Actions

Introduction

Playwright can interact with HTML Input elements such as text inputs, checkboxes, radio buttons, select options, mouse clicks, type characters, keys and shortcuts as well as upload files and focus elements.

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 } });
```

Under the hood, this and other pointer-related methods:

- wait for element with given selector to be in DOM
- wait for it to become displayed, i.e. not empty, no `display:none`, no `visibility:hidden`
- wait for it to stop moving, for example, until css transition finishes
- scroll the element into view
- wait for it to receive pointer events at the action point, for example, waits until element becomes non-obscured by other elements
- retry if the element is detached during any of the above checks

Forcing the click

Sometimes, apps use non-trivial logic where hovering the element overlays it with another element that intercepts the click. This behavior is indistinguishable from a bug where element gets covered and the click is dispatched elsewhere. If you know this is taking place, you can bypass the actionability checks and force the click:

```
await page.getByRole('button').click({ force: true });
```

Programmatic click

If you are not interested in testing your app under the real conditions and want to simulate the click by any means possible, you can trigger the `HTMLElement.click()` behavior via simply dispatching a click event on the element with locator.dispatchEvent():

```
await page.getByRole('button').dispatchEvent('click');
```

Type characters

CAUTION

Most of the time, you should input text with locator.fill(). See the Text input section above. You only need to type characters if there is special keyboard handling on the page.

Type into the field character by character, as if it was a user with a real keyboard with locator.pressSequentially().

```
// Press keys one by one
await page.locator('#area').pressSequentially('Hello World!');
```

This method will emit all the necessary keyboard events, with all the `keydown`, `keyup`, `keypress` events in place. You can even specify the optional `delay` between the key presses to simulate real user behavior.

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.
```

- You can alternatively specify a single character you'd like to produce such as `"a"` or `"#"`.
- Following modification shortcuts are also supported: `Shift, Control, Alt, Meta`.

Simple version produces a single character. This character is case-sensitive, so `"a"` and `"A"` will produce different results.

```
// <input id=name>
await page.locator('#name').press('Shift+A');

// <input id=name>
await page.locator('#name').press('Shift+ArrowLeft');
```

Shortcuts such as `"Control+o"` or `"Control+Shift+T"` are supported as well. When specified with the modifier, modifier is pressed and being held while the subsequent key is being pressed.

Note that you still need to specify the capital `A` in `Shift-A` to produce the capital character. `Shift-a` produces a lower-case one as if you had the `CapsLock` toggled.

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();
```

NOTE

If your page relies on the `dragover` event being dispatched, you need at least two mouse moves to trigger it in all browsers. To reliably issue the second mouse move, repeat your mouse.move() or locator.hover() twice. The sequence of operations would be: hover the drag element, mouse down, hover the drop element, hover the drop element second time, mouse up.

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);
```

Previous
Accessibility testing

Next
Assertions

- Introduction
- Text input
- Checkboxes and radio buttons
- Select options
- Mouse click
- Type characters
- Keys and shortcuts
- Upload files
- Focus element
- Drag and Drop
  - Dragging manually
- Scrolling

Learn

- Getting started
- Playwright Training
- Learn Videos
- Feature Videos

Community

- Stack Overflow
- Discord
- Twitter

- LinkedIn

More

- GitHub
- YouTube
- Blog
- Ambassadors