

Locators

Introduction

Locators are the central piece of Playwright's auto-waiting and retry-ability. In a nutshell, locators represent a way to find element(s) on the page at any moment.

Quick Guide

These are the recommended built-in locators.

- `page.get_by_role()` to locate by explicit and implicit accessibility attributes.
- `page.get_by_text()` to locate by text content.
- `page.get_by_label()` to locate a form control by associated label's text.
- `page.get_by_placeholder()` to locate an input by placeholder.
- `page.get_by_alt_text()` to locate an element, usually image, by its text alternative.
- `page.get_by_title()` to locate an element by its title attribute.
- `page.get_by_test_id()` to locate an element based on its `data-testid` attribute (other attributes can be configured).

Sync **Async**

```
page.get_by_label("User Name").fill("John")

page.get_by_label("Password").fill("secret-password")

page.get_by_role("button", name="Sign in").click()

expect(page.get_by_text("Welcome, John!")).to_be_visible()
```

Locating elements

Playwright comes with multiple built-in locators. To make tests resilient, we recommend prioritizing user-facing attributes and explicit contracts such as `page.get_by_role()`.

For example, consider the following DOM structure.



Locate the element by its role of `button` with name "Sign in".

Sync **Async**

```
page.get_by_role("button", name="Sign in").click()
```

i NOTE

Use the [code generator](#) to generate a locator, and then edit it as you'd like.

Every time a locator is used for an action, an up-to-date DOM element is located in the page. In the snippet below, the underlying DOM element will be located twice, once prior to every action. This means that if the DOM changes in between the calls due to re-render, the new element corresponding to the locator will be used.

Sync **Async**

```
locator = page.get_by_role("button", name="Sign in")

locator.hover()
locator.click()
```

Note that all methods that create a locator, such as `page.get_by_label()`, are also available on the `Locator` and `FrameLocator` classes, so you can chain them and iteratively narrow down your locator.

Sync **Async**

```
locator = page.frame_locator("my-frame").get_by_role("button", name="Sign in")
```

```
locator.click()
```

Locate by role

The `page.get_by_role()` locator reflects how users and assistive technology perceive the page, for example whether some element is a button or a checkbox. When locating by role, you should usually pass the accessible name as well, so that the locator pinpoints the exact element.

For example, consider the following DOM structure.



The screenshot shows a web browser window with the address `http://localhost:3000`. The page content is a 'Sign up' form with a checkbox labeled 'Subscribe' and a 'Submit' button. To the right of the form, the DOM structure is displayed, showing the HTML elements and their roles:

```
<h3>Sign up</h3>
<label>
  <input type="checkbox" /> Subscribe
</label>
<br/>
<button>Submit</button>
```

You can locate each element by its implicit role:

Sync **Async**

```
expect(page.get_by_role("heading", name="Sign up")).to_be_visible()

page.get_by_role("checkbox", name="Subscribe").check()

page.get_by_role("button", name=re.compile("submit",
re.IGNORECASE)).click()
```

Role locators include **buttons**, **checkboxes**, **headings**, **links**, **lists**, **tables**, and **many more** and follow W3C specifications for **ARIA role**, **ARIA attributes** and **accessible name**. Note that many html elements like `<button>` have an **implicitly defined role** that is recognized by the role locator.

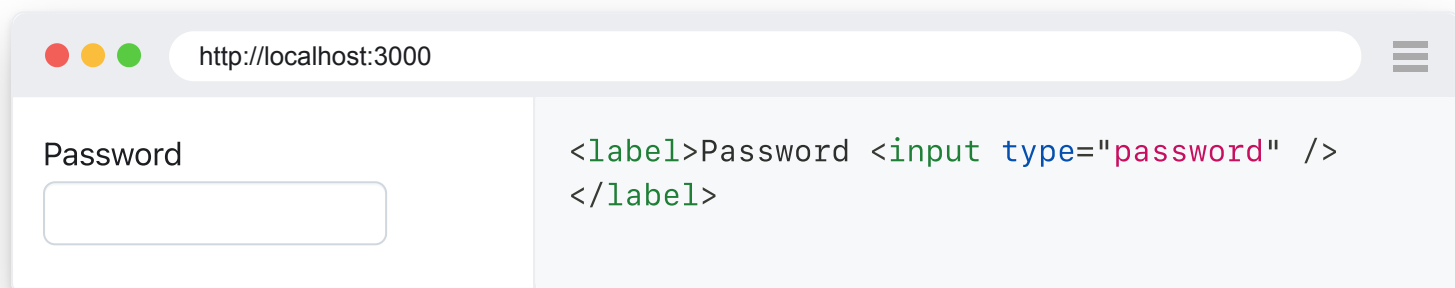
Note that role locators **do not replace** accessibility audits and conformance tests, but rather give early feedback about the ARIA guidelines.

We recommend prioritizing role locators to locate elements, as it is the closest way to how users and assistive technology perceive the page.

Locate by label

Most form controls usually have dedicated labels that could be conveniently used to interact with the form. In this case, you can locate the control by its associated label using `page.get_by_label()`.

For example, consider the following DOM structure.



You can fill the input after locating it by the label text:

Sync Async

```
page.get_by_label("Password").fill("secret")
```

WHEN TO USE LABEL LOCATORS

Use this locator when locating form fields.

Locate by placeholder

Inputs may have a placeholder attribute to hint to the user what value should be entered. You can locate such an input using `page.get_by_placeholder()`.

For example, consider the following DOM structure.



You can fill the input after locating it by the placeholder text:

Sync

Async

```
page.get_by_placeholder("name@example.com").fill("playwright@microsoft.com")
```

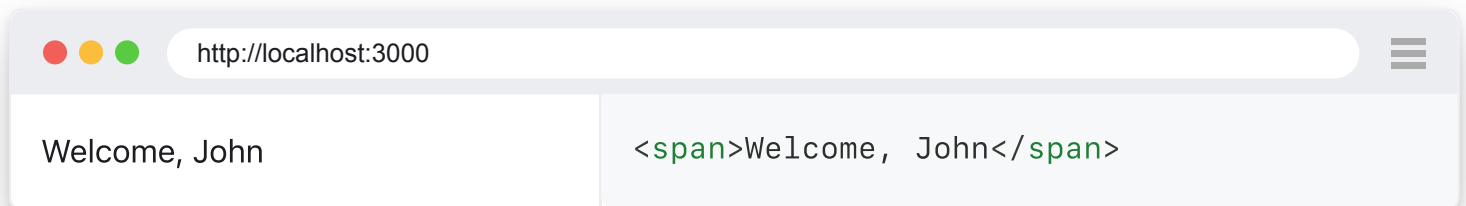
i WHEN TO USE PLACEHOLDER LOCATORS

Use this locator when locating form elements that do not have labels but do have placeholder texts.

Locate by text

Find an element by the text it contains. You can match by a substring, exact string, or a regular expression when using `page.get_by_text()`.

For example, consider the following DOM structure.



You can locate the element by the text it contains:

Sync

Async

```
expect(page.get_by_text("Welcome, John")).to_be_visible()
```

Set an exact match:

Sync

Async

```
expect(page.get_by_text("Welcome, John", exact=True)).to_be_visible()
```

Match with a regular expression:

```
expect(page.get_by_text(re.compile("welcome, john",  
re.IGNORECASE))).to_be_visible()
```

i NOTE

Matching by text always normalizes whitespace, even with exact match. For example, it turns multiple spaces into one, turns line breaks into spaces and ignores leading and trailing whitespace.

i WHEN TO USE TEXT LOCATORS

We recommend using text locators to find non interactive elements like `div`, `span`, `p`, etc. For interactive elements like `button`, `a`, `input`, etc. use [role locators](#).

You can also [filter by text](#) which can be useful when trying to find a particular item in a list.

Locate by alt text

All images should have an `alt` attribute that describes the image. You can locate an image based on the text alternative using `page.get_by_alt_text()`.

For example, consider the following DOM structure.



You can click on the image after locating it by the text alternative:

```
page.get_by_alt_text("playwright logo").click()
```

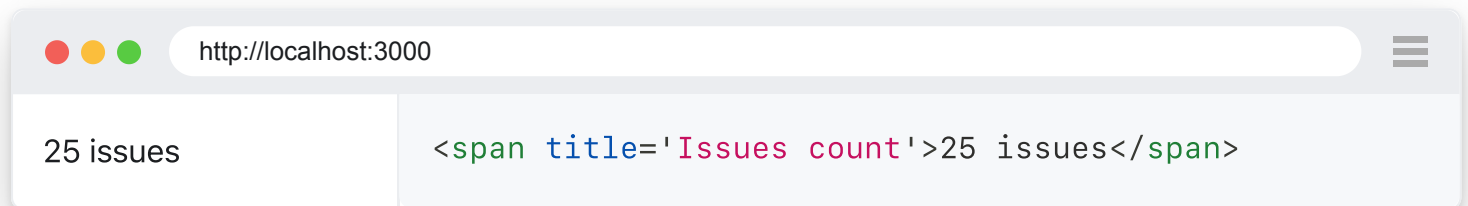
WHEN TO USE ALT LOCATORS

Use this locator when your element supports alt text such as `img` and `area` elements.

Locate by title

Locate an element with a matching title attribute using `page.get_by_title()`.

For example, consider the following DOM structure.



You can check the issues count after locating it by the title text:

Sync Async

```
expect(page.get_by_title("Issues count")).to_have_text("25 issues")
```

WHEN TO USE TITLE LOCATORS

Use this locator when your element has the `title` attribute.

Locate by test id

Testing by test ids is the most resilient way of testing as even if your text or role of the attribute changes, the test will still pass. QA's and developers should define explicit test ids and query them with `page.get_by_test_id()`. However testing by test ids is not user facing. If the role or text value is important to you then consider using user facing locators such as `role` and `text locators`.

For example, consider the following DOM structure.



You can locate the element by its test id:

Sync Async

```
page.get_by_test_id("directions").click()
```

i WHEN TO USE TESTID LOCATORS

You can also use test ids when you choose to use the test id methodology or when you can't locate by role or text.

Set a custom test id attribute

By default, `page.get_by_test_id()` will locate elements based on the `data-testid` attribute, but you can configure it in your test config or by calling `selectors.set_test_id_attribute()`.

Set the test id to use a custom data attribute for your tests.

Sync Async

```
playwright.selectors.set_test_id_attribute("data-pw")
```

In your html you can now use `data-pw` as your test id instead of the default `data-testid`.



And then locate the element as you would normally do:

Sync Async

```
page.get_by_test_id("directions").click()
```

Locate by CSS or XPath

If you absolutely must use CSS or XPath locators, you can use `page.locator()` to create a locator that takes a selector describing how to find an element in the page. Playwright supports CSS and XPath selectors, and auto-detects them if you omit `css=` or `xpath=` prefix.

Sync **Async**

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

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

XPath and CSS selectors can be tied to the DOM structure or implementation. These selectors can break when the DOM structure changes. Long CSS or XPath chains below are an example of a **bad practice** that leads to unstable tests:

Sync **Async**

```
page.locator(
  "#tsf > div:nth-child(2) > div.A8SBwf > div.RNNXgb > div > div.a4bIc
  > input"
).click()

page.locator('//*[@id="tsf"]/div[2]/div[1]/div[1]/div/div[2]/input').click()
```

WHEN TO USE THIS

CSS and XPath are not recommended as the DOM can often change leading to non resilient tests. Instead, try to come up with a locator that is close to how the user perceives the page such as [role locators](#) or [define an explicit testing contract](#) using test ids.

Locate in Shadow DOM

All locators in Playwright **by default** work with elements in Shadow DOM. The exceptions are:

- Locating by XPath does not pierce shadow roots.
- **Closed-mode shadow roots** are not supported.

Consider the following example with a custom web component:

```
<x-details role=button aria-expanded=true aria-controls=inner-details>
  <div>Title</div>
  #shadow-root
    <div id=inner-details>Details</div>
</x-details>
```

You can locate in the same way as if the shadow root was not present at all.

To click `<div>Details</div>`:

Sync **Async**

```
page.get_by_text("Details").click()
```

```
<x-details role=button aria-expanded=true aria-controls=inner-details>
  <div>Title</div>
  #shadow-root
    <div id=inner-details>Details</div>
</x-details>
```

To click `<x-details>`:

Sync **Async**

```
page.locator("x-details", has_text="Details" ).click()
```

```
<x-details role=button aria-expanded=true aria-controls=inner-details>
  <div>Title</div>
  #shadow-root
    <div id=inner-details>Details</div>
</x-details>
```

To ensure that `<x-details>` contains the text "Details":

Sync **Async**

```
expect(page.locator("x-details")).to_contain_text("Details")
```

Filtering Locators

Consider the following DOM structure where we want to click on the buy button of the second product card. We have a few options in order to filter the locators to get the right one.



Filter by text

Locators can be filtered by text with the `locator.filter()` method. It will search for a particular string somewhere inside the element, possibly in a descendant element, case-insensitively. You can also pass a regular expression.

Sync **Async**

```
page.get_by_role("listitem").filter(has_text="Product 2").get_by_role(
    "button", name="Add to cart"
).click()
```

Use a regular expression:

Sync **Async**

```
page.get_by_role("listitem").filter(has_text=re.compile("Product
2"))).get_by_role(
```

```
"button", name="Add to cart")
.click()
```

Filter by not having text

Alternatively, filter by **not having** text:

Sync Async

```
# 5 in-stock items
expect(page.get_by_role("listitem").filter(has_not_text="Out of
stock")).to_have_count(5)
```

Filter by child/descendant

Locators support an option to only select elements that have or have not a descendant matching another locator. You can therefore filter by any other locator such as a `locator.get_by_role()`, `locator.get_by_test_id()`, `locator.get_by_text()` etc.

The screenshot shows a web browser window with the address bar displaying `http://localhost:3000`. The page content consists of two product cards. The first card, titled "Product 1", contains an "Add to cart" button. The second card, titled "Product 2", also contains an "Add to cart" button. To the right of the browser window, the HTML structure of the page is displayed, showing a `` containing two `` elements. Each `` contains a `<h3>` element for the product name and a `<button>` element for the "Add to cart" button.

Sync Async

```
page.get_by_role("listitem").filter(
  has=page.get_by_role("heading", name="Product 2")
).get_by_role("button", name="Add to cart").click()
```

We can also assert the product card to make sure there is only one:

Sync **Async**

```
expect(
  page.get_by_role("listitem").filter(
    has=page.get_by_role("heading", name="Product 2")
  )
).to_have_count(1)
```

The filtering locator **must be relative** to the original locator and is queried starting with the original locator match, not the document root. Therefore, the following will not work, because the filtering locator starts matching from the `` list element that is outside of the `` list item matched by the original locator:

Sync **Async**

```
# ✖ WRONG
expect(
  page.get_by_role("listitem").filter(
    has=page.get_by_role("list").get_by_role("heading", name="Product
2")
  )
).to_have_count(1)
```

Filter by not having child/descendant

We can also filter by **not having** a matching element inside.

Sync **Async**

```
expect(
  page.get_by_role("listitem").filter(
    has_not=page.get_by_role("heading", name="Product 2")
  )
).to_have_count(1)
```

Note that the inner locator is matched starting from the outer one, not from the document root.

Locator operators

Matching inside a locator

You can chain methods that create a locator, like `page.get_by_text()` or `locator.get_by_role()`, to narrow down the search to a particular part of the page.

In this example we first create a locator called `product` by locating its role of `listitem`. We then filter by text. We can use the `product` locator again to get by role of `button` and click it and then use an assertion to make sure there is only one product with the text "Product 2".

Sync **Async**

```
product = page.get_by_role("listitem").filter(has_text="Product 2")

product.get_by_role("button", name="Add to cart").click()
```

You can also chain two locators together, for example to find a "Save" button inside a particular dialog:

Sync **Async**

```
save_button = page.get_by_role("button", name="Save")
# ...
dialog = page.get_by_test_id("settings-dialog")
dialog.locator(save_button).click()
```

Matching two locators simultaneously

Method `locator.and_()` narrows down an existing locator by matching an additional locator. For example, you can combine `page.get_by_role()` and `page.get_by_title()` to match by both role and title.

Sync **Async**

```
button = page.get_by_role("button").and_(page.getByTitle("Subscribe"))
```

Matching one of the two alternative locators

If you'd like to target one of the two or more elements, and you don't know which one it will be, use `locator.or_()` to create a locator that matches any one or both of the alternatives.

For example, consider a scenario where you'd like to click on a "New email" button, but sometimes a security settings dialog shows up instead. In this case, you can wait for either a "New email" button, or a dialog and act accordingly.

NOTE

If both "New email" button and security dialog appear on screen, the "or" locator will match both of them, possibly throwing the "strict mode violation" error. In this case, you can use `locator.first` to only match one of them.

Sync Async

```
new_email = page.get_by_role("button", name="New")
dialog = page.get_by_text("Confirm security settings")
expect(new_email.or_(dialog).first).to_be_visible()
if (dialog.is_visible()):
    page.get_by_role("button", name="Dismiss").click()
new_email.click()
```

Matching only visible elements

NOTE

It's usually better to find a more reliable way to uniquely identify the element instead of checking the visibility.

Consider a page with two buttons, the first invisible and the 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").locator("visible=true").click()
```

Lists

Count items in a list

You can assert locators in order to count the items in a list.

For example, consider the following DOM structure:



The screenshot shows a web browser window with the address bar displaying 'http://localhost:3000'. The browser content is split into two panels. The left panel displays a bulleted list of fruits: 'apple', 'banana', and 'orange'. The right panel displays the corresponding HTML DOM structure for this list, which is an unordered list containing three list items: 'apple', 'banana', and 'orange'.

```
• apple
• banana
• orange
```

```
<ul>
  <li>apple</li>
  <li>banana</li>
  <li>orange</li>
</ul>
```

Use the count assertion to ensure that the list has 3 items.

Sync Async

```
expect(page.get_by_role("listitem")).to_have_count(3)
```

Assert all text in a list

You can assert locators in order to find all the text in a list.

For example, consider the following DOM structure:



Use `expect(locator).to_have_text()` to ensure that the list has the text "apple", "banana" and "orange".

Sync Async

```
expect(page.get_by_role("listitem")).to_have_text(["apple", "banana", "orange"])
```

Get a specific item

There are many ways to get a specific item in a list.

Get by text

Use the `page.get_by_text()` method to locate an element in a list by its text content and then click on it.

For example, consider the following DOM structure:



Locate an item by its text content and click it.

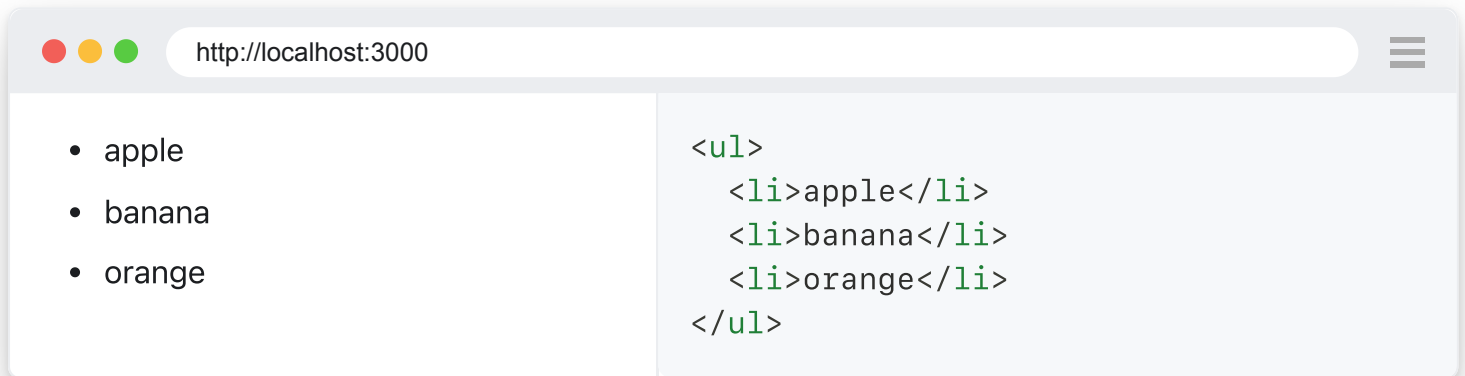
Sync Async

```
page.get_by_text("orange").click()
```

Filter by text

Use the `locator.filter()` to locate a specific item in a list.

For example, consider the following DOM structure:



The screenshot shows a web browser window with the address bar displaying `http://localhost:3000`. The browser content displays a bulleted list of fruits: apple, banana, and orange. To the right of the list, the DOM structure is shown as a code block:

```
<ul>
  <li>apple</li>
  <li>banana</li>
  <li>orange</li>
</ul>
```

Locate an item by the role of "listitem" and then filter by the text of "orange" and then click it.

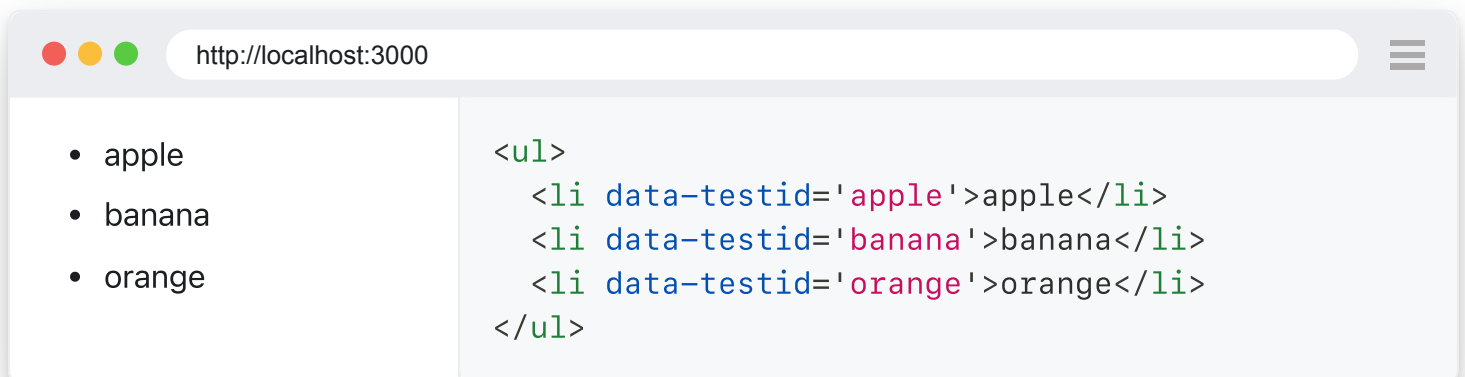
Sync **Async**

```
page.get_by_role("listitem").filter(has_text="orange").click()
```

Get by test id

Use the `page.get_by_test_id()` method to locate an element in a list. You may need to modify the html and add a test id if you don't already have a test id.

For example, consider the following DOM structure:



The screenshot shows a web browser window with the address bar displaying `http://localhost:3000`. The browser content displays a bulleted list of fruits: apple, banana, and orange. To the right of the list, the DOM structure is shown as a code block:

```
<ul>
  <li data-testid='apple'>apple</li>
  <li data-testid='banana'>banana</li>
  <li data-testid='orange'>orange</li>
</ul>
```

Locate an item by its test id of "orange" and then click it.

Sync

Async

```
page.get_by_test_id("orange").click()
```

Get by nth item

If you have a list of identical elements, and the only way to distinguish between them is the order, you can choose a specific element from a list with `locator.first`, `locator.last` or `locator.nth()`.

Sync

Async

```
banana = page.get_by_role("listitem").nth(1)
```

However, use this method with caution. Often times, the page might change, and the locator will point to a completely different element from the one you expected. Instead, try to come up with a unique locator that will pass the `strictness criteria`.

Chaining filters

When you have elements with various similarities, you can use the `locator.filter()` method to select the right one. You can also chain multiple filters to narrow down the selection.

For example, consider the following DOM structure:

The screenshot shows a web browser window with the address bar displaying `http://localhost:3000`. The browser content is divided into two panels. The left panel displays a list of four items, each with a name and a button:

- John (Say hello)
- Mary (Say hello)
- John (Say goodbye)
- Mary (Say goodbye)

The right panel displays the corresponding DOM structure as a code block:

```
<ul>
  <li>
    <div>John</div>
    <div><button>Say hello</button></div>
  </li>
  <li>
    <div>Mary</div>
    <div><button>Say hello</button></div>
  </li>
  <li>
    <div>John</div>
    <div><button>Say goodbye</button></div>
  </li>
  <li>
```

```
<div>Mary</div>
<div><button>Say goodbye</button></div>
</li>
</ul>
```

To take a screenshot of the row with "Mary" and "Say goodbye":

Sync **Async**

```
row_locator = page.get_by_role("listitem")

row_locator.filter(has_text="Mary").filter(
    has=page.get_by_role("button", name="Say goodbye")
).screenshot(path="screenshot.png")
```

You should now have a "screenshot.png" file in your project's root directory.

Rare use cases

Do something with each element in the list

Iterate elements:

Sync **Async**

```
for row in page.get_by_role("listitem").all():
    print(row.text_content())
```

Iterate using regular for loop:

Sync **Async**

```
rows = page.get_by_role("listitem")
count = rows.count()
for i in range(count):
    print(rows.nth(i).text_content())
```

Evaluate in the page

The code inside `locator.evaluate_all()` runs in the page, you can call any DOM apis there.

Sync **Async**

```
rows = page.get_by_role("listitem")
texts = rows.evaluate_all("list => list.map(element =>
  element.textContent)")
```

Strictness

Locators are strict. This means that all operations on locators that imply some target DOM element will throw an exception if more than one element matches. For example, the following call throws if there are several buttons in the DOM:

Throws an error if more than one

Sync **Async**

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

On the other hand, Playwright understands when you perform a multiple-element operation, so the following call works perfectly fine when the locator resolves to multiple elements.

Works fine with multiple elements

Sync **Async**

```
page.get_by_role("button").count()
```

You can explicitly opt-out from strictness check by telling Playwright which element to use when multiple elements match, through `locator.first`, `locator.last`, and `locator.nth()`. These methods are **not recommended** because when your page changes, Playwright may click on an element you did not intend. Instead, follow best practices above to create a locator that uniquely identifies the target element.

More Locators

For less commonly used locators, look at the [other locators](#) guide.