What is a CSS Selector?

CSS (Cascading Style Sheets) Selectors in Selenium are used to identify and locate web elements based on their id, class, name, attributes and other attributes. CSS is a preferred locator strategy as it is simpler to write and faster as compared to XPath.

**By.cssSelector(String cssSelector)** method is used to locate the elements in Selenium WebDriver. This method accepts a CSS Selector String as an argument which defines the selection method for the web elements.

**There are five types of CSS Selectors in Selenium tests:**

1. ID
2. Class
3. Attribute

**Example on** [**https://bstackdemo.com/**](https://bstackdemo.com/)

**ID:** driver.findElement(By.cssSelector(“#<id value>”));

**#offers**

**Class:** driver.findElement(By.cssSelector(“.<class value>”));

**.Navbar\_logo\_\_26S5Y**

**Attribute:** driver.findElement(By.cssSelector(“<tagname>[href=’<href value>’]”));

**a[href='/favourites']**

Xpath: I is XML Path Language

It is used to find element on webpage.

**Types of XPath in Selenium**

Here is a quick overview of the two types of Selenium XPath:

**Absolute XPath:** Begins from the root of the HTML document and specifies the complete path to the element. It’s not as flexible and can break if the page structure changes.

**Relative XPath:** Starts from a specific element and navigates through the DOM hierarchy to locate the desired element. It’s more flexible and resilient to changes in the page structure.

**1. Absolute Path**

The simplest XPath locator example in Selenium is to provide the absolute path of an element in the DOM structure.

For instance, consider the HTML below:

<html>

<head>...</head>

<body>

...

<form id="loginForm">

<input name="name" type="text" value="First Name" />

<input name="name" type="text" value="Last Name" />

<input name="email" type="text" value="Business Email" />

<input name="password" type="password" />

<input name="continue" type="submit" value="Sign Me Up" />

</form>

</body>

</html>

**The syntax to select the business email field is as follows:**

html/body/form/input[3]

This searches for the first form tag in the body of the page and selects the third input field in the form. This format, though simple, is also the most vulnerable to minor changes in the page’s structure. This method is also known as a single slash search.

2. Relative Path

A relative path, or a double slash search, begins with double slashes. The double slashes signify a break in the absolute path. Here is how to select the same business email field using a relative path.

//form/input[3]

If multiple forms exist on the page, one may need to provide an extra identifier for the form field.

**Handle Dynamic Elements in Selenium using XPath:**

**1. Using Attributes**

While the example shown above is feasible if only a single form is on the page, one can make the search patterns more robust by using attributes.

//form[@id='loginForm']/input[3]

In place of id, one can use any attribute and its corresponding value to locate an element with Selenium.

While this example shows a single attribute, one can also use multiple attributes of the same tag to locate it on the page.

For instance, to select the Last Name field, one can use the following XPath syntax in Selenium:

//input[@name='name'][@value='Last Name']

**2. Logical Operators in Selections**

While attributes may be sufficient to locate elements in most cases, testers may also need to use logical operators.

For instance, if the HTML structure has name or id attributes populated by the value “name”, one may use the following syntax to select them.

//input[@id='name' or @name='name']

Similarly, one can replace the or keyword with and to only select an element that satisfies all conditions.

3. Using Text

One may search for an element using the text that it contains too. For instance, to select a link that says “Click Me”, one can use the following search:

//a[text()='Click Me']

This snippet searches for any hyperlink containing the text “Click Me”. Replace the tag with a wildcard \* to search for any element that contains the text “Click Me”.

//\*[text()='Click Me']

**Basic Format:**

Xpath = //tagname[@Attribute=‘value’]

**Finding Xpath for Gmail:**

Write xpath of gmail using extension. : find 2.

Then find xpath of image. Find one. Then use preceeding:

Xpath for image: //a[normalize-space()='Images']

\*[normalize-space(): This function removes any leading, trailing, or multiple whitespace characters (spaces, tabs, newlines) from the text content of the element. It ensures consistent matching regardless of how the text appears in the HTML (e.g., "Images", " Images ", "Images\n").]\*

Xpath for gmail: //a[normalize-space()='Images']/preceding::a

Using contains:

Xpath for gmail : //a[contains(text(), 'Images')]/preceding::a

**Axes**

There are thirteen different axes in the XPath specification. An axis represents a relationship to the context node, and is used to locate nodes relative to that node on the tree. For further information on using XPath expressions, please see the For Further Reading section at the end of Transforming XML with XSLT document. Also see the 'axes' section in the xpath spec.

**Site: https://bstackdemo.com/**

**Self :** //\*[@id="orders"]

**following-sibling:** //\*[@id="orders"]/following-sibling::a

**preceding-sibling:** //\*[@id="orders"]/preceding-sibling::a

**Site: https://practice.expandtesting.com/dynamic-table**

**Ancestor Axis (ancestor): //td[text()='Chrome']/ancestor::tr/td**

**Parent Axis (parent): //td[text()='Chrome']/parent::tr/td**

**Child Axis (child)://tr[td[text()='Chrome']]/td**

**Descendant Axis (descendant): //tr[td[text()='Chrome']]/descendant::td**

**Following-sibling Axis (following-sibling): //td[text()='Chrome']/following-sibling::td**

**Preceding-sibling Axis (preceding-sibling): //td[text()='Chrome']/preceding-sibling::td**

**Following Axis (following): //td[text()='Chrome']/following::td**

**Preceding Axis (preceding): //td[text()='Chrome']/preceding::td**

**Self Axis (self): //td[text()='Chrome']/self::td**

**Conversion of xpath to css**

**1. Simple Conversions:**

For basic XPath expressions targeting elements by tag name, ID, or class, you might be able to create equivalent CSS selectors:

* **XPath:** //h1 (targets all <h1> elements)
* **CSS:** h1 (targets all <h1> elements)
* **XPath:** //div[@id='my-element'] (targets the element with ID "my-element")
* **CSS:** #my-element (targets the element with ID "my-element")
* **XPath:** //button[@class='submit'] (targets all <button> elements with class "submit")
* **CSS:** .submit (targets all elements with class "submit")

**2. Descendant Selectors:**

XPath can easily traverse the HTML hierarchy using descendant axes (//, /). CSS offers limited descendant selection capabilities, but you can sometimes achieve similar results by nesting selectors:

* **XPath:** //div/p (targets all <p> elements that are descendants of <div> elements)
* **CSS:** div > p (targets direct child <p> elements of <div> elements, might not capture all descendants)

**3. Attribute Selectors:**

Both XPath and CSS support attribute-based selection with various operators. However, CSS has limitations in terms of complex comparisons and axis navigation:

* **XPath:** //input[@type='text'] (targets all <input> elements with type attribute set to "text")
* **CSS:** input[type="text"] (targets all <input> elements with type attribute set to "text")
* **XPath:** //a[contains(@href, 'product')] (targets all <a> elements with links containing "product")
* **CSS:** a[href\*="product"] (targets all <a> elements with links containing "product" anywhere in the value, might not be as precise as XPath's contains)