Tips on working with locators

When to use which locators and how best to manage them in your code.

Take a look at examples of the supported locator strategies.

In general, if HTML IDs are available, unique, and consistently predictable, they are the preferred method for locating an element on a page. They tend to work very quickly, and forego much processing that comes with complicated DOM traversals.

If unique IDs are unavailable, a well-written CSS selector is the preferred method of locating an element. XPath works as well as CSS selectors, but the syntax is complicated and frequently difficult to debug. Though XPath selectors are very flexible, they are typically not performance tested by browser vendors and tend to be quite slow.

Selection strategies based on *linkText* and *partialLinkText* have drawbacks in that they only work on link elements. Additionally, they call down to querySelectorAll selectors internally in WebDriver.

Tag name can be a dangerous way to locate elements. There are frequently multiple elements of the same tag present on the page. This is mostly useful when calling the *findElements(By)* method which returns a collection of elements.

The recommendation is to keep your locators as compact and readable as possible. Asking WebDriver to traverse the DOM structure is an expensive operation, and the more you can narrow the scope of your search, the better.

class name	Locates elements whose class name contains the search value (compound class names are not permitted)
css selector	Locates elements matching a CSS selector
id	Locates elements whose ID attribute matches the search value
name	Locates elements whose NAME attribute matches the search value
link text	Locates anchor elements whose visible text matches the search value
partial link text	Locates anchor elements whose visible text contains the search value. If multiple elements are matching, only the first one will be selected.
tag name	Locates elements whose tag name matches the search value
xpath	Locates elements matching an XPath expression

class name

The HTML page web element can have attribute class. We can see an example in the above shown HTML snippet. We can identify these elements using the class name locator available in Selenium.

Move Code

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.className("information"));
```

css selector

CSS is the language used to style HTML pages. We can use css selector locator strategy to identify the element on the page. If the element has an id, we create the locator as css = #id. Otherwise the format we follow is css =[attribute=value] . Let us see an example from above HTML snippet. We will create locator for First Name textbox, using css.

Move Code

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.cssSelector("#fname"));
```

id

We can use the ID attribute of an element in a web page to locate it. Generally the ID property should be unique for each element on the web page. We will identify the Last Name field using it.

Move Code

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.id("lname"));
```

name

We can use the NAME attribute of an element in a web page to locate it. Generally the NAME property should be unique for each element on the web page. We will identify the Newsletter checkbox using it.

Move Code

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.name("newsletter"));
```

link text

If the element we want to locate is a link, we can use the link text locator to identify it on the web page. The link text is the text displayed of the link. In the HTML snippet shared, we have a link available, let's see how will we locate it.

Move Code

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.linkText("Selenium Official Page"));
```

partial link text

If the element we want to locate is a link, we can use the partial link text locator to identify it on the web page. The link text is the text displayed of the link. We can pass partial text as value. In the HTML snippet shared, we have a link available, lets see how will we locate it.

Move Code

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.partialLinkText("Official Page"));
```

tag name

We can use the HTML TAG itself as a locator to identify the web element on the page. From the above HTML snippet shared, lets identify the link, using its html tag "a".

Move Code

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.tagName("a"));
```

xpath

A HTML document can be considered as a XML document, and then we can use xpath which will be the path traversed to reach the element of interest to locate the element. The XPath could be absolute xpath, which is created from the root of the document. Example - /html/form/input[1]. This will return the male radio button. Or the xpath could be relative. Example- //input[@name='fname']. This will return the first name text box. Let us create locator for female radio button using xpath.

Move Code

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.xpath("//input[@value='f']"));
```

Relative Locators

Selenium 4 introduces Relative Locators (previously called *Friendly Locators*). These locators are helpful when it is not easy to construct a locator for the desired element, but easy to describe spatially where the element is in relation to an element that does have an easily constructed locator.

How it works

Selenium uses the JavaScript function getBoundingClientRect() to determine the size and position of elements on the page, and can use this information to locate neighboring elements.

Relative locator methods can take as the argument for the point of origin, either a previously located element reference, or another locator. In these examples we'll be using locators only, but you could swap the locator in the final method with an element object and it will work the same.

Available relative locators

Above

If the email text field element is not easily identifiable for some reason, but the password text field element is, we can locate the text field element using the fact that it is an "input" element "above" the password element.

Move Code

```
By emailLocator =
RelativeLocator.with(By.tagName("input")).above(By.id("password"));
```
Copy

Below

If the password text field element is not easily identifiable for some reason, but the email text field element is, we can locate the text field element using the fact that it is an “input” element “below” the email element.

Move Code

```
By passwordLocator =
RelativeLocator.with(By.tagName("input")).below(By.id("email"));
```

Left of

If the cancel button is not easily identifiable for some reason, but the submit button element is, we can locate the cancel button element using the fact that it is a “button” element to the “left of” the submit element.

Move Code

```
By cancelLocator =
RelativeLocator.with(By.tagName("button")).toLeftOf(By.id("submit"))
;
```

Right of

If the submit button is not easily identifiable for some reason, but the cancel button element is, we can locate the submit button element using the fact that it is a “button” element “to the right of” the cancel element.

Move Code

```
By submitLocator =
RelativeLocator.with(By.tagName("button")).toRightOf(By.id("cancel")
);
```

Near

If the relative positioning is not obvious, or it varies based on window size, you can use the near method to identify an element that is at most `50px` away from the provided locator. One great use case for this is to work with a form element that doesn't have an easily constructed locator, but its associated input label element does.

Move Code

```
By emailLocator =
RelativeLocator.with(By.tagName("input")).near(By.id("lbl-email"));
```

Chaining relative locators

You can also chain locators if needed. Sometimes the element is most easily identified as being both above/below one element and right/left of another.

Move Code

```
By submitLocator =
RelativeLocator.with(By.tagName("button")).below(By.id("email")).toR
ightOf(By.id("cancel"));
```