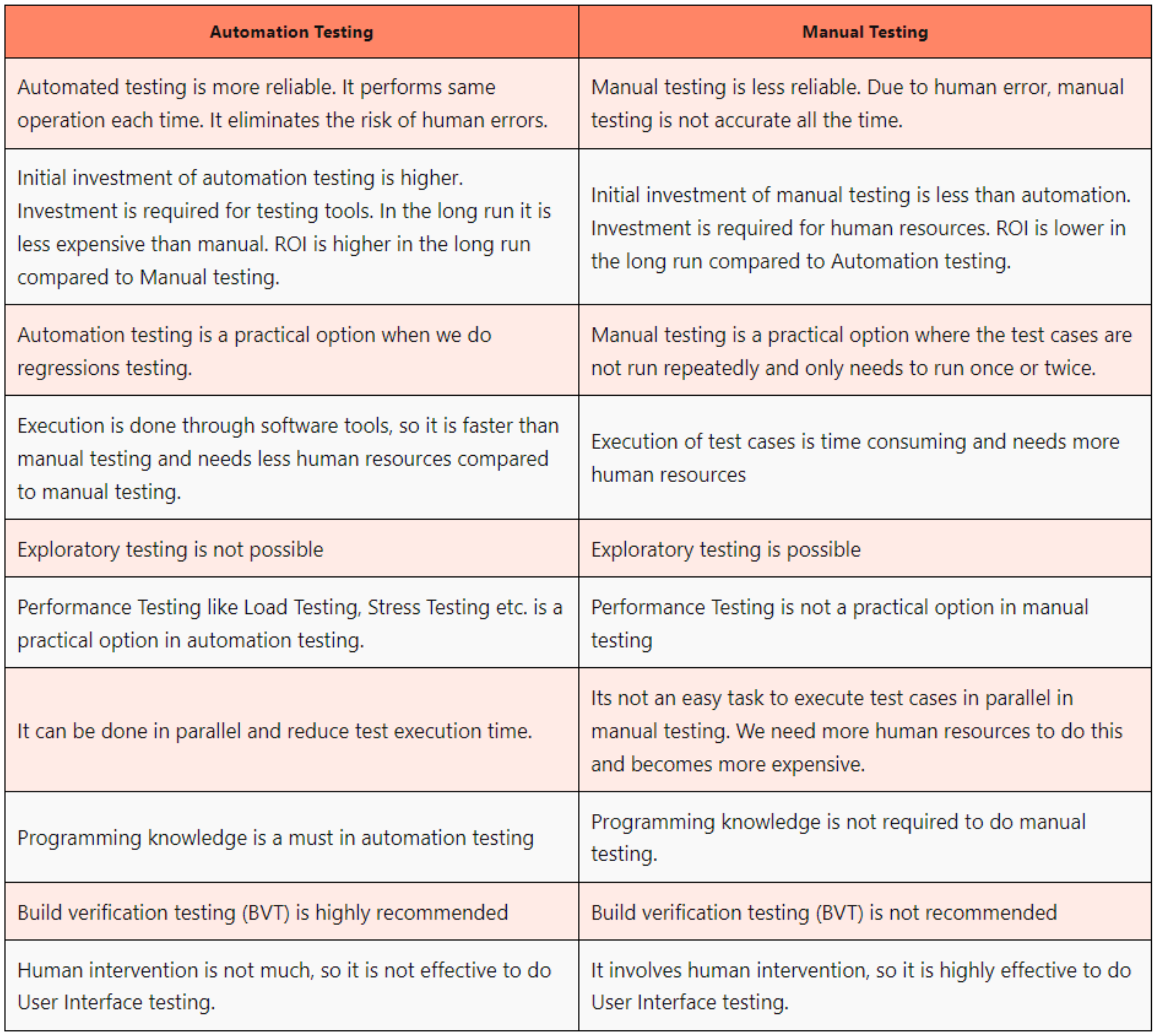
Browser Support: Selenium enables you to test your website on different browsers such as Google Chrome, Mozilla Firefox, Microsoft Edge, Safari etc.

Scalability: Automated testing with Selenium can easily scale to cover a wide range of test cases, scenarios, and user interactions. This scalability ensures maximum test coverage of the application's functionality.

Reusable Test Scripts: Selenium allows testers to create reusable test scripts that can be used across different test cases and projects. This reusability saves time and effort in test script creation and maintenance.

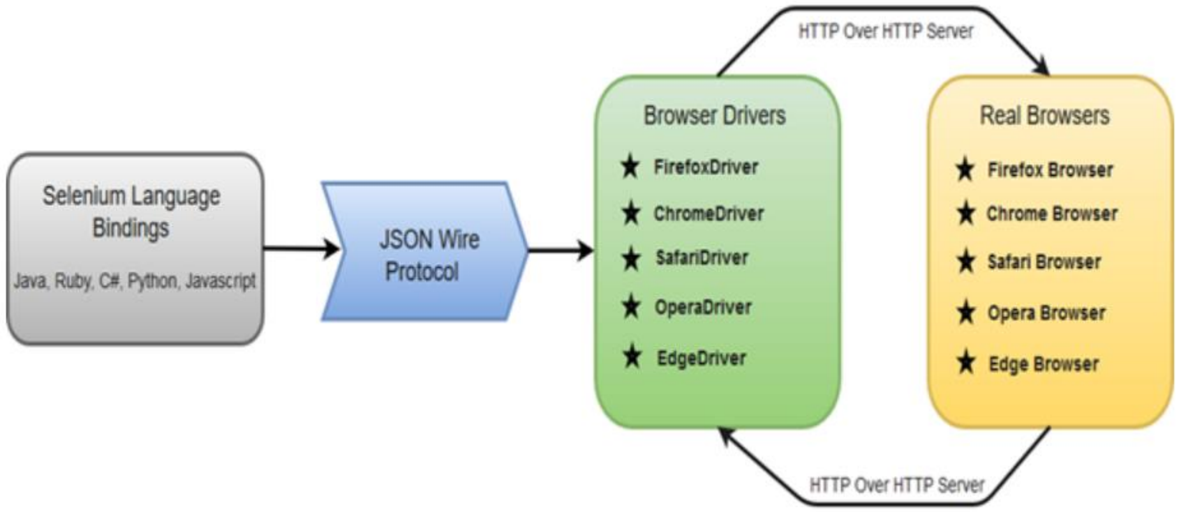
Parallel Testing: Selenium supports parallel test execution, allowing multiple tests to run concurrently. This helps reduce the overall testing time, making the development process more efficient.

Difference between Automation and Manual Testing:



Automation Testing	Manual Testing
Automated testing is more reliable. It performs same operation each time. It eliminates the risk of human errors.	Manual testing is less reliable. Due to human error, manual testing is not accurate all the time.
Initial investment of automation testing is higher. Investment is required for testing tools. In the long run it is less expensive than manual. ROI is higher in the long run compared to Manual testing.	Initial investment of manual testing is less than automation. Investment is required for human resources. ROI is lower in the long run compared to Automation testing.
Automation testing is a practical option when we do regressions testing.	Manual testing is a practical option where the test cases are not run repeatedly and only needs to run once or twice.
Execution is done through software tools, so it is faster than manual testing and needs less human resources compared to manual testing.	Execution of test cases is time consuming and needs more human resources
Exploratory testing is not possible	Exploratory testing is possible
Performance Testing like Load Testing, Stress Testing etc. is a practical option in automation testing.	Performance Testing is not a practical option in manual testing
It can be done in parallel and reduce test execution time.	Its not an easy task to execute test cases in parallel in manual testing. We need more human resources to do this and becomes more expensive.
Programming knowledge is a must in automation testing	Programming knowledge is not required to do manual testing.
Build verification testing (BVT) is highly recommended	Build verification testing (BVT) is not recommended
Human intervention is not much, so it is not effective to do User Interface testing.	It involves human intervention, so it is highly effective to do User Interface testing.

Selenium Architecture:



There are four basic components of WebDriver Architecture:

- Selenium Language Bindings
- JSON Wire Protocol
- Browser Drivers
- Real Browsers

Selenium Language Bindings / Selenium Client Libraries

Selenium developers have built language bindings/Selenium Client Libraries in order to support multiple languages. For instance, if you want to use the browser driver in java, use the java bindings. All the supported language bindings can be downloaded from the official of Selenium.

JSON Wire Protocol

JSON (JavaScript Object Notation) is an open standard for exchanging data on web. It supports data structures like object and array. So, it is easy to write and read data from JSON. JSON Wire Protocol provides a transport mechanism to transfer data between a server and a client. JSON Wire Protocol serves as an industry standard for various REST web services.

Browser Drivers

Selenium uses drivers, specific to each browser in order to establish a secure connection with the browser without revealing the internal logic of browser's functionality. The browser driver is also specific to the language used for automation such as Java, C#, etc.

When we execute a test script using WebDriver, the following operations are performed internally.

- HTTP request is generated and sent to the browser driver for each Selenium command.
- The driver receives the HTTP request through HTTP server.
- HTTP Server decides all the steps to perform instructions which are executed on browser.
- Execution status is sent back to HTTP Server which is subsequently sent back to automation script.

Browsers

Browsers supported by Selenium WebDriver:

- Internet Explorer
- Mozilla Firefox
- Google Chrome
- Safari

Selenium WebDriver- Features

Some of the most important features of Selenium WebDriver are:

- **Multiple Browser Support**: Selenium WebDriver supports a diverse range of web browsers such as Firefox, Chrome, Internet Explorer, Opera and many more. It also supports some of the non-conventional or rare browsers like HTMLUnit.
- **Multiple Languages Support**: WebDriver also supports most of the commonly used programming languages like Java, C#, JavaScript, PHP, Ruby, Pearl and Python. Thus, the user can choose any one of the supported programming language based on his/her competency and start building the test scripts.

- **Speed**: WebDriver performs faster as compared to other tools of Selenium Suite. Unlike RC, it doesn't require any intermediate server to communicate with the browser; rather the tool directly communicates with the browser.
- **Simple Commands**: Most of the commands used in Selenium WebDriver are easy to implement. For instance, to launch a browser in WebDriver following commands are used,

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

  driver.getTitle();

  driver.manage().window().maximize();

  WebElement element = driver.findElement(By.xpath("//div[@id='elementId']"));
  element.click();

  driver.quit();
  ```

TestNG Frameworks using Selenium:

TestNG Overview:

Using TestNG with Selenium WebDriver creates a powerful testing framework that can handle complex testing scenarios with ease. TestNG's extensive features for managing test execution, such as grouping, prioritization, parallel execution, combined with Selenium's browser automation capabilities, provide a comprehensive solution for automated web testing.

Whether you are running a few simple tests or a large suite of complex, data-driven, and dependent tests, the combination of Selenium and TestNG offers the flexibility and control needed to effectively manage and execute your automated testing efforts.

Key Features of TestNG

Annotations: TestNG provides a variety of annotations to define tests and configure their behavior (`@Test`, `@BeforeMethod`, `@AfterMethod`, etc...)

Test Configuration: It allows configuration at different levels, such as suite, test, class, and method.

Parallel Execution: TestNG can run tests in parallel, speeding up the test execution process.

Flexible Test Configuration: You can include/exclude tests, prioritize them, and group them.

Setting Up TestNG with Selenium

Install TestNG:

In an IDE like IntelliJ IDEA or Eclipse, you can install TestNG as a plugin.

For Maven projects, add the TestNG dependency to your `pom.xml` file:

```
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>7.7.0</version>
  <scope>test</scope>
</dependency>
```

Setup Selenium WebDriver:

Add the Selenium dependency to your `pom.xml` file:

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>4.10.0</version>
</dependency>
```

Download the WebDriver executable for the browser you want to automate (e.g., ChromeDriver).

Basic TestNG Example with Selenium

Here’s an example of using TestNG with Selenium WebDriver to automate a simple test:

Create a Test Class:

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.Assert;

import org.testng.annotations.AfterClass;

import org.testng.annotations.BeforeClass;

import org.testng.annotations.Test;


public class GoogleSearchTest {


  WebDriver driver;


  @BeforeClass

  public void setup() {

    // Set the path to the WebDriver executable

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

    driver = new ChromeDriver();

  }


  @Test

  public void testGoogleSearch() {

    // Open Google

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


    // Find the search box and enter a query

    WebElement searchBox = driver.findElement(By.name("q"));

    searchBox.sendKeys("Selenium WebDriver");

    searchBox.submit();


    // Verify the page title
```

```
        Assert.assertTrue(driver.getTitle().contains("Selenium WebDriver"));

    }


    @AfterClass

    public void closeBrowser() {

        // Close the browser

            driver.quit();

        }

    }

}
```

TestNG XML Configuration:

Create a testng.xml file to define the test suite and test cases

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

<suite name="Selenium Test Suite">

    <test name="Google Search Test">

        <classes>

            <class name="com.example.GoogleSearchTest"/>

        </classes>

    </test>

</suite>
```

Running the Tests:

- You can run the tests directly from your IDE if it supports TestNG.

Grouping Tests: TestNG allows you to group tests so that you can run specific groups of tests selectively. This is useful for organizing tests based on functionality, type, or priority.

```
@Test(groups = { "search" })
```

```
public void testGoogleSearch() {
  driver.get("https://www.google.com");
  WebElement searchBox = driver.findElement(By.name("q"));
  searchBox.sendKeys("Selenium WebDriver");
  searchBox.submit();
  Assert.assertTrue(driver.getTitle().contains("Selenium WebDriver"));
}
@Test(groups = { "navigation" })
public void testNavigation() {
  driver.get("https://www.example.com");
  Assert.assertTrue(driver.getTitle().contains("Example Domain"));
}
```

In the testng.xml file, you can specify which groups to include or exclude:

```
<suite name="Selenium Test Suite">
  <test name="Grouped Tests">
    <groups>
      <run>
        <include name="search"/>
      </run>
    </groups>
    <classes>
      <class name="com.example.GoogleSearchTest"/>
    </classes>
  </test>
</suite>
```

Test Prioritization: TestNG allows you to set the priority of tests to control the order in which they are executed.

```
@Test(priority = 1)
public void testGoogleSearch() {
  driver.get("https://www.google.com");
  WebElement searchBox = driver.findElement(By.name("q"));
  searchBox.sendKeys("Selenium WebDriver");
  searchBox.submit();
  Assert.assertTrue(driver.getTitle().contains("Selenium WebDriver"));
}


@Test(priority = 2)
public void testNavigation() {
  driver.get("https://www.example.com");
  Assert.assertTrue(driver.getTitle().contains("Example Domain"));
}
```

Dependency Tests: You can set dependencies between tests so that certain tests are only run if other tests pass.

```
@Test
public void loginTest() {
  driver.get("https://www.example.com/login");
  WebElement username = driver.findElement(By.name("username"));
  WebElement password = driver.findElement(By.name("password"));
  WebElement loginButton = driver.findElement(By.name("login"));

  username.sendKeys("user");
  password.sendKeys("password");
  loginButton.click();


  Assert.assertTrue(driver.getTitle().contains("Dashboard"));
}


@Test(dependsOnMethods = { "loginTest" })
public void testDashboard() {
  WebElement profileLink = driver.findElement(By.id("profile"));
  profileLink.click();
  Assert.assertTrue(driver.getTitle().contains("Profile"));
}
```

Key Annotations in TestNG

@Test:

- Marks a method as a test case.
- Can include parameters like priority, dependsOnMethods, groups, enabled, etc.

@BeforeSuite and **@AfterSuite**:

- Methods annotated with @BeforeSuite run before any tests in the suite.
- Methods annotated with @AfterSuite run after all tests in the suite have executed.

@BeforeTest and **@AfterTest**:

- Methods annotated with @BeforeTest run before any test methods in the <test> tag in the XML file.
- Methods annotated with @AfterTest run after all test methods in the <test> tag have executed.

@BeforeClass and **@AfterClass**:

- Methods annotated with @BeforeClass run before the first test method in the current class is invoked.

- Methods annotated with @AfterClass run after all the test methods in the current class have been run.

@BeforeMethod and **@AfterMethod**:

- Methods annotated with @BeforeMethod run before each test method.
- Methods annotated with @AfterMethod run after each test method.

Web Locators

Selenium Locators act as navigational tools within a web page's Document Object Model (DOM), serving as a compass for Selenium to identify, manipulate, and interact with various elements such as text boxes, buttons, checkboxes, and more. They form the backbone of automated testing, allowing testers to locate elements regardless of their position in the DOM, facilitating efficient and reliable test scripts.

1. ID Locator

The ID Locator in Selenium targets elements using their unique identifiers within the HTML structure. It's one of the most preferred locators due to its inherent uniqueness, offering a fast and reliable means of element identification. For instance, consider an HTML button element with the ID attribute set as "submit_button":

```
<button id="submit_button">Submit</button>
```

To locate this button using Selenium in Java:

```
WebElement element = driver.findElement(By.id("submit_button"));
```

2. Name Locator

The Name Locator relies on the 'name' attribute of HTML elements. While not always unique across elements, it can effectively identify elements, especially those like form elements where 'name' attributes are commonly used. For example, a text input field in an HTML form:

```
<input type="text" name="username">
```

To locate this input field using Selenium in Java:

```
WebElement element = driver.findElement(By.name("username"));
```

3. Class Name Locator

The Class Name Locator in Selenium targets elements based on the ‘class’ attribute within HTML elements. While not unique to a specific element, classes are valuable when multiple elements share the same class. For instance, consider multiple menu items with the same class:

```
<ul>
 <li>class="menu_item">Home</li>
 <li>class="menu_item">About</li>
 <li>class="menu_item">Services</li>
</ul>
```

To locate these menu items using Selenium in Java:

```
WebElement element = driver.findElement(By.className(“menu_item”))
```

4. Tag Name Locator

This locator identifies elements based on their HTML tag type. It becomes useful when dealing with multiple instances of the same tag. For example, consider multiple input fields:

```
<input type="text"id="username"/>
<input type="text"id="email"/>
<input type="text" id="password"/>
```

To locate all input fields using Selenium in Java:

```
WebElement element = driver.findElement(By.tagName("input"));
```

5. Link Text Locator

The Link Text Locator in Selenium targets elements using the text within anchor (<a>) tags, primarily used for links and hyperlinks. For example, consider a hyperlink with the text “Contact Us”:

```
<a href="contact.html">Contact Us</a>
```

To locate this link using Selenium in Java:

```
WebElement element = driver.findElement(By.linkText("Contact Us"))
```

6. Partial Link Text Locator

Similar to the Link Text Locator, the Partial Link Text Locator allows partial matching of the text within anchor tags. For instance, finding a link when the complete text is not known:

```
<a href="contact.html">Contact</a>
```

To locate this link using Selenium in Java:

```
WebElement element = driver.findElement(By.partialLinkText("Contact"))
```

7. CSS Selector Locator

CSS Selector Locator in Selenium employs CSS selectors to identify elements based on their attributes, hierarchy, and more. For example, consider a div element with a specific structure:

```
<div id="content">
  <h2>Welcome</h2>
</div>
```

To locate the h2 element using CSS Selector in Java:

```
WebElement element = driver.findElement(By.cssSelector(“#content > h2”))
```

8. XPath Locator

The XPath Locator in Selenium uses XPath expressions to navigate through the XML structure of a web page, providing precise element identification, even within complex DOM structures. For instance, consider an input field with a specific ID:

```
<input type="text" id="username">
```

To locate this input field using XPath in Python:

```
element = driver.findElement(By.xpath("//input[@id='username']"))
```

Locators Usage of Each

Each type of locator within Selenium possesses distinct characteristics that make them suitable for different scenarios:

- **ID Locators**: These are highly preferred due to their speed and uniqueness. When an element has a unique ID attribute, using an ID Locator ensures swift identification without ambiguity. However, they might not be available for all elements on a page.
- **CSS Selectors and XPath**: These offer advanced querying capabilities. CSS Selectors leverage CSS syntax to identify elements based on attributes, relationships, or positions in the DOM. XPath, a query language, provides even more granular control over element selection by navigating through the XML structure of a webpage. They are particularly useful for complex structures or when elements lack distinguishing attributes like IDs.

Actions

In Selenium, actions are used to handle complex user interactions such as drag and drop, click and hold, and other advanced gestures. Selenium provides an 'Actions' class to build and perform these actions.

Examples on different actions

1. Click and Hold

Definition: The click_and_hold() action simulates pressing the mouse button down on an element and holding it.

Use Case: Useful for actions like dragging an element or selecting multiple items.

```
WebElement element = driver.findElement(By.id("some_id"));

Actions actions = new Actions(driver);

actions.clickAndHold(element).perform();
```

2. Double Click

Definition: The double_click() action simulates a double mouse click on an element.

Use Case: Useful for actions that require a double-click, such as opening a file or folder in a file explorer interface.

```
WebElement element = driver.findElement(By.id("some_id"));

Actions actions = new Actions(driver);

actions.doubleClick(element).perform();
```

3. Key Down and Key Up

Definition: The key_down() action simulates pressing a key down (holding it), and key_up() simulates releasing the key.

Use Case: Useful for actions that require holding down keys, such as shortcuts (e.g., Ctrl+C for copy).

```
Actions actions = new Actions(driver);

actions.keyDown(Keys.CONTROL).sendKeys("a").keyUp(Keys.CONTROL).perform();
```

4. Drag and Drop

Definition: The drag_and_drop() action simulates clicking on a source element, holding the mouse button down, moving to the target element, and releasing the mouse button.

Use Case: Useful for actions that involve moving elements from one location to another.

```
WebElement sourceElement = driver.findElement(By.id("source_id"));

WebElement targetElement = driver.findElement(By.id("target_id"));

Actions actions = new Actions(driver);

actions.dragAndDrop(sourceElement, targetElement).perform();
```

5. Context Click (Right Click)

Definition: The context_click() action simulates a right-click on an element, typically to open a context menu.

Use Case: Useful for interacting with context menus or performing right-click actions.

```
WebElement element = driver.findElement(By.id("some_id"));

Actions actions = new Actions(driver);
```

actions.contextClick(element).perform();

Windows

In Selenium, managing windows (or tabs) is a common task, especially when dealing with applications that open new windows or tabs as part of their workflow. Selenium WebDriver provides several methods to handle multiple windows, such as switching between them, closing specific windows, and getting window handles.

Key Concepts

- **Window Handle**: A unique identifier for each window or tab.
- **Switching Windows**: Changing the WebDriver's context to interact with a different window or tab.

Basic Example

Here's a basic example of how to handle multiple windows in Selenium:

1. **Open a new window or tab**:
   - This can be done through JavaScript execution or by interacting with an element that opens a new window or tab.
2. **Get window handles**:
   - Use driver.window_handles to get a list of all window handles.
3. **Switch to a different window**:
   - Use driver.switch_to.window(window_handle) to switch to a specific window.
4. **Perform actions on the new window**:
   - Interact with the elements in the newly switched window.
5. **Close a window**:
   - Use driver.close() to close the current window.
6. **Switch back to the original window**:
   - Use the window handle of the original window to switch back.

DataProvider:

A DataProvider in TestNG is a method annotated with ‘@DataProvider’ that supplies data to a test method. The DataProvider method must return an array of objects (Object[][]), where each element in the outer array represents a test iteration, and each element in the inner array represents the data for the parameters of the test method.

It allows for parameterized testing, where test methods can run multiple times with different sets of data. This is particularly useful for testing a wide range of inputs to ensure that the application behaves as expected under various conditions.

How to Implement DataProvider?

1. **Define the DataProvider Method**: The DataProvider method is defined in the same class or a different class and is annotated with @DataProvider.

   ```
   import org.testng.annotations.DataProvider;

    public class TestData {

     @DataProvider(name = "loginData")

     public Object[][] getData() {

       return new Object[][] {

         {"user1", "password1"},

         {"user2", "password2"},

         {"user3", "password3"}

       };

     }

   }
   ```

2. **Use the DataProvider in Test Method**:

   The test method uses the @Test annotation with the ‘dataProvider’ attribute to link to the DataProvider.

   ```
   import org.testng.annotations.Test;

    public class LoginTest {

     @Test(dataProvider = "loginData", dataProviderClass = TestData.class)

     public void loginTest(String username, String password) {

       System.out.println("Username: " + username);

       System.out.println("Password: " + password);   }

   }
   ```

JavaScript Executors:

Executing JavaScript in Selenium is a powerful technique used to interact with web elements or perform operations that are not possible using standard WebDriver commands. Selenium provides the ‘JavascriptExecutor’ interface, which allows you to run JavaScript code within the context of the browser.

Implementing JavaScript Execution in Selenium

Scroll to a Specific Element:

```
JavascriptExecutor js = (JavascriptExecutor) driver;

WebElement element = driver.findElement(By.id("elementId"));
js.executeScript("arguments[0].scrollIntoView(true);", element);
```

Click an Element:

```
JavascriptExecutor js = (JavascriptExecutor) driver;

WebElement element = driver.findElement(By.id("buttonId"));
js.executeScript("arguments[0].click();", element);
```

Fetch the Title of the Page:

```
JavascriptExecutor js = (JavascriptExecutor) driver;

String title = (String) js.executeScript("return document.title;");
System.out.println("Page title is: " + title);
```

Highlight an Element:

```
JavascriptExecutor js = (JavascriptExecutor) driver;

WebElement element = driver.findElement(By.id("elementId"));
js.executeScript("arguments[0].style.border='3px solid red'", element);
```

Allure Reports:

Generating an Allure report in Selenium involves integrating Allure with your testing framework and configuring it to generate reports from your test results. Below are the steps to set up and generate an Allure report for a Selenium project using TestNG.

Step 1: Add Dependencies

```
<!-- Allure TestNG Dependency -->

<dependency>

<groupId>io.qameta.allure</groupId>

 <artifactId>allure-testng</artifactId>

<version>2.14.0</version>

 </dependency>
```

Step 2: Configure Allure TestNG Listener

Create a testng.xml file to configure TestNG and include the Allure TestNG listener.

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

<suite name="Suite">

  <listeners>

    <listener class-name="io.qameta.allure.testng.AllureTestNg"/>

  </listeners>



  <test name="Test">

    <classes>

      <class name="com.example.tests.YourTestClass"/>

    </classes>

  </test>
```

```
</suite>
```

Step 3: Add Allure Annotations (Optional)

Enhance your test methods with Allure annotations to provide detailed information in the report.

```
import io.qameta.allure.Description;

import io.qameta.allure.Step;

import org.testng.Assert;

import org.testng.annotations.Test;

public class YourTestClass {

 @Test

  @Description("Test to verify the login functionality")

  public void loginTest() {

    openLoginPage();

    enterCredentials("user", "password");

    submitForm();

    verifyLoginSuccess();

  }

@Step("Open login page")

  public void openLoginPage() {

    // Code to open the login page

  }

@Step("Enter credentials: {username} / {password}")

  public void enterCredentials(String username, String password) {
    // Code to enter username and password
  }
@Step("Submit the login form")
  public void submitForm() {
    // Code to submit the form
  }
@Step("Verify login success")
  public void verifyLoginSuccess() {
    // Code to verify login was successful
    Assert.assertTrue(true);
  }
}
```

Step 4: Run Your Tests

Step 5: Generate Allure Report

Once the tests have run, it will automatically open a web browser displaying the Allure report.