Selenium and It's Locator:

XPATH and CSS

Selenium is a popular automation testing tool used for web application testing. It provides various locators to find web elements on a webpage, such as ID, name, class, tag, link text, and partial link text. However, there are certain scenarios where these locators may not be sufficient to locate the web element you need to interact with. This is where XPath and CSS selectors come in handy.

Selenium is a popular automation testing tool used for web application testing. It provides various locators to find web elements on a webpage, such as ID, name, class, tag, link text, and partial link text. However, there are certain scenarios where these locators may not be sufficient to locate the web element you need to interact with. This is where XPath and CSS selectors come in handy.

There are several cases in which web elements cannot be easily found using the traditional locators like ID, name, or tag. Some of these cases are:

1.

Dynamic or changing IDs or names: When the IDs or names of web elements are dynamically generated by the web application, they can change every time the page is loaded. In such cases, traditional locators like ID or name may not be useful as they can become invalid or not unique. XPath and CSS selectors can be used to locate web elements based on their attributes, position in the DOM hierarchy, or other relationships with other elements.

Example:

Consider a web application that generates a unique ID for every session or user. Let's say that there is a login page on this web application that has a username input field and a password input field. The HTML code for this page may look something like this:

```
<form>
  <label for="username">Username:</label>
  <input type="text" id="session123_username" name="username">

  <label for="password">Password:</label>
  <input type="password" id="session123_password" name="password">

  <input type="submit" value="Login">
</form>
```

As you can see, the IDs for the username and password input fields are dynamically generated based on the session ID. If we use traditional locators like ID or name to locate these input fields, they may not be useful as the IDs can change every time the page is loaded.

To handle this situation, we can use XPath or CSS selectors to locate the input fields based on their attributes or relationships with other elements. For example, we can use the following XPath expression to locate the username input field:

//input[contains(@id, 'username')]

This expression selects any input element that has an ID containing the substring "username". We can use a similar expression to locate the password input field:

//input[contains(@id, 'password')]

This expression selects any input element that has an ID containing the substring "password". By using these XPath expressions, we can locate the input fields even if their IDs change every time the page is loaded.

Similarly, we can use CSS selectors to locate the input fields based on their attributes. For example, we can use the following CSS selector to locate the username input field:

input[id*='username']

This selector selects any input element that has an ID containing the substring "username". We can use a similar selector to locate the password input field:

input[id*='password']

This selector selects any input element that has an ID containing the substring "password". By using these CSS selectors, we can locate the input fields even if their IDs change every time the page is loaded.

2.

Nested or complex structure: When web elements are nested within other elements, it can be difficult to locate them using traditional locators like ID or name. For example, if you have a complex form with multiple nested divs, finding a specific input field using traditional locators can be difficult. XPath and CSS selectors can be used to locate web elements based on their position in the DOM hierarchy or their relationship with other elements.

Example:

Consider a web page that has a complex form with multiple nested divs. Let's say that there is an input field for the email address, which is nested within several divs. The HTML code for this form may look something like this:

```
<div class="form">
  <div class="row">
    <label for="email">Email:</label>
    <div class="input-wrapper">
      <input type="email" id="email" name="email">
    </div>
  </div>
  <div class="row">
    <label for="password">Password:</label>
```

```
    <div class="input-wrapper">
      <input type="password" id="password" name="password">
    </div>
  </div>
  <input type="submit" value="Submit">
</div>
```

As you can see, the email input field is nested within several divs. If we use traditional locators like ID or name to locate this input field, it can be difficult to find the element. In such cases, we can use XPath or CSS selectors to locate the element based on its position in the DOM hierarchy or its relationship with other elements.

For example, we can use the following XPath expression to locate the email input field:

```
//div[@class='form']//input[@id='email']
```

This expression selects the input element with an ID of 'email' that is located anywhere within a div element with a class of 'form'. By using this expression, we can locate the email input field even if it is nested within several divs.

Similarly, we can use CSS selectors to locate the email input field. For example, we can use the following selector:

```
div.form input#email
```

This selector selects the input element with an ID of 'email' that is a descendant of a div element with a class of 'form'. By using this selector, we can locate the email input field even if it is nested within several divs.

3.

Multiple instances of the same element: When there are multiple instances of the same type of web element on a webpage, it can be difficult to locate a specific element using traditional locators. For example, if you have multiple input fields with the same name or class, it can be difficult to differentiate between them. XPath and CSS selectors can be used to locate web elements based on their position in the DOM hierarchy or their relationship with other elements.

Example:

Consider a web page that has multiple instances of the same type of web element. Let's say that there are multiple input fields with the same name attribute. The HTML code for this page may look something like this:

```
<form>
  <label for="input1">Input 1:</label>
  <input type="text" name="input" id="input1">

  <label for="input2">Input 2:</label>
  <input type="text" name="input" id="input2">
```

```
  <label for="input3">Input 3:</label>
  <input type="text" name="input" id="input3">

  <input type="submit" value="Submit">
</form>
```

As you can see, there are three input fields with the same name attribute. If we use traditional locators like ID or name to locate these input fields, it can be difficult to differentiate between them. In such cases, we can use XPath or CSS selectors to locate the elements based on their position in the DOM hierarchy or their relationship with other elements.

For example, we can use the following XPath expression to locate the second input field with the name attribute of 'input':

```
//input[@name='input'][2]
```

This expression selects the second input element with a name attribute of 'input'. By using this expression, we can locate a specific instance of the input field, even if there are multiple instances of the same type of web element.

Similarly, we can use CSS selectors to locate the second input field with the name attribute of 'input'. For example, we can use the following selector:

```
input[name='input']:nth-of-type(2)
```

This selector selects the second input element with a name attribute of 'input'. By using this selector, we can locate a specific instance of the input field, even if there are multiple instances of the same type of web element.

4.

Elements with dynamic content: When the content of web elements is dynamic and changes frequently, it can be difficult to locate them using traditional locators like ID or name. For example, if you have a dynamic search bar that displays search results in real-time, locating the search results using traditional locators can be difficult. XPath and CSS selectors can be used to locate web elements based on their attributes or other criteria.

Example:

Consider a web page that has a dynamic search bar that displays search results in real-time. The search bar has an input field and a list of search results that are updated as the user types. The HTML code for this page may look something like this:

```
<div class="search-bar">
  <input type="text" name="search" id="search" placeholder="Search...">
  <ul class="search-results">
    <li>Search Result 1</li>
    <li>Search Result 2</li>
```

```
    <li>Search Result 3</li>
  </ul>
</div>
```

As you can see, the search results are dynamic and change frequently as the user types in the search bar. If we use traditional locators like ID or name to locate the search results, it can be difficult to handle the dynamic nature of the content. In such cases, we can use XPath or CSS selectors to locate the elements based on their attributes or other criteria.

For example, we can use the following XPath expression to locate the first search result:

```
//ul[@class='search-results']/li[1]
```

This expression selects the first li element inside the ul element with a class of 'search-results'. By using this expression, we can locate the first search result, even if the content is dynamic and changes frequently.

Similarly, we can use CSS selectors to locate the first search result. For example, we can use the following selector:

```
ul.search-results li:first-of-type
```

This selector selects the first li element inside the ul element with a class of 'search-results'. By using this selector, we can locate the first search result, even if the content is dynamic and changes frequently.

5.

Elements not visible in the DOM: When web elements are not visible in the DOM, it can be difficult to locate them using traditional locators. For example, if you have a hidden element that appears only on certain user actions, locating it using traditional locators can be difficult. XPath and CSS selectors can be used to locate web elements based on their attributes or other criteria, even if they are not visible in the DOM.

Example:

Consider a web page that has a hidden element that appears only on certain user actions. The element has an ID of "hidden-element" and is initially hidden using the display: none style. The HTML code for this page may look something like this:

```
<div class="container">
  <div id="visible-element">This element is visible</div>
  <div id="hidden-element" style="display: none">This element is hidden</div>
</div>
```

As you can see, the hidden element is not visible in the DOM and cannot be located using traditional locators like ID or name. In such cases, we can use XPath or CSS selectors to locate the element based on its attributes or other criteria.

For example, we can use the following XPath expression to locate the hidden element:

```
//div[@id='hidden-element']
```

This expression selects the div element with an id of 'hidden-element'. By using this expression, we can locate the hidden element, even if it is not visible in the DOM.

Similarly, we can use CSS selectors to locate the hidden element. For example, we can use the following selector:

```
#hidden-element
```

This selector selects the element with an id of 'hidden-element'. By using this selector, we can locate the hidden element, even if it is not visible in the DOM.

XPATH Locator:

XPath is defined as XML path. It is a syntax or language for finding any element on the web page using XML path expression.

XPath is a language used for locating elements in an XML or HTML document. In Selenium, XPath is used to locate elements on a webpage by defining their path relative to the root element of the document.

There are two types of XPath in Selenium:

- Absolute XPath
- Relative XPath

1. Absolute XPath:

An absolute XPath is a complete path from the root element to the desired element in the HTML document. It starts with a single slash / and lists all the parent elements leading to the target element.

- Absolute XPath is direct way to find the element in HTML DOM.
- It begins with the single forward slash(/) ,which means you can select the element from the root node.

Here is an example of an absolute XPath expression:

```
/html/body/div[2]/form/input[1]
```

In this expression, we start with the root element /html, then navigate to the body element, then to the second div element, then to the form element, and finally to the first input element.

One of the main disadvantages of using an absolute XPath is that if any of the parent elements change in the HTML structure, the absolute XPath will fail to locate the target element.

2. Relative XPath:

A relative XPath locates the element relative to another element in the HTML document. It starts with a double slash // and lists only the child elements of the parent element that is being referenced.

For Relative Xpath the path starts from the middle of the HTML DOM structure. It starts with the double forward slash (//), which means it can search the element anywhere at the webpage.

➤ Syntax of XPath :

Xpath=//tagname[@attribute='value']

• // : Select current node.

• Tagname: Tagname of the particular node.

• @: Select attribute.

• Attribute: Attribute name of the node.

• Value: Value of the attribute.

Here is an example of a relative XPath expression:

//form/input[@name='username']

In this expression, we start with a double slash // to locate the form element, then we locate the input element with the attribute name equal to username. This is a more reliable way to locate the element as it is not dependent on the complete structure of the HTML document.

Example:

Consider the following HTML code:

```
<html>
  <body>
    <div class="container">
      <h1>Registration Form</h1>
      <form>
        <input type="text" name="username" id="username" />
        <input type="password" name="password" id="password" />
        <button type="submit">Submit</button>
      </form>
    </div>
  </body>
</html>
```

Here are some examples of how we can use XPath to locate the elements on this page:

To locate the h1 element, we can use the following XPath expression:

//h1

To locate the input element with the name username, we can use the following XPath expression:

//input[@name='username']

To locate the button element, we can use the following XPath expression:

//button

To locate the input element with the ID password, we can use the following XPath expression:

//input[@id='password']

To locate the form element, we can use the following XPath expression:

//form

In summary, when web elements are not visible in the DOM, it can be difficult to locate them using traditional locators like ID or name. XPath and CSS selectors can be used to locate web elements based on their attributes or other criteria, even if they are not visible in the DOM.

CSS Locator:

CSS selectors are another type of locator used in Selenium to identify web elements on a web page. CSS stands for Cascading Style Sheets, which is a style sheet language used for describing the look and formatting of a document written in HTML.

CSS locators are often preferred over XPath locators as they are faster and have better browser compatibility. They can be used to locate web elements based on various attributes such as ID, class, name, tag name, and many more.

Here's how to use CSS locators in Selenium:

Finding an element by ID:

To find an element by its ID using CSS locator, use the '#' symbol followed by the ID name.

Syntax: #ID_name

Example:

If the HTML code is: <input type="text" id="username">

To locate this element using CSS locator, use the following command:

driver.find_element_by_css_selector('#username')

Finding an element by class name:

To find an element by its class name using CSS locator, use the '.' symbol followed by the class name.

Syntax: .class_name

Example:

If the HTML code is: <input type="text" class="form-control">

To locate this element using CSS locator, use the following command:

driver.find_element_by_css_selector('.form-control')

Finding an element by tag name:

To find an element by its tag name using CSS locator, simply use the tag name.

Syntax: tag_name

Example:

If the HTML code is: <button>Click me</button>

To locate this element using CSS locator, use the following command:

driver.find_element_by_css_selector('button')

Finding an element by attribute:

To find an element by its attribute using CSS locator, use the attribute name inside square brackets.

Syntax: [attribute_name='attribute_value']

Example:

If the HTML code is: <input type="text" id="username" name="username">

To locate this element using CSS locator, use the following command:

driver.find_element_by_css_selector('input[name="username"]')

Finding an element by multiple attributes:

To find an element by multiple attributes using CSS locator, use the attributes inside square brackets separated by commas.

Syntax: [attribute1_name='attribute1_value'][attribute2_name='attribute2_value']

Example:

If the HTML code is: <input type="text" id="username" class="form-control" name="username">

To locate this element using CSS locator, use the following command:

driver.find_element_by_css_selector('input[class="form-control"][name="username"]')

These are some of the commonly used CSS selectors in Selenium. It is also possible to use CSS locators with wildcards and regular expressions to locate web elements with dynamic attributes. CSS selectors are an efficient and reliable way to locate web elements on a web page.

XPATH VS CSS

The main differences between XPath and CSS selector are:

- **Syntax:** XPath has a more complex syntax compared to CSS selectors.

XPath syntax can be more difficult to learn and understand compared to CSS selectors due to its more complex structure. XPath expressions typically start with a double forward slash (//) to indicate the starting point for the search, followed by one or more element names, attributes, and other selectors. For example, the XPath expression **//div[@class='example']** selects all div elements with a class of "example".

In contrast, CSS selectors use a simpler syntax that is more intuitive for many developers. CSS selectors start with the name of the element to be selected, followed by one or more attribute selectors, class names, or other modifiers. For example, the CSS selector **div.example** selects all div elements with a class of "example".

- **Flexibility:** XPath is more flexible than CSS selectors and can be used to select elements based on their position in the document and their relationship with other elements.

XPath offers more flexibility than CSS selectors in terms of how elements can be selected based on their position in the document hierarchy and their relationships with other elements. For example, XPath allows for the selection of all child elements of a particular parent element using the child axis, or all elements that are descendants of a particular ancestor element using the ancestor axis.

CSS selectors, on the other hand, are more limited in their ability to select elements based on their position in the document hierarchy. CSS selectors can only select immediate children (>), adjacent siblings (+), and general siblings (~) of an element.

- **Hierarchy:** XPath locators allow for more flexible navigation through the HTML hierarchy, whereas CSS locators are more limited to the immediate descendants or siblings of an element.

XPath locators allow for more flexible navigation through the HTML hierarchy, which makes them better suited for complex web pages with nested elements. For example, XPath can be used to select all elements within a certain level of the document hierarchy, or to select elements based on their relationship with other elements.

CSS locators, on the other hand, are more limited to the immediate descendants or siblings of an element. This makes them better suited for simpler web pages with a flatter document hierarchy.

- **Performance:** In general, CSS locators tend to be faster than XPath locators due to the simpler syntax and more limited scope.

  In general, CSS locators tend to be faster than XPath locators due to their simpler syntax and more limited scope. XPath expressions can be more complex and require more processing power to execute, which can slow down page rendering times.

  However, the performance difference between XPath and CSS selectors may be negligible for simpler web pages with a flatter document hierarchy.

- **Cross-browser compatibility:** CSS locators are more widely supported across browsers than XPath locators.

CSS locators are more widely supported across browsers than XPath locators. Most modern web browsers support CSS selectors, but not all of them support XPath expressions. This makes CSS selectors a more reliable choice for cross-browser testing and compatibility.

- **Regular expression support:** XPath locators have built-in support for regular expressions, whereas CSS locators do not.

XPath locators have built-in support for regular expressions, which allows for more powerful and flexible element selection. Regular expressions can be used to select elements based on complex patterns of text or attribute values.

CSS selectors do not have built-in support for regular expressions, which makes them less flexible in this regard. However, some CSS selectors allow for partial matches of attribute values using the ^, $, and * modifiers.

Use cases for XPath:

- Selecting elements based on their position in the document.
- Selecting elements based on their relationship with other elements.
- Selecting elements based on their attributes.

Use cases for CSS selector:

- Selecting elements based on their class, id, or name attributes.
- Selecting elements based on their relationship with other elements.
- Selecting elements based on their attributes.

Waits In Selenium:

Waits are necessary to capture the web-application reaction.

It is one of the biggest pains for an automation engineer to handle sync issues between elements.

Sometimes an application will not be able to load elements due to the below issues.

- Network issue
- Application issues
- Browser stopping JavaScript call.

And so on.

➤ Waits are Helpful when page have the Ajax Calls.

➤ Waits help user to create the Stable Automation.

Types of Wait in Selenium :

- Sleep
- ImplicitlyWait
- ExplicitWait

Sleep :

Sleep() is a method present in thread class.

- It makes selenium to sleep for the specified time.
- Sleep() method will not worry about whether the element is present or not, it just sleeps till mentioned time.
- Sleep() is also called as static wait or blind wait, dead wait.

Implicit Wait:

- By default, Selenium will not wait for an element once the page load completes. It checks for an element on the page then it performs some operation based on your script but if the element is not present then it throws NoSuchElement Exception.
- Our main intention to use Implicit wait in selenium is to instruct Webdriver that waits for a specific amount before throwing an exception.

- So, the implicit wait will tell to the webdriver to wait for certain amount of time before it throws a "NoSuchElement Exception".
- Default wait time of the selenium is 500 milli-seconds, implicitly wait overrides the default wait time of the selenium WebDriver.
- If element is found before implicitly wait time, selenium moves to next commands in the program without waiting to complete the implicitly wait time.
- If driver is not able to find-out the web element within the limit of implicit wait, then it will throw the “NoSuchElementException”.

Note- Implicit wait in selenium webdriver will be applicable throughout your script and will works on all elements in the script once your specified implicit wait. It is also known as Global wait.

Syntax of Implicit wait in selenium webdriver

- driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS); Selenium -3
- driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(30)); Selenium-4

Explicit Wait:

- It is a concept of the dynamic wait which wait dynamically for specific conditions. It can be implemented by WebDriverWait class.
- The explicit wait is used to tell the Web Driver to wait for certain conditions (Expected Conditions) or the maximum time exceeded before throwing an "ElementNotVisibleException" exception.
- Explicit wait will be applicable for only one line (one condition), we have to use it with ExpectedConditions class.
- WebDriverWait by default calls the Expected Condition every 500 milliseconds until it returns successfully.
- Expected Condition will not impact findelement() and findelements() in Selenium.

Syntax of Explicit wait in selenium webdriver

```
// Create object of WebDriverWait class

WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(30));


// Wait till the element is not visible

wait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath("ur xpath here")));
```

Condition for Explicit wait in selenium webdriver

Condition 1- I have a web page which has some login form and after login, it takes a lot of time to load Account page or home page. This page is dynamic it means sometimes it takes 10 seconds to load the homepage, sometimes its 15 second and so on. In this situation, the Explicit wait can help us which will wait until specific page/page title is not present it will keep waiting.

Condition 2- You are working on travel application, and you have filled the web form and clicked on submit button. Now you have to wait until complete data is not loaded or specific data is not loaded. In this case, again we can use Explicit wait in which we can give wait till specific or set of elements are not loaded.

Condition 3- There are some elements on a web page which are hidden and it will be displayed only when specific conditions get true, so we have to wait until these elements are not visible. In this case, again explicit wait will help in which we can specify wait till the element or elements are not visible.

ExpectedConditions is a class in Selenium which has some predefined condition which makes our task easy.

Expected Conditions present in Explicit Wait :

- alertIsPresent()
- elementSelectionStateToBe(locator, selected)
- elementToBeClickable(locator)
- elementToBeSelected(locator)
- frameToBeAvailableAndSwitchToIt(locator)
- presenceOfElementLocated(locator)
- textToBePresentInElement(locator, text)
- title()
- titleIs(title)
- visibilityOf(element)

difference between Implicit and Explicit Wait:

Implicit wait– It only checks the presence of element on WebPage that's all if elements are hidden or any other condition then it will not handle and it will fail your script.

It is applicable for all the element after initialization.

Explicit wait– It has so much capability which we already discussed and it is applicable to the specific element.