How to Create/Write Playwright Tests

Ways to Write Playwright Test Cases

In Playwright, we can create/write test cases in two different ways:

1. **Using Code Gen Plugin** – Playwright provides a built-in test generator that automatically records all the steps.
2. **Writing Custom Scripts** – Manually creating test scripts without using the code generator.

All test cases must be created inside the `tests` folder because the Playwright configuration file (`playwright.config.js`) specifies `./tests` as the test directory.

Creating a Test File

Create a new file under the `tests` folder. The file name should be related to the test case and must have a `.spec.js` extension. Example: `loginpage.spec.js`

Sample Test Cases

1. **Launching the Application**
2. **Verifying the Page Title**
3. **Verifying the Page URL**

Importing Required Node Modules

Before writing test cases, we need to import the necessary modules.

- Playwright automatically creates a `node_modules` folder when installed.
- Inside `node_modules`, there is an `@playwright` folder containing various modules.
- We need to import `@playwright/test` to access Playwright's built-in functions and features.

Importing Required Modules

Navigate to `loginpage.spec.js` and use the `require()` function to import Playwright modules.

```
const { test, expect } = require('@playwright/test');
```

Module Explanation:

- **`test`**: Required for writing test cases.
- **`expect`**: Required for adding assertions in test cases.

Writing a Test Case

To write a test case, use the `test` block with an anonymous function.

```
test('Login page', () => {
});
```

What is an Anonymous Function?

An anonymous function is a function without a name, commonly used as an argument inside other functions.

Using Fixtures in Playwright

In Playwright, pages are represented with a fixture called `page`. This fixture allows interaction with the application.

```
test('Login page', ({ page }) => {
});
```

What is a Fixture?

A fixture provides built-in functions for:

- Interacting with elements
- Clicking elements
- Retrieving text from the page
- Checking element presence
- Launching a web page

Using Async and Await

Since JavaScript is an asynchronous language, we need `async` and `await` to ensure execution happens in order.

```
const { test, expect } = require('@playwright/test');
test('Login page', async ({ page }) => {
  await page.goto('https://example.com');
});
```

Why Use `async` and `await`?

JavaScript executes steps independently, but automation requires sequential execution. Using `async` and `await` ensures each step completes before moving to the next.

JavaScript Promise and Execution Order

- `async` makes a function return a promise.
- `await` waits for a promise to resolve before executing the next command.

```
test('Login page', async ({ page }) => {
  await page.goto('https://example.com'); // Waits for the page to load
  await page.close(); // Executes only after the previous step
});
```

Additional Test Cases

1. **Verify the Title of the Web Page**
2. **Verify the URL of the Web Page**

Verify the Title of the Web Page

To capture the title of a webpage, we use the `page.title()` method. This method retrieves the title of the currently loaded webpage.

Steps to Verify the Title of the Web Page:

1. Store the page title in a variable.
2. Use `expect` from Playwright to compare the actual title with the expected title.

Example Code:

```
const { test, expect } = require('@playwright/test');

test('Verify Web Page Title', async ({ page }) => {
    await page.goto('https://example.com'); // Replace with actual URL

    // Capture the title of the webpage
    const pageTitle = await page.title();

    // Assertion to verify the title
    expect(pageTitle).toBe('Expected Title'); // Replace with actual
expected title
});
```

Key Points:

- `await page.title();` fetches the page title.
- `expect(pageTitle).toBe('Expected Title');` ensures the title matches the expected value.

Verify the URL of the Web Page

To verify the URL of the currently loaded webpage, we use the `page.url()` method. This method returns the URL that the browser is currently on.

Steps to Verify the URL of the Web Page:

1. Store the current URL in a variable.
2. Use `expect` to compare the actual URL with the expected URL.

Example Code:

```
const { test, expect } = require('@playwright/test');

test('Verify Web Page URL', async ({ page }) => {
    await page.goto('https://example.com'); // Replace with actual URL

    // Capture the current URL
    const currentUrl = await page.url();

    // Assertion to verify the URL
    expect(currentUrl).toBe('https://example.com'); // Replace with actual
expected URL
});
```

Key Points:

- `await page.url();` fetches the current URL.
- `expect(currentUrl).toBe('https://example.com');` ensures the URL matches the expected value.

Additional Points for Playwright Testing:

Before writing Playwright test scripts, knowing how `var`, `let`, and `const` work helps in handling test data and selectors efficiently. Here's how:

Variable Declaration in JavaScript

JavaScript provides three ways to declare variables:

1. **var** – Can be used anywhere in a function, can be changed and redeclared.
2. **let** – Block-scoped, can be changed but not redeclared.
3. **const** – Block-scoped, cannot be changed or redeclared.

Example:

```
var x = 10;
console.log(x); // 10
var x = 20; // Allowed
console.log(x); // 20

let y = 30;
console.log(y); // 30
y = 40; // Allowed
console.log(y); // 40
// let y = 50; // Error: Cannot redeclare

const z = 50;
console.log(z); // 50
// z = 60; // Error: Cannot change value
// const z = 70; // Error: Cannot redeclare
```

Java vs JavaScript Variable Declaration

Feature	Java	JavaScript
Type Declaration	Required (`int`, `String`)	Not required (`var`, `let`, `const`)
Final/Const	`final` keyword	`const` keyword
Redeclaration	Not allowed	Allowed with `var`