PLAYWRIGHT CHEAT SHEET TYPESCRIPT AND JAVA SCRIPT

Installation

npm init playwright@latest

System requirements

- Node.js 18+
- Windows 10+, Windows Server 2016+ or Windows Subsystem for Linux (WSL).
- macOS 13 Ventura, or later.
- Debian 12, Ubuntu 22.04, Ubuntu 24.04, on x86-64 and arm64 architecture.

What's Installed

Playwright will download the browsers needed as well as create the following files.

```
playwright.config.ts
package.json
package-lock.json
tests/
  example.spec.ts
tests-examples/
  demo-todo-app.spec.ts
```

The playwright.config is where you can add configuration for Playwright including modifying which browsers you would like to run Playwright on. If you are running tests inside an already existing project then dependencies will be added directly to your `package.json`.

The `tests` folder contains a basic example test to help you get started with testing. For a more detailed example check out the `tests-examples` folder which contains tests written to test a todo app.

Running the Example Test	npx playwright test
HTML Test Reports	npx playwright show-report
Running the Example Test in UI Mode	npx playwright test --ui
Updating Playwright	npm install -D @playwright/test@latest # Also download new browser binaries and their dependencies: npx playwright install --with-deps
Running Codegen	npx playwright codegen demo.playwright.dev/todomvc
Run tests in headed mode	npx playwright test –headed
Run tests on different browsers	npx playwright test --project webkit npx playwright test --project webkit --project firefox
Run specific tests	npx playwright test landing-page.spec.ts npx playwright test tests/todo-page/ tests/landing-page/ npx playwright test landing login
Run last failed tests	npx playwright test --last-failed
Debug tests in UI mode	npx playwright test –ui
Debug tests with the Playwright Inspector	npx playwright test –debug npx playwright test example.spec.ts --debug

Trace viewer

Recording a Trace

By default the playwright.config file will contain the configuration needed to create a `trace.zip` file for each test. Traces are setup to run `on-first-retry` meaning they will be run on the first retry of a failed test. Also `retries` are set to 2 when running on CI and 0 locally. This means the traces will be recorded on the first retry of a failed test but not on the first run and not on the second retry.

playwright.config.ts

```
import { defineConfig } from '@playwright/test';
export default defineConfig({
  retries: process.env.CI ? 2 : 0, // set to 2 when running on CI
  // ...
  use: {
    trace: 'on-first-retry', // record traces on first retry of each test
  },
});
```

To learn more about available options to record a trace check out our detailed guide on Trace Viewer.

Traces are normally run in a Continuous Integration(CI) environment, because locally you can use UI Mode for developing and debugging tests. However, if you want to run traces locally without using UI Mode, you can force tracing to be on with `--trace on`.

```
npx playwright test --trace on
```

Actions

Text input

Using locator.fill() is the easiest way to fill out the form fields. It focuses the element and triggers an input event with the entered text. It works for <input>, <textarea> and [contenteditable] elements.

```
// Text input
await page.getByRole('textbox').fill('Peter');

// Date input
await page.getByLabel('Birth date').fill('2020-02-02');

// Time input
await page.getByLabel('Appointment time').fill('13:15');

// Local datetime input
await page.getByLabel('Local time').fill('2020-03-02T05:15');
```

Checkboxes and radio buttons

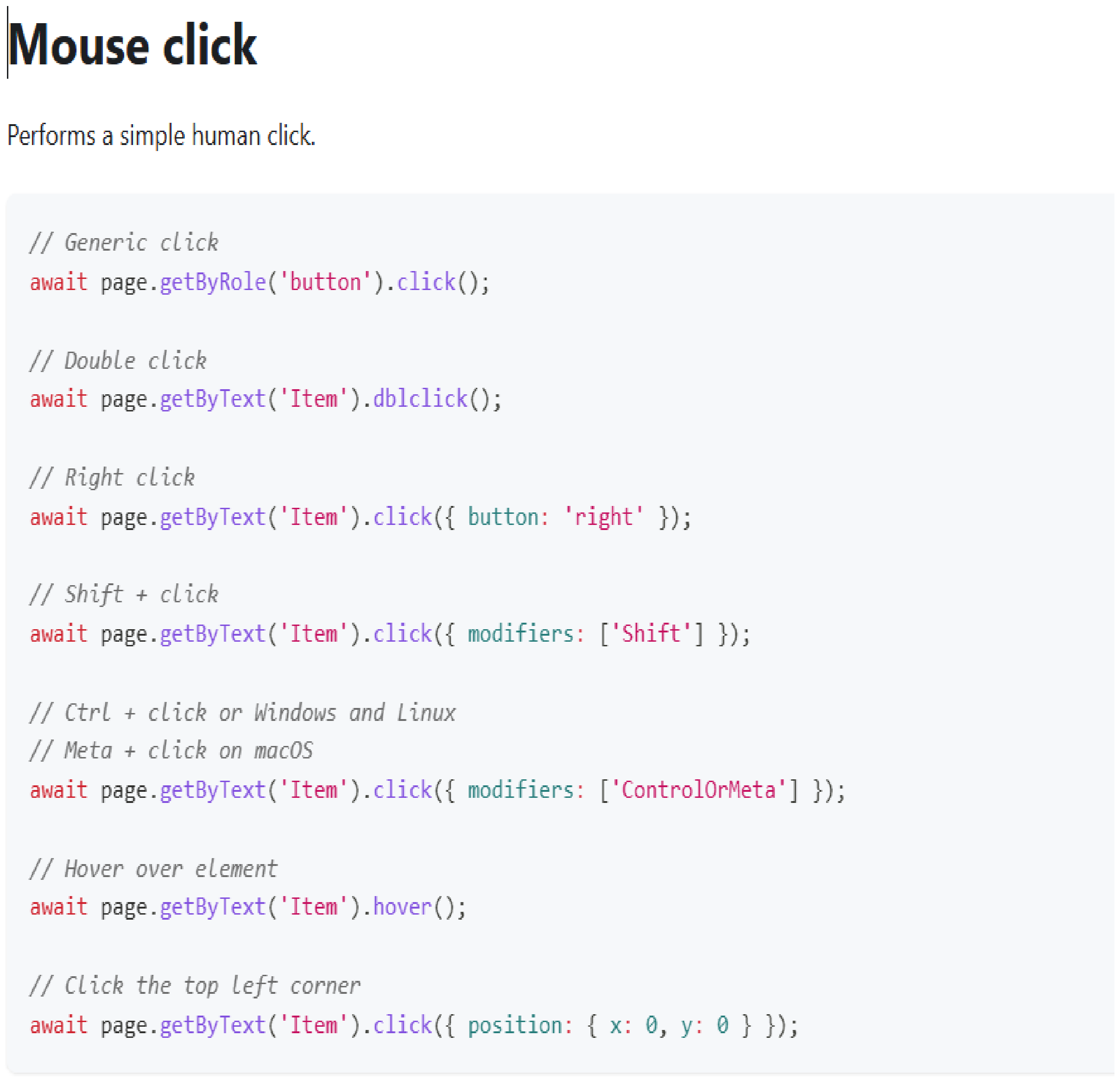
Using locator.setChecked() is the easiest way to check and uncheck a checkbox or a radio button. This method can be used with input[type=checkbox], input[type=radio] and [role=checkbox] elements.

```
// Check the checkbox
await page.getByLabel('I agree to the terms above').check();

// Assert the checked state
expect(page.getByLabel('Subscribe to newsletter')).toBeChecked();

// Select the radio button
await page.getByLabel('XL').check();
```

Select options

Selects one or multiple options in the <select> element with locator.selectOption(). You can specify option value, or label to select. Multiple options can be selected.

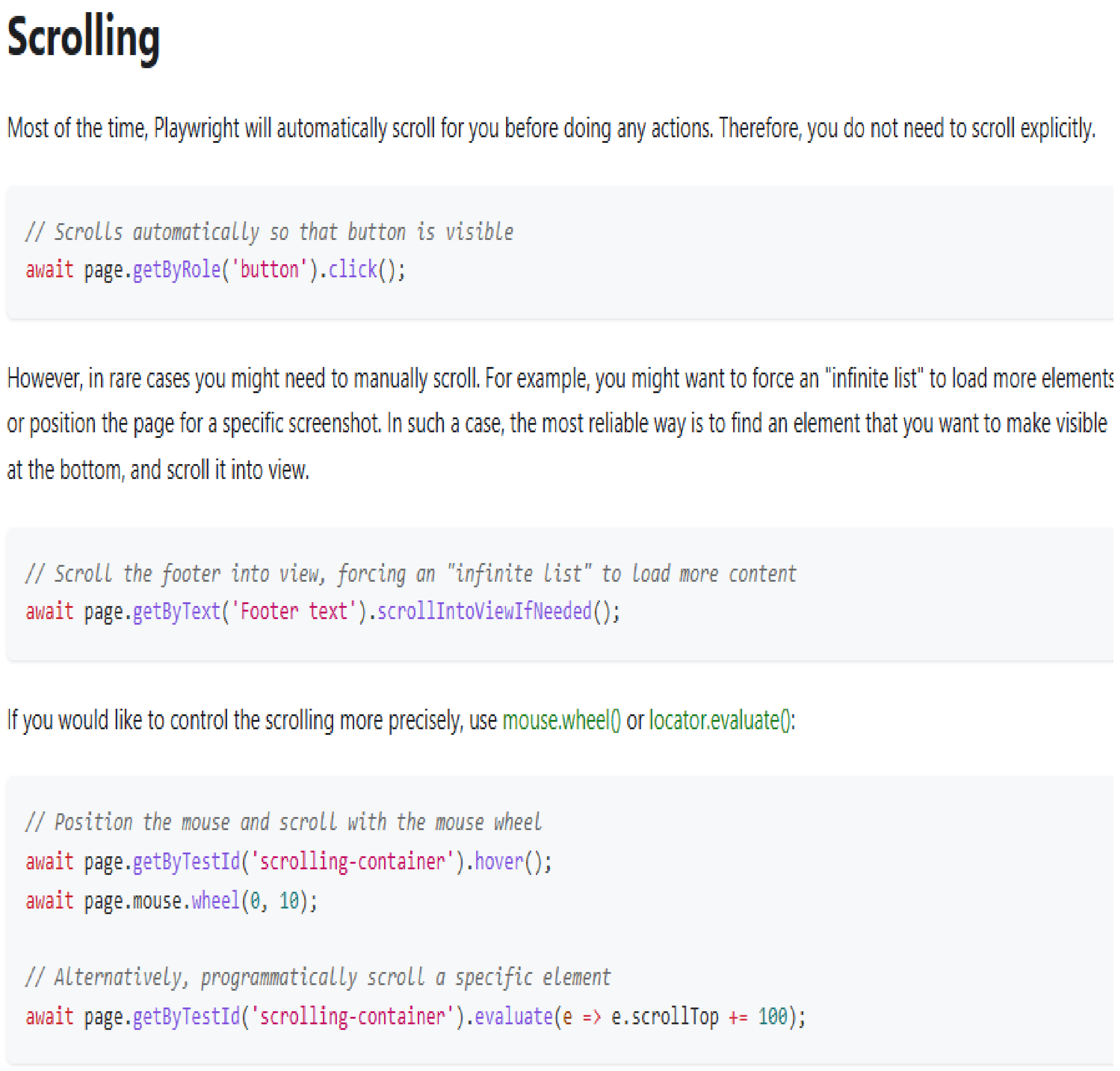
```
// Single selection matching the value or label
await page.getByLabel('Choose a color').selectOption('blue');

// Single selection matching the label
await page.getByLabel('Choose a color').selectOption({ label: 'Blue' });

// Multiple selected items
await page.getByLabel('Choose multiple colors').selectOption(['red', 'green', 'blue']);
```



Mouse click

Performs a simple human click.

```
// Generic click
await page.getByRole('button').click();

// Double click
await page.getByText('Item').dblclick();

// Right click
await page.getByText('Item').click({ button: 'right' });

// Shift + click
await page.getByText('Item').click({ modifiers: ['Shift'] });

// Ctrl + click or Windows and Linux
// Meta + click on macOS
await page.getByText('Item').click({ modifiers: ['ControlOrMeta'] });

// Hover over element
await page.getByText('Item').hover();

// Click the top left corner
await page.getByText('Item').click({ position: { x: 0, y: 0 } });
```



Scrolling

Most of the time, Playwright will automatically scroll for you before doing any actions. Therefore, you do not need to scroll explicitly.

```
// Scrolls automatically so that button is visible
await page.getByRole('button').click();
```

However, in rare cases you might need to manually scroll. For example, you might want to force an "infinite list" to load more elements or position the page for a specific screenshot. In such a case, the most reliable way is to find an element that you want to make visible at the bottom, and scroll it into view.

```
// Scroll the footer into view, forcing an "infinite list" to load more content
await page.getByText('Footer text').scrollIntoViewIfNeeded();
```

If you would like to control the scrolling more precisely, use mouse.wheel() or locator.evaluate():

```
// Position the mouse and scroll with the mouse wheel
await page.getByTestId('scrolling-container').hover();
await page.mouse.wheel(0, 10);

// Alternatively, programmatically scroll a specific element
await page.getByTestId('scrolling-container').evaluate(e => e.scrollTop += 100);
```

Keys and shortcuts

```
// Hit Enter
await page.getByText('Submit').press('Enter');

// Dispatch Control+Right
await page.getByRole('textbox').press('Control+ArrowRight');

// Press $ sign on keyboard
await page.getByRole('textbox').press('$');
```

The locator.press() method focuses the selected element and produces a single keystroke. It accepts the logical key names that are emitted in the keyboardEvent.key property of the keyboard events:

```
Backquote, Minus, Equal, Backslash, Backspace, Tab, Delete, Escape,
ArrowDown, End, Enter, Home, Insert, PageDown, PageUp, ArrowRight,
ArrowUp, F1 - F12, Digit0 - Digit9, KeyA - KeyZ, etc.
```

Upload files

You can select input files for upload using the locator.setInputFiles() method. It expects first argument to point to an input element with the type `"file"`. Multiple files can be passed in the array. If some of the file paths are relative, they are resolved relative to the current working directory. Empty array clears the selected files.

```
// Select one file
await page.getByLabel('Upload file').setInputFiles(path.join(__dirname, 'myfile.pdf'));

// Select multiple files
await page.getByLabel('Upload files').setInputFiles([
  path.join(__dirname, 'file1.txt'),
  path.join(__dirname, 'file2.txt'),
]);

// Select a directory
await page.getByLabel('Upload directory').setInputFiles(path.join(__dirname, 'mydir'));

// Remove all the selected files
await page.getByLabel('Upload file').setInputFiles([]);

// Upload buffer from memory
await page.getByLabel('Upload file').setInputFiles({
  name: 'file.txt',
  mimeType: 'text/plain',
  buffer: Buffer.from('this is test')
});
```

If you don't have input element in hand (it is created dynamically), you can handle the page.on('filechooser') event or use a corresponding waiting method upon your action:

```
// Start waiting for file chooser before clicking. Note no await.
const fileChooserPromise = page.waitForEvent('filechooser');
await page.getByLabel('Upload file').click();
const fileChooser = await fileChooserPromise;
await fileChooser.setFiles(path.join(__dirname, 'myfile.pdf'));
```

Focus element

For the dynamic pages that handle focus events, you can focus the given element with locator.focus().

```
await page.getByLabel('Password').focus();
```

Drag and Drop

You can perform drag&drop operation with locator.dragTo(). This method will:

- Hover the element that will be dragged.
- Press left mouse button.
- Move mouse to the element that will receive the drop.
- Release left mouse button.

```
await page.locator('#item-to-be-dragged').dragTo(page.locator('#item-to-drop-at'));
```

Dragging manually

If you want precise control over the drag operation, use lower-level methods like locator.hover(), mouse.down(), mouse.move() and mouse.up().

```
await page.locator('#item-to-be-dragged').hover();
await page.mouse.down();
await page.locator('#item-to-drop-at').hover();
await page.mouse.up();
```

Assertions

Auto-retrying assertions

The following assertions will retry until the assertion passes, or the assertion timeout is reached. Note that retrying assertions are async, so you must `await` them.

Assertion	Description
await expect(locator).toBeAttached()	Element is attached
await expect(locator).toBeChecked()	Checkbox is checked
await expect(locator).toBeDisabled()	Element is disabled
await expect(locator).toBeEditable()	Element is editable
await expect(locator).toBeEmpty()	Container is empty
await expect(locator).toBeEnabled()	Element is enabled
await expect(locator).toBeFocused()	Element is focused
await expect(locator).toBeHidden()	Element is not visible
await expect(locator).toBeInViewport()	Element intersects viewport
await expect(locator).toBeVisible()	Element is visible
await expect(locator).toContainText()	Element contains text
await expect(locator).toHaveAccessibleDescription()	Element has a matching accessible description
await expect(locator).toHaveAccessibleName()	Element has a matching accessible name
await expect(locator).toHaveAttribute()	Element has a DOM attribute
await expect(locator).toHaveClass()	Element has a class property
await expect(locator).toHaveCount()	List has exact number of children
await expect(locator).toHaveCSS()	Element has CSS property
await expect(locator).toHaveId()	Element has an ID
await expect(locator).toHaveJSProperty()	Element has a JavaScript property
await expect(locator).toHaveRole()	Element has a specific ARIA role
await expect(locator).toHaveScreenshot()	Element has a screenshot
await expect(locator).toHaveText()	Element matches text
await expect(locator).toHaveValue()	Input has a value
await expect(locator).toHaveValues()	Select has options selected
await expect(page).toHaveScreenshot()	Page has a screenshot
await expect(page).toHaveTitle()	Page has a title
await expect(page).toHaveURL()	Page has a URL
await expect(response).toBeOK()	Response has an OK status

Non-retrying assertions

These assertions allow to test any conditions, but do not auto-retry. Most of the time, web pages show information asynchronously, and using non-retrying assertions can lead to a flaky test.

Prefer auto-retrying assertions whenever possible. For more complex assertions that need to be retried, use `expect.poll` or `expect.toPass`.

Assertion	Description
expect(value).toBe()	Value is the same
expect(value).toBeCloseTo()	Number is approximately equal
expect(value).toBeDefined()	Value is not `undefined`
expect(value).toBeFalsy()	Value is falsy, e.g. `false`, `0`, `null`, etc.
expect(value).toBeGreaterThan()	Number is more than
expect(value).toBeGreaterThanOrEqual()	Number is more than or equal
expect(value).toBeInstanceOf()	Object is an instance of a class
expect(value).toBeLessThan()	Number is less than
expect(value).toBeLessThanOrEqual()	Number is less than or equal
expect(value).toBeNaN()	Value is `NaN`
expect(value).toContain()	String contains a substring
expect(value).toContain()	Array or set contains an element
expect(value).toContainEqual()	Array or set contains a similar element
expect(value).toEqual()	Value is similar - deep equality and pattern matching
expect(value).toHaveLength()	Array or string has length
expect(value).toHaveProperty()	Object has a property
expect(value).toMatch()	String matches a regular expression
expect(value).toMatchObject()	Object contains specified properties
expect(value).toStrictEqual()	Value is similar, including property types
expect(value).toThrow()	Function throws an error
expect(value).any()	Matches any instance of a class/primitive
expect(value).anything()	Matches anything
expect(value).arrayContaining()	Array contains specific elements
expect(value).closeTo()	Number is approximately equal
expect(value).objectContaining()	Object contains specific properties
expect(value).stringContaining()	String contains a substring
expect(value).stringMatching()	String matches a regular expression

Negating matchers

In general, we can expect the opposite to be true by adding a `.not` to the front of the matchers:

```
expect(value).not.toEqual(0);
await expect(locator).not.toContainText('some text');
```

Soft assertions

By default, failed assertion will terminate test execution. Playwright also supports *soft assertions*: failed soft assertions **do not** terminate test execution, but mark the test as failed.

```
// Make a few checks that will not stop the test when failed...
await expect.soft(page.getByTestId('status')).toHaveText('Success');
await expect.soft(page.getByTestId('eta')).toHaveText('1 day');

// ... and continue the test to check more things.
await page.getByRole('link', { name: 'next page' }).click();
await expect.soft(page.getByRole('heading', { name: 'Make another order' })).toBeVisible();
```

At any point during test execution, you can check whether there were any soft assertion failures:

```
// Make a few checks that will not stop the test when failed...
await expect.soft(page.getByTestId('status')).toHaveText('Success');
await expect.soft(page.getByTestId('eta')).toHaveText('1 day');

// Avoid running further if there were soft assertion failures.
expect(test.info().errors).toHaveLength(0);
```

Note that soft assertions only work with Playwright test runner.

Custom expect message

You can specify a custom expect message as a second argument to the `expect` function, for example:

```
await expect(page.getByText('Name'), 'should be logged in').toBeVisible();
```

This message will be shown in reporters, both for passing and failing expects, providing more context about the assertion.

When expect passes, you might see a successful step like this:

```
✅ should be logged in    @example.spec.ts:18
```

When expect fails, the error would look like this:

```
    Error: should be logged in

    Call log:
      - expect.toBeVisible with timeout 5000ms
      - waiting for "getByText('Name')"


      2 |
      3 | test('example test', async({ page }) => {
    > 4 |   await expect(page.getByText('Name'), 'should be logged in').toBeVisible();
        |                                                                  ^
      5 | });
      6 |
```

Soft assertions also support custom message:

```
expect.soft(value, 'my soft assertion').toBe(56);
```

Dialogs

Introduction

Playwright can interact with the web page dialogs such as `alert`, `confirm`, `prompt` as well as `beforeunload` confirmation. For print dialogs, see Print.

alert(), confirm(), prompt() dialogs

By default, dialogs are auto-dismissed by Playwright, so you don't have to handle them. However, you can register a dialog handler before the action that triggers the dialog to either dialog.accept() or dialog.dismiss() it.

```
page.on('dialog', dialog => dialog.accept());
await page.getByRole('button').click();
```

ⓘ **NOTE**

page.on('dialog') listener **must handle** the dialog. Otherwise your action will stall, be it locator.click() or something else. That's because dialogs in Web are modals and therefore block further page execution until they are handled.

```
page.on('dialog', dialog => console.log(dialog.message()));
await page.getByRole('button').click(); // Will hang here
```

ⓘ **NOTE**

If there is no listener for page.on('dialog'), all dialogs are automatically dismissed.

beforeunload dialog

When page.close() is invoked with the truthy runBeforeUnload value, the page runs its unload handlers. This is the only case when page.close() does not wait for the page to actually close, because it might be that the page stays open in the end of the operation.

You can register a dialog handler to handle the `beforeunload` dialog yourself:

```
page.on('dialog', async dialog => {
  assert(dialog.type() === 'beforeunload');
  await dialog.dismiss();
});
await page.close({ runBeforeUnload: true });
```

Print dialogs

In order to assert that a print dialog via `window.print` was triggered, you can use the following snippet:

```
await page.goto('<url>');

await page.evaluate('(() => {window.waitForPrintDialog = new Promise(f => window.print = f);})()');
await page.getByText('Print it!').click();

await page.waitForFunction('window.waitForPrintDialog');
```

This will wait for the print dialog to be opened after the button is clicked. Make sure to evaluate the script before clicking the button / after the page is loaded.

Downloads

Introduction

For every attachment downloaded by the page, page.on('download') event is emitted. All these attachments are downloaded into a temporary folder. You can obtain the download url, file name and payload stream using the Download object from the event.

You can specify where to persist downloaded files using the downloadsPath option in browserType.launch().

> **NOTE**
>
> Downloaded files are deleted when the browser context that produced them is closed.

Here is the simplest way to handle the file download:

```
// Start waiting for download before clicking. Note no await.
const downloadPromise = page.waitForEvent('download');
await page.getByText('Download file').click();
const download = await downloadPromise;

// Wait for the download process to complete and save the downloaded file somewhere.
await download.saveAs('/path/to/save/at/' + download.suggestedFilename());
```

Variations

If you have no idea what initiates the download, you can still handle the event:

```
page.on('download', download => download.path().then(console.log));
```

Note that handling the event forks the control flow and makes the script harder to follow. Your scenario might end while you are downloading a file since your main control flow is not awaiting for this operation to resolve.

Waiting for event

Most of the time, scripts will need to wait for a particular event to happen. Below are some of the typical event awaiting patterns.

Wait for a request with the specified url using page.waitForRequest():

```
// Start waiting for request before goto. Note no await.
const requestPromise = page.waitForRequest('**/*logo*.png');
await page.goto('https://wikipedia.org');
const request = await requestPromise;
console.log(request.url());
```

Wait for popup window:

```
// Start waiting for popup before clicking. Note no await.
const popupPromise = page.waitForEvent('popup');
await page.getByText('open the popup').click();
const popup = await popupPromise;
await popup.goto('https://wikipedia.org');
```

Adding/removing event listener

Sometimes, events happen in random time and instead of waiting for them, they need to be handled. Playwright supports traditional language mechanisms for subscribing and unsubscribing from the events:

```
page.on('request', request => console.log(`Request sent: ${request.url()}`));
const listener = request => console.log(`Request finished: ${request.url()}`);
page.on('requestfinished', listener);
await page.goto('https://wikipedia.org');

page.off('requestfinished', listener);
await page.goto('https://www.openstreetmap.org/');
```

Frames

Introduction

A Page can have one or more Frame objects attached to it. Each page has a main frame and page-level interactions (like `click`) are assumed to operate in the main frame.

A page can have additional frames attached with the `iframe` HTML tag. These frames can be accessed for interactions inside the frame.

```
// Locate element inside frame
const username = await page.frameLocator('.frame-class').getByLabel('User Name');
await username.fill('John');
```

Frame objects

One can access frame objects using the page.frame() API:

```
// Get frame using the frame's name attribute
const frame = page.frame('frame-login');

// Get frame using frame's URL
const frame = page.frame({ url: /.*domain.*/ });

// Interact with the frame
await frame.fill('#username-input', 'John');
```

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

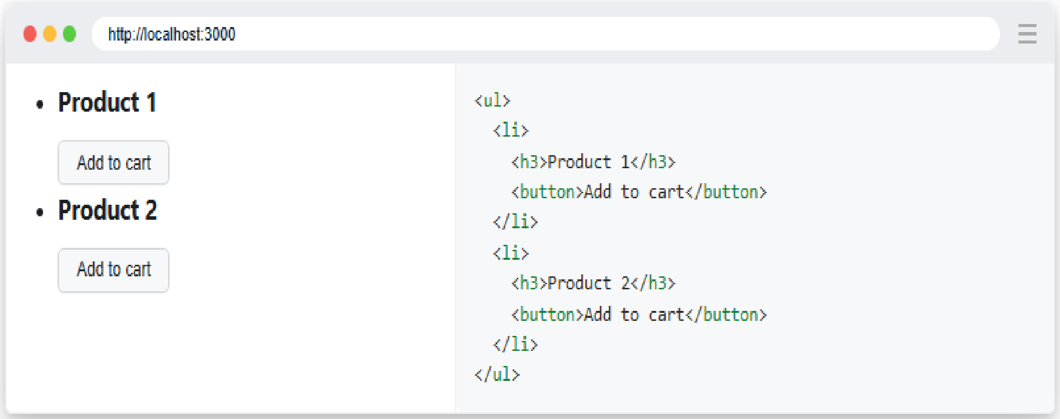
These are the recommended built-in locators.

- page.getByRole() to locate by explicit and implicit accessibility attributes.
- page.getByText() to locate by text content.
- page.getByLabel() to locate a form control by associated label's text.
- page.getByPlaceholder() to locate an input by placeholder.
- page.getByAltText() to locate an element, usually image, by its text alternative.
- page.getByTitle() to locate an element by its title attribute.
- page.getByTestId() to locate an element based on its `data-testid` attribute (other attributes can be configured).

```
await page.getByLabel('User Name').fill('John');

await page.getByLabel('Password').fill('secret-password');

await page.getByRole('button', { name: 'Sign in' }).click();

await expect(page.getByText('Welcome, John!')).toBeVisible();
```

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.



```
<ul>
  <li>
    <h3>Product 1</h3>
    <button>Add to cart</button>
  </li>
  <li>
    <h3>Product 2</h3>
    <button>Add to cart</button>
  </li>
</ul>
```

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.

```
await page
    .getByRole('listitem')
    .filter({ hasText: 'Product 2' })
    .getByRole('button', { name: 'Add to cart' })
    .click();
```

Use a regular expression:

```
await page
    .getByRole('listitem')
    .filter({ hasText: /Product 2/ })
    .getByRole('button', { name: 'Add to cart' })
    .click();
```

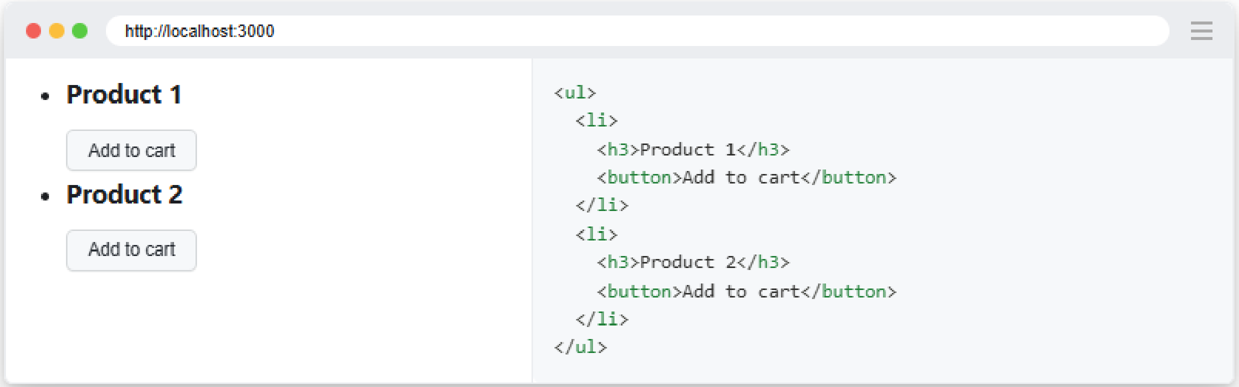
Filter by not having text

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

```
// 5 in-stock items
await expect(page.getByRole('listitem').filter({ hasNotText: 'Out of stock' })).toHaveCount(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.getByRole(), locator.getByTestId(), locator.getByText() etc.



```
<ul>
  <li>
    <h3>Product 1</h3>
    <button>Add to cart</button>
  </li>
  <li>
    <h3>Product 2</h3>
    <button>Add to cart</button>
  </li>
</ul>
```

```
await page
    .getByRole('listitem')
    .filter({ has: page.getByRole('heading', { name: 'Product 2' }) })
    .getByRole('button', { name: 'Add to cart' })
    .click();
```

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

```
await expect(page
    .getByRole('listitem')
    .filter({ has: page.getByRole('heading', { name: 'Product 2' }) }))
    .toHaveCount(1);
```

Locator operators

Matching inside a locator

You can chain methods that create a locator, like page.getByText() or locator.getByRole(), 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".

```
const product = page.getByRole('listitem').filter({ hasText: 'Product 2' });

await product.getByRole('button', { name: 'Add to cart' }).click();

await expect(product).toHaveCount(1);
```

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

```
const saveButton = page.getByRole('button', { name: 'Save' });
// ...
const dialog = page.getByTestId('settings-dialog');
await dialog.locator(saveButton).click();
```

Matching two locators simultaneously

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

```
const button = page.getByRole('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.

```
const newEmail = page.getByRole('button', { name: 'New' });
const dialog = page.getByText('Confirm security settings');
await expect(newEmail.or(dialog).first()).toBeVisible();
if (await dialog.isVisible())
  await page.getByRole('button', { name: 'Dismiss' }).click();
await newEmail.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:

  ```
  await page.locator('button').click();
  ```

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

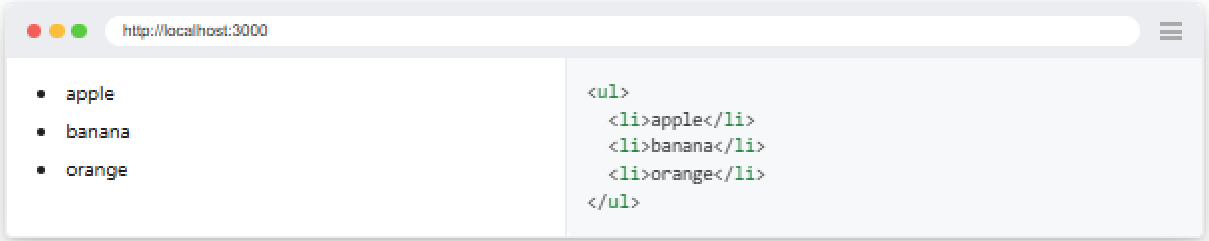
  ```
  await page.locator('button').locator('visible=true').click();
  ```

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:



- apple
- banana
- orange

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

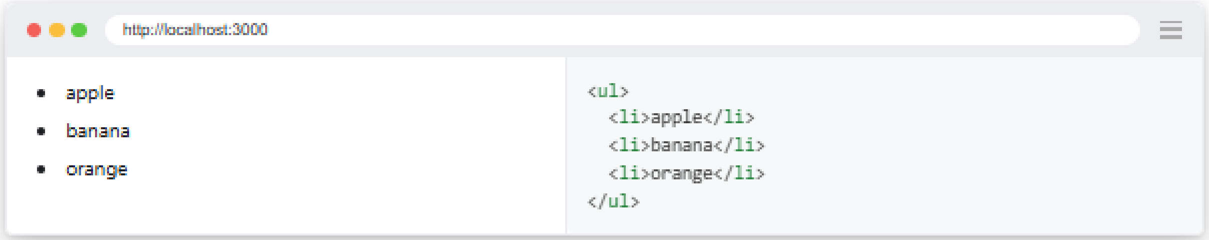
Use the count assertion to ensure that the list has 3 items.

```js
await expect(page.getByRole('listitem')).toHaveCount(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:



- apple
- banana
- orange

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

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

```js
await expect(page
    .getByRole('listitem'))
    .toHaveText(['apple', 'banana', 'orange']);
```

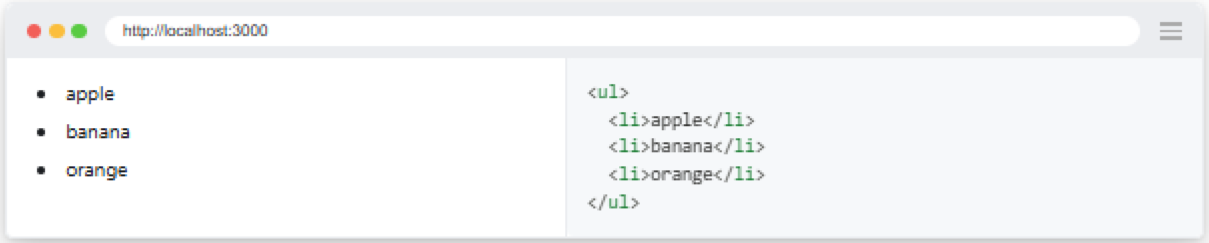
Get a specific item

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

Get by text

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

For example, consider the following DOM structure:



- apple
- banana
- orange

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

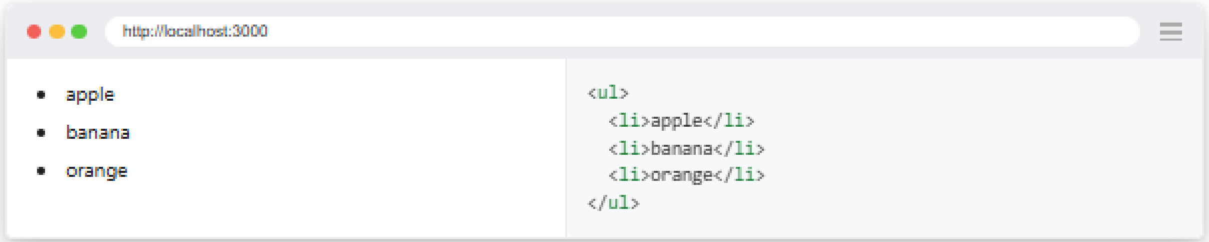
Locate an item by its text content and click it.

```js
await page.getByText('orange').click();
```

Filter by text

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

For example, consider the following DOM structure:



- apple
- banana
- orange

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

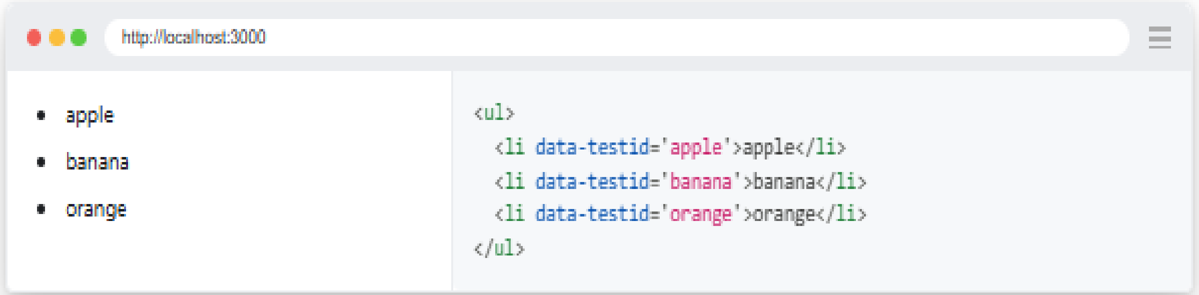
Locate an item by the role of "listitem" and then filter by the text of "orange" and then click it.

```js
await page
    .getByRole('listitem')
    .filter({ hasText: 'orange' })
    .click();
```

Get by test id

Use the page.getByTestId() 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:



- apple
- banana
- orange

```html
<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.

```js
await page.getByTestId('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().

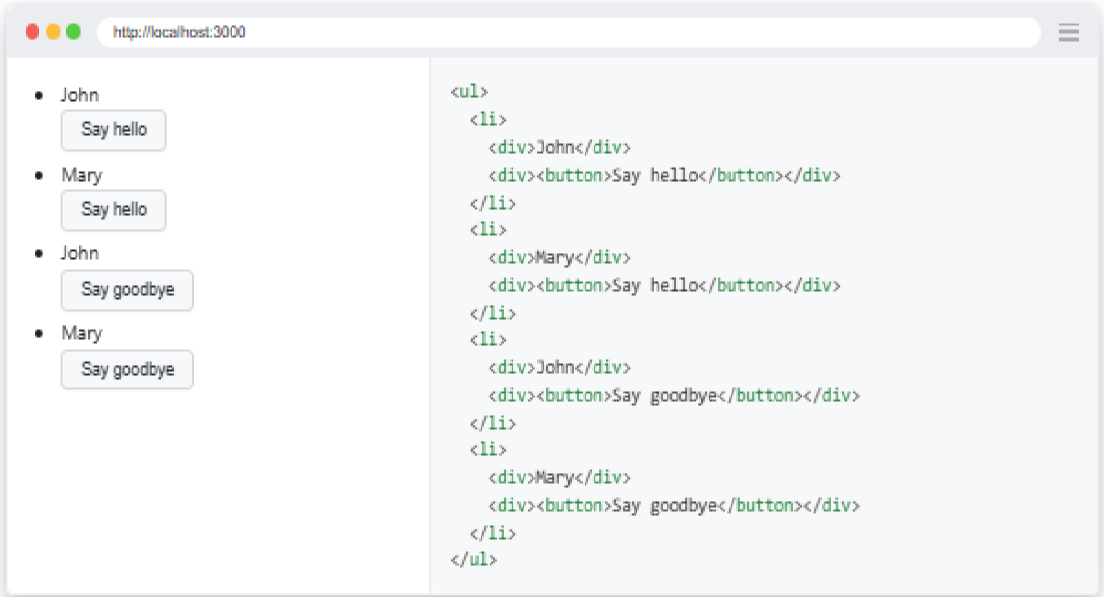
```js
const banana = await page.getByRole('listitem').nth(1);
```

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:



- John
- Mary
- John
- Mary

```html
<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":

```js
const rowLocator = page.getByRole('listitem');

await rowLocator
    .filter({ hasText: 'Mary' })
    .filter({ has: page.getByRole('button', { name: 'Say goodbye' }) })
    .screenshot({ path: 'screenshot.png' });
```

Pages

Each BrowserContext can have multiple pages. A Page refers to a single tab or a popup window within a browser context. It should be used to navigate to URLs and interact with the page content.

```
// Create a page.
const page = await context.newPage();

// Navigate explicitly, similar to entering a URL in the browser.
await page.goto('http://example.com');
// Fill an input.
await page.locator('#search').fill('query');

// Navigate implicitly by clicking a link.
await page.locator('#submit').click();
// Expect a new url.
console.log(page.url());
```

Multiple pages

Each browser context can host multiple pages (tabs).

- Each page behaves like a focused, active page. Bringing the page to front is not required.
- Pages inside a context respect context-level emulation, like viewport sizes, custom network routes or browser locale.

```
// Create two pages
const pageOne = await context.newPage();
const pageTwo = await context.newPage();

// Get pages of a browser context
const allPages = context.pages();
```

Handling new pages

The `page` event on browser contexts can be used to get new pages that are created in the context. This can be used to handle new pages opened by `target="_blank"` links.

```
// Start waiting for new page before clicking. Note no await.
const pagePromise = context.waitForEvent('page');
await page.getByText('open new tab').click();
const newPage = await pagePromise;
// Interact with the new page normally.
await newPage.getByRole('button').click();
console.log(await newPage.title());
```

If the action that triggers the new page is unknown, the following pattern can be used.

```
// Get all new pages (including popups) in the context
context.on('page', async page => {
  await page.waitForLoadState();
  console.log(await page.title());
});
```

Introduction

Here is a quick way to capture a screenshot and save it into a file:

```
await page.screenshot({ path: 'screenshot.png' });
```

Screenshots API accepts many parameters for image format, clip area, quality, etc. Make sure to check them out.

Full page screenshots

Full page screenshot is a screenshot of a full scrollable page, as if you had a very tall screen and the page could fit it entirely.

```
await page.screenshot({ path: 'screenshot.png', fullPage: true });
```

Capture into buffer

Rather than writing into a file, you can get a buffer with the image and post-process it or pass it to a third party pixel diff facility.

```
const buffer = await page.screenshot();
console.log(buffer.toString('base64'));
```

Element screenshot

Sometimes it is useful to take a screenshot of a single element.

```
await page.locator('.header').screenshot({ path: 'screenshot.png' });
```