Other locators

Introduction

> **NOTE**
>
> Check out the main locators guide for most common and recommended locators.

In addition to recommended locators like page.get_by_role() and page.get_by_text(), Playwright supports a variety of other locators described in this guide.

CSS locator

> **NOTE**
>
> We recommend prioritizing user-visible locators like text or accessible role instead of using CSS that is tied to the implementation and could break when the page changes.

Playwright can locate an element by CSS selector.

Sync Async

```
page.locator("css=button").click()
```

Playwright augments standard CSS selectors in two ways:

- CSS selectors pierce open shadow DOM.
- Playwright adds custom pseudo-classes like `:visible`, `:has-text()`, `:has()`, `:is()`, `:nth-match()` and more.

CSS: matching by text

Playwright include a number of CSS pseudo-classes to match elements by their text content.

- `article:has-text("Playwright")` - the `:has-text()` matches any element containing specified text somewhere inside, possibly in a child or a descendant element. Matching is case-insensitive, trims whitespace and searches for a substring.

  For example, `article:has-text("Playwright")` matches `<article><div>Playwright</div></article>`.

  Note that `:has-text()` should be used together with other CSS specifiers, otherwise it will match all the elements containing specified text, including the `<body>`.

  Sync **Async**

  ```
  # Wrong, will match many elements including <body>
  page.locator(':has-text("Playwright")').click()
  # Correct, only matches the <article> element
  page.locator('article:has-text("All products")').click()
  ```

- `#nav-bar :text("Home")` - the `:text()` pseudo-class matches the smallest element containing specified text. Matching is case-insensitive, trims whitespace and searches for a substring.

  For example, this will find an element with text "Home" somewhere inside the `#nav-bar` element:

  Sync **Async**

  ```
  page.locator("#nav-bar :text('Home')").click()
  ```

- `#nav-bar :text-is("Home")` - the `:text-is()` pseudo-class matches the smallest element with exact text. Exact matching is case-sensitive, trims whitespace and searches for the full string.

  For example, `:text-is("Log")` does not match `<button>Log in</button>` because `<button>` contains a single text node `"Log in"` that is not equal to `"Log"`. However, `:text-is("Log")` matches `<button> Log <span>in</span></button>`, because `<button>` contains a text node `" Log "`.

Similarly, `:text-is("Download")` will not match `<button>download</button>` because it is case-sensitive.

- `#nav-bar :text-matches("reg?ex", "i")` - the `:text-matches()` pseudo-class matches the smallest element with text content matching the JavaScript-like regex.

  For example, `:text-matches("Log\s*in", "i")` matches `<button>Login</button>` and `<button>log IN</button>`.

> **NOTE**
>
> Text matching always normalizes whitespace. For example, it turns multiple spaces into one, turns line breaks into spaces and ignores leading and trailing whitespace.

> **NOTE**
>
> Input elements of the type `button` and `submit` are matched by their `value` instead of text content. For example, `:text("Log in")` matches `<input type=button value="Log in">`.

CSS: matching only visible elements

Playwright supports the `:visible` pseudo class in CSS selectors. For example, `css=button` matches all the buttons on the page, while `css=button:visible` only matches visible buttons. This is useful to distinguish elements that are very similar but differ in visibility.

Consider a page with two buttons, first invisible and second visible.

```
<button style='display: none'>Invisible</button>
<button>Visible</button>
```

- This will find both buttons and throw a strictness violation error:

Sync **Async**

```
page.locator("button").click()
```

- This will only find a second button, because it is visible, and then click it.

Sync Async

```
page.locator("button:visible").click()
```

CSS: elements that contain other elements

The `:has()` pseudo-class is an experimental CSS pseudo-class. It returns an element if any of the selectors passed as parameters relative to the `:scope` of the given element match at least one element.

Following snippet returns text content of an `<article>` element that has a `<div class=promo>` inside.

Sync Async

```
page.locator("article:has(div.promo)").text_content()
```

CSS: elements matching one of the conditions

Comma-separated list of CSS selectors will match all elements that can be selected by one of the selectors in that list.

Sync Async

```
# Clicks a <button> that has either a "Log in" or "Sign in" text.
page.locator('button:has-text("Log in"), button:has-text("Sign in")').click()
```

The `:is()` pseudo-class is an experimental CSS pseudo-class that may be useful for specifying a list of extra conditions on an element.

CSS: matching elements based on layout

> **NOTE**

> Matching based on layout may produce unexpected results. For example, a different element could be matched when layout changes by one pixel.

Sometimes, it is hard to come up with a good selector to the target element when it lacks distinctive features. In this case, using Playwright layout CSS pseudo-classes could help. These can be combined with regular CSS to pinpoint one of the multiple choices.

For example, `input:right-of(:text("Password"))` matches an input field that is to the right of text "Password" - useful when the page has multiple inputs that are hard to distinguish between each other.

Note that layout pseudo-classes are useful in addition to something else, like `input`. If you use a layout pseudo-class alone, like `:right-of(:text("Password"))`, most likely you'll get not the input you are looking for, but some empty element in between the text and the target input.

Layout pseudo-classes use bounding client rect to compute distance and relative position of the elements.

- `:right-of(div > button)` - Matches elements that are to the right of any element matching the inner selector, at any vertical position.
- `:left-of(div > button)` - Matches elements that are to the left of any element matching the inner selector, at any vertical position.
- `:above(div > button)` - Matches elements that are above any of the elements matching the inner selector, at any horizontal position.
- `:below(div > button)` - Matches elements that are below any of the elements matching the inner selector, at any horizontal position.
- `:near(div > button)` - Matches elements that are near (within 50 CSS pixels) any of the elements matching the inner selector.

Note that resulting matches are sorted by their distance to the anchor element, so you can use locator.first to pick the closest one. This is only useful if you have something like a list of similar elements, where the closest is obviously the right one. However, using locator.first in other cases most likely won't work as expected - it will not target the element you are searching for, but some other element that happens to be the closest like a random empty `<div>`, or an element that is scrolled out and is not currently visible.

Sync **Async**

```
# Fill an input to the right of "Username".
page.locator("input:right-of(:text(\"Username\"))").fill("value")

# Click a button near the promo card.
page.locator("button:near(.promo-card)").click()

# Click the radio input in the list closest to the "Label 3".
page.locator("[type=radio]:left-of(:text(\"Label 3\"))").first.click()
```

All layout pseudo-classes support optional maximum pixel distance as the last argument. For example `button:near(:text("Username"), 120)` matches a button that is at most 120 CSS pixels away from the element with the text "Username".

CSS: pick n-th match from the query result

> **NOTE**
>
> It is usually possible to distinguish elements by some attribute or text content, which is more resilient to page changes.

Sometimes page contains a number of similar elements, and it is hard to select a particular one. For example:

```
<section> <button>Buy</button> </section>
<article><div> <button>Buy</button> </div></article>
<div><div> <button>Buy</button> </div></div>
```

In this case, `:nth-match(:text("Buy"), 3)` will select the third button from the snippet above. Note that index is one-based.

Sync **Async**

```
# Click the third "Buy" button
page.locator(":nth-match(:text('Buy'), 3)").click()
```

`:nth-match()` is also useful to wait until a specified number of elements appear, using locator.wait_for().

Sync Async

```
# Wait until all three buttons are visible
page.locator(":nth-match(:text('Buy'), 3)").wait_for()
```

> **NOTE**
>
> Unlike `:nth-child()`, elements do not have to be siblings, they could be anywhere on the page. In the snippet above, all three buttons match `:text("Buy")` selector, and `:nth-match()` selects the third button.

N-th element locator

You can narrow down query to the n-th match using the `nth=` locator passing a zero-based index.

Sync Async

```
# Click first button
page.locator("button").locator("nth=0").click()

# Click last button
page.locator("button").locator("nth=-1").click()
```

Parent element locator

When you need to target a parent element of some other element, most of the time you should locator.filter() by the child locator. For example, consider the following DOM structure:

```
<li><label>Hello</label></li>
<li><label>World</label></li>
```

If you'd like to target the parent `<li>` of a label with text `"Hello"`, using locator.filter() works best:

Sync Async

```
child = page.get_by_text("Hello")
parent = page.get_by_role("listitem").filter(has=child)
```

Alternatively, if you cannot find a suitable locator for the parent element, use `xpath=..`. Note that this method is not as reliable, because any changes to the DOM structure will break your tests. Prefer locator.filter() when possible.

Sync Async

```
parent = page.get_by_text("Hello").locator('xpath=..')
```

React locator

> **NOTE**
>
> React locator is experimental and prefixed with `_`. The functionality might change in future.

React locator allows finding elements by their component name and property values. The syntax is very similar to CSS attribute selectors and supports all CSS attribute selector operators.

In React locator, component names are transcribed with **CamelCase**.

Sync Async

```
page.locator("_react=BookItem").click()
```

More examples:

- match by **component**: `_react=BookItem`
- match by component and **exact property value**, case-sensitive: `_react=BookItem[author = "Steven King"]`
- match by property value only, **case-insensitive**: `_react=[author = "steven king" i]`

- match by component and **truthy property value**: `_react=MyButton[enabled]`
- match by component and **boolean value**: `_react=MyButton[enabled = false]`
- match by property **value substring**: `_react=[author *= "King"]`
- match by component and **multiple properties**: `_react=BookItem[author *= "king" i][year = 1990]`
- match by **nested** property value: `_react=[some.nested.value = 12]`
- match by component and property value **prefix**: `_react=BookItem[author ^= "Steven"]`
- match by component and property value **suffix**: `_react=BookItem[author $= "Steven"]`
- match by component and **key**: `_react=BookItem[key = '2']`
- match by property value **regex**: `_react=[author = /Steven(\\s+King)?/i]`

To find React element names in a tree use React DevTools.

> **NOTE**
>
> React locator supports React 15 and above.

> **NOTE**
>
> React locator, as well as React DevTools, only work against **unminified** application builds.

Vue locator

> **NOTE**
>
> Vue locator is experimental and prefixed with `_`. The functionality might change in future.

Vue locator allows finding elements by their component name and property values. The syntax is very similar to CSS attribute selectors and supports all CSS attribute selector operators.

In Vue locator, component names are transcribed with **kebab-case**.

Sync **Async**

```
page.locator("_vue=book-item").click()
```

More examples:

- match by **component**: `_vue=book-item`
- match by component and **exact property value**, case-sensitive: `_vue=book-item[author = "Steven King"]`
- match by property value only, **case-insensitive**: `_vue=[author = "steven king" i]`
- match by component and **truthy property value**: `_vue=my-button[enabled]`
- match by component and **boolean value**: `_vue=my-button[enabled = false]`
- match by property **value substring**: `_vue=[author *= "King"]`
- match by component and **multiple properties**: `_vue=book-item[author *= "king" i][year = 1990]`
- match by **nested** property value: `_vue=[some.nested.value = 12]`
- match by component and property value **prefix**: `_vue=book-item[author ^= "Steven"]`
- match by component and property value **suffix**: `_vue=book-item[author $= "Steven"]`
- match by property value **regex**: `_vue=[author = /Steven(\\s+King)?/i]`

To find Vue element names in a tree use Vue DevTools.

> **NOTE**
>
> Vue locator supports Vue2 and above.

> **NOTE**
>
> Vue locator, as well as Vue DevTools, only work against **unminified** application builds.

XPath locator

> **DANGER**
>
> We recommend prioritizing user-visible locators like text or accessible role instead of using XPath that is tied to the implementation and easily break when the page changes.

XPath locators are equivalent to calling `Document.evaluate`.

Sync **Async**

```
page.locator("xpath=//button").click()
```

> **NOTE**
>
> Any selector string starting with `//` or `..` are assumed to be an xpath selector. For example, Playwright converts `'//html/body'` to `'xpath=//html/body'`.

> **NOTE**
>
> XPath does not pierce shadow roots.

XPath union

Pipe operator (`|`) can be used to specify multiple selectors in XPath. It will match all elements that can be selected by one of the selectors in that list.

Sync | Async

```
# Waits for either confirmation dialog or load spinner.
page.locator("//span[contains(@class,
'spinner__loading')]|//div[@id='confirmation']").wait_for()
```

Label to form control retargeting

> **DANGER**
>
> We recommend locating by label text instead of relying to label-to-control retargeting.

Targeted input actions in Playwright automatically distinguish between labels and controls, so you can target the label to perform an action on the associated control.

For example, consider the following DOM structure: `<label for="password">Password:</label> <input id="password" type="password">`. You can target the label by it's "Password" text using page.get_by_text(). However, the following actions will be performed on the input instead of the label:

- locator.click() will click the label and automatically focus the input field;

- locator.fill() will fill the input field;
- locator.input_value() will return the value of the input field;
- locator.select_text() will select text in the input field;
- locator.set_input_files() will set files for the input field with `type=file`;
- locator.select_option() will select an option from the select box.

Sync **Async**

```
# Fill the input by targeting the label.
page.get_by_text("Password").fill("secret")
```

However, other methods will target the label itself, for example expect(locator).to_have_text() will assert the text content of the label, not the input field.

Sync **Async**

```
# Fill the input by targeting the label.
expect(page.locator("label")).to_have_text("Password")
```

Legacy text locator

> **DANGER**
>
> We recommend the modern text locator instead.

Legacy text locator matches elements that contain passed text.

Sync **Async**

```
page.locator("text=Log in").click()
```

Legacy text locator has a few variations:

- `text=Log in` - default matching is case-insensitive, trims whitespace and searches for a substring. For example, `text=Log` matches `<button>Log in</button>`.

Sync **Async**

```
page.locator("text=Log in").click()
```

- `text="Log in"` - text body can be escaped with single or double quotes to search for a text node with exact content after trimming whitespace.

  For example, `text="Log"` does not match `<button>Log in</button>` because `<button>` contains a single text node `"Log in"` that is not equal to `"Log"`. However, `text="Log"` matches `<button> Log <span>in</span></button>`, because `<button>` contains a text node `" Log "`. This exact mode implies case-sensitive matching, so `text="Download"` will not match `<button>download</button>`.

  Quoted body follows the usual escaping rules, e.g. use `\"` to escape double quote in a double-quoted string: `text="foo\"bar"`.

Sync **Async**

```
page.locator("text='Log in'").click()
```

- `/Log\s*in/i` - body can be a JavaScript-like regex wrapped in `/` symbols. For example, `text=/Log\s*in/i` matches `<button>Login</button>` and `<button>log IN</button>`.

Sync **Async**

```
page.locator("text=/Log\s*in/i").click()
```

> **NOTE**
>
> String selectors starting and ending with a quote (either `"` or `'`) are assumed to be a legacy text locators. For example, `"Log in"` is converted to `text="Log in"` internally.

NOTE

Matching always normalizes whitespace. For example, it turns multiple spaces into one, turns line breaks into spaces and ignores leading and trailing whitespace.

NOTE

Input elements of the type `button` and `submit` are matched by their `value` instead of text content. For example, `text=Log in` matches `<input type=button value="Log in">`.

id, data-testid, data-test-id, data-test selectors

DANGER

We recommend locating by test id instead.

Playwright supports shorthand for selecting elements using certain attributes. Currently, only the following attributes are supported:

- `id`
- `data-testid`
- `data-test-id`
- `data-test`

Sync Async

```
# Fill an input with the id "username"
page.locator('id=username').fill('value')

# Click an element with data-test-id "submit"
page.locator('data-test-id=submit').click()
```

NOTE

Attribute selectors are not CSS selectors, so anything CSS-specific like `:enabled` is not supported. For more features, use a proper css selector, e.g. `css=[data-`

`test="login"]:enabled`.

> **NOTE**
>
> Attribute selectors pierce shadow DOM. To opt-out from this behavior, use `:light` suffix after attribute, for example `page.locator('data-test-id:light=submit').click()`

Chaining selectors

> **DANGER**
>
> We recommend chaining locators instead.

Selectors defined as `engine=body` or in short-form can be combined with the `>>` token, e.g. `selector1 >> selector2 >> selectors3`. When selectors are chained, the next one is queried relative to the previous one's result.

For example,

```
css=article >> css=.bar > .baz >> css=span[attr=value]
```

is equivalent to

```
document
    .querySelector('article')
    .querySelector('.bar > .baz')
    .querySelector('span[attr=value]');
```

If a selector needs to include `>>` in the body, it should be escaped inside a string to not be confused with chaining separator, e.g. `text="some >> text"`.

Intermediate matches

> **DANGER**
>
> We recommend filtering by another locator to locate elements that contain other elements.

By default, chained selectors resolve to an element queried by the last selector. A selector can be prefixed with `*` to capture elements that are queried by an intermediate selector.

For example, `css=article >> text=Hello` captures the element with the text `Hello`, and `*css=article >> text=Hello` (note the `*`) captures the `article` element that contains some element with the text `Hello`.