async and await in Playwright (JavaScript/TypeScript)

In JavaScript/TypeScript, the **async** and **await** keywords are used to write asynchronous code in a more readable and manageable way. In Playwright, which heavily relies on interacting with web pages asynchronously (e.g., loading pages, clicking buttons, fetching data), these keywords are crucial for handling asynchronous operations within tests.

**1. What Are async and await?**

* **async**: Declares a function as asynchronous, allowing it to use the **await** keyword within its body. An async function automatically returns a **Promise**.
* **await**: Pauses the execution of the code within an async function until the Promise returned by an asynchronous operation is either resolved or rejected.

In the context of Playwright, most browser interactions such as navigation, clicks, and input handling are asynchronous, requiring the use of async/await to ensure that the test waits for these actions to complete before proceeding.

**2. Why Use async and await in Playwright?**

In web automation, asynchronous operations are very common. Actions like navigating to a page, waiting for elements to appear, or handling network requests all happen asynchronously. Using async/await helps ensure that Playwright waits for these operations to complete before moving on to the next step in your test, preventing race conditions and timing issues.

Here’s an example:

typescript

Copy code

import { test, expect } from '@playwright/test';

test('load homepage and verify title', async ({ page }) => {

await page.goto('https://example.com');

const title = await page.title();

expect(title).toBe('Example Domain');

});

In this example:

* The test function contains an **asynchronous anonymous function** (declared with async).
* The **await** keyword is used before page.goto() and page.title() to ensure that Playwright waits for the page to fully load and the title to be retrieved before proceeding.

**3. Key Scenarios for async and await in Playwright**

The combination of async and await is widely used in Playwright for handling various types of asynchronous actions such as:

**a) Navigating to Web Pages**

Navigating to a new URL is an asynchronous operation in Playwright, requiring the use of await to ensure that the page is fully loaded before proceeding.

typescript

Copy code

await page.goto('https://example.com');

Without await, the test might continue executing before the page is completely loaded, leading to potential failures.

**b) Waiting for Elements to Appear**

Many web pages have dynamic content that loads asynchronously (e.g., after a button click or an AJAX call). Playwright provides built-in methods like page.waitForSelector() or page.locator().waitFor() to wait for elements to appear, which need to be awaited:

typescript

Copy code

await page.waitForSelector('#submit-button');

This ensures that the test doesn’t proceed before the element is visible on the page.

**c) Handling Asynchronous Actions like Clicks and Inputs**

Clicking buttons, filling forms, and handling other user interactions are asynchronous in Playwright. Here’s an example:

typescript

Copy code

await page.click('text="Submit"');

await page.fill('#username', 'my-username');

The await keyword ensures that Playwright waits for the button click or input operation to finish before executing the next line of code.

**d) Performing Assertions After Async Operations**

Assertions like checking text, visibility, or the state of elements often need to happen **after** certain asynchronous operations (like navigation or clicking) are completed. Using await ensures that assertions are only made after the necessary operations are finished.

typescript

Copy code

await page.click('text="More Info"');

const infoText = await page.textContent('#info');

expect(infoText).toBe('Here is more information.');

In this example, the await before page.textContent() ensures that Playwright fetches the text after the page responds to the click event.

**4. Playwright-Specific Asynchronous Features**

**a) Auto-Waiting in Playwright**

One of Playwright’s key features is **auto-waiting**. This means that Playwright automatically waits for the elements to be ready before interacting with them, so you don’t always need to use explicit waits like waitForSelector. However, you still need to use async/await to ensure that the interaction itself is properly awaited:

typescript

Copy code

await page.click('#submit');

Even though Playwright auto-waits for the button to be clickable, the click() method itself is asynchronous and requires await to ensure that the test waits for the click action to finish before proceeding.

**b) Parallel Execution and Asynchronous Isolation**

Playwright allows you to run tests in parallel to optimize test suite execution time. While the tests run in parallel, **each test** is still an isolated async function. Playwright ensures that each test runs independently without interfering with other tests.

typescript

Copy code

test('first test', async ({ page }) => {

await page.goto('https://example.com/first');

// First test logic

});

test('second test', async ({ page }) => {

await page.goto('https://example.com/second');

// Second test logic

});

In this scenario, both tests can run simultaneously, but async/await ensures that each test handles its own asynchronous operations cleanly.

**5. Handling Promises with await**

In JavaScript, asynchronous operations typically return **Promises**. The await keyword is essentially a way of handling Promises in a synchronous-like fashion, making it easier to manage asynchronous flow without deeply nested callbacks.

Here’s a simple example of how await handles a Promise:

typescript

Copy code

const data = await fetchDataFromAPI();

This code pauses until the Promise returned by fetchDataFromAPI() resolves, allowing the test to proceed only once the data is available.

In Playwright, this pattern is extremely useful when dealing with async operations like API calls or asynchronous page interactions:

typescript

Copy code

const response = await page.waitForResponse('https://api.example.com/data');

expect(response.status()).toBe(200);

Here, the test will wait for the network request to complete and will proceed only once the response is available and its status can be checked.

**6. async/await vs. .then() in Playwright**

Before async/await was introduced, the traditional way to handle asynchronous code in JavaScript was by using the .then() method on Promises. async/await offers a more concise and readable syntax compared to .then(), which is especially helpful in Playwright test code:

**Using .then():**

typescript

Copy code

page.goto('https://example.com').then(() => {

return page.title();

}).then((title) => {

expect(title).toBe('Example Domain');

});

**Using async/await:**

typescript

Copy code

await page.goto('https://example.com');

const title = await page.title();

expect(title).toBe('Example Domain');

The async/await syntax makes the code more readable, easier to maintain, and avoids deeply nested callbacks (often referred to as "callback hell").

**7. Error Handling with async/await**

When using async/await, error handling can be managed using **try/catch** blocks, allowing you to catch and handle errors that occur during asynchronous operations. This is extremely useful in tests where you need to assert specific outcomes even when an error occurs.

typescript

Copy code

try {

await page.goto('https://example.com');

const title = await page.title();

expect(title).toBe('Example Domain');

} catch (error) {

console.error('Test failed:', error);

}

This pattern ensures that any errors during asynchronous operations are caught and handled gracefully, preventing unhandled Promise rejections.

**Conclusion**

The **async** and **await** keywords are fundamental in Playwright for JavaScript/TypeScript to handle asynchronous operations in a clean and readable way. By allowing tests to wait for asynchronous actions to complete before moving on, async/await ensures that tests are reliable, sequential, and free from race conditions. Whether you're navigating pages, handling user interactions, or making network requests, async/await offers a powerful mechanism to manage asynchronous flow within your tests. Combined with Playwright’s auto-waiting and parallel execution capabilities, async/await makes it easy to write robust, maintainable, and scalable test suites.