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.

Sync Async

```
# Text input
page.get_by_role("textbox").fill("Peter")

# Date input
page.get_by_label("Birth date").fill("2020-02-02")

# Time input
page.get_by_label("Appointment time").fill("13:15")

# Local datetime input
page.get_by_label("Local time").fill("2020-03-02T05:15")
```

Checkboxes and radio buttons

Using locator.set_checked() 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.

Sync Async

```
# Check the checkbox
page.get_by_label('I agree to the terms above').check()

# Assert the checked state
expect(page.get_by_label('Subscribe to newsletter')).to_be_checked()

# Select the radio button
page.get_by_label('XL').check()
```

Select options

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

Sync Async

```
# Single selection matching the value or label
page.get_by_label('Choose a color').select_option('blue')

# Single selection matching the label
page.get_by_label('Choose a color').select_option(label='Blue')

# Multiple selected items
page.get_by_label('Choose multiple colors').select_option(['red', 'green',
'blue'])
```

Mouse click

Performs a simple human click.

Sync Async

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

# Double click
page.get_by_text("Item").dblclick()

# Right click
page.get_by_text("Item").click(button="right")

# Shift + click
page.get_by_text("Item").click(modifiers=["Shift"])

# Hover over element
page.get_by_text("Item").hover()

# Click the top left corner
page.get_by_text("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:

Sync Async

```
page.get_by_role("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.dispatch_event():

Sync Async

```
page.get_by_role("button").dispatch_event('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.press_sequentially().

Sync Async

```
# Press keys one by one
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

Sync Async

```
# Hit Enter
page.get_by_text("Submit").press("Enter")

# Dispatch Control+Right
page.get_by_role("textbox").press("Control+ArrowRight")

# Press $ sign on keyboard
page.get_by_role("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.

Sync Async

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

```
# <input id=name>
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.set_input_files() 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.

Sync **Async**

```
# Select one file
page.get_by_label("Upload file").set_input_files('myfile.pdf')

# Select multiple files
page.get_by_label("Upload files").set_input_files(['file1.txt', 'file2.txt'])

# Remove all the selected files
page.get_by_label("Upload file").set_input_files([])

# Upload buffer from memory
page.get_by_label("Upload file").set_input_files(
    files=[
        {"name": "test.txt", "mimeType": "text/plain", "buffer": b"this is a
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:

Sync Async

```
with page.expect_file_chooser() as fc_info:
    page.get_by_label("Upload file").click()
file_chooser = fc_info.value
file_chooser.set_files("myfile.pdf")
```

Focus element

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

Sync Async

```
page.get_by_label('password').focus()
```

Drag and Drop

You can perform drag&drop operation with locator.drag_to(). 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.

Sync Async

```
page.locator("#item-to-be-dragged").drag_to(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().

Sync Async

```
page.locator("#item-to-be-dragged").hover()
page.mouse.down()
page.locator("#item-to-drop-at").hover()
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.