Hooks in Playwright
What are Playwright Hooks?

In Playwright, hooks allow you to run specific code before or after tests. Hooks help you set up preconditions, clean up after tests or repeat common actions across multiple tests without redundancy. They ensure code reusability and make your tests more efficient.

Types of Hooks in Playwright

Playwright provides four different types of hooks:

1. **beforeEach**
    - This hook is executed **before each individual test**.
    - It's useful when you want to perform actions like logging in before every test.
2. **afterEach**
    - This hook is executed **after each individual test**.
    - It's typically used to perform cleanup actions, such as logging out after each test.
3. **beforeAll**
    - This hook is executed **once before any of the tests start running**.
    - It's useful when you need to set up a condition that should be shared across multiple tests.
4. **afterAll**
    - This hook is executed **once after all the tests have run**.
    - It's ideal for final clean-up steps, such as logging out after all tests are completed.

When and Where to Use These Hooks

Consider the following test scenarios for an **eCommerce application**:

- **Test 1:**
    - Login → Homepage test → Logout
- **Test 2:**
    - Login → Add Product to Cart → Logout

In both tests, **login** and **logout** are common actions. However, writing the login and logout steps in each test is **not advisable** because:

- It creates redundancy.
- It decreases the reusability of code.

Using Hooks for Better Reusability:

1. **Identify the test scenario:**
   - The tests we are looking at are "Homepage" and "Add to Cart".
   - Login and Logout are common for both tests.
2. **Use `beforeEach` and `afterEach` hooks:**
   - **beforeEach:** This hook will execute the login step **before every test**.
   - **afterEach:** This hook will execute the logout step **after every test**.

   Example:

   ```
   beforeEach(async () => {
     // Code for logging in
   });

   afterEach(async () => {
     // Code for logging out
   });
   ```

   This ensures that login and logout are automatically executed before and after each individual test.

Using beforeAll and afterAll Hooks:

If you want to run a series of tests sequentially (one after the other), such as:

- Login → Homepage Test → Add to Cart Test → Logout

In this case, **login** and **logout** are still common, but you don't want to repeat them for each test. This is where **beforeAll** and **afterAll** hooks come in:

- **beforeAll** will execute the login step **once before any tests**.
- **afterAll** will execute the logout step **once after all tests**.

This approach ensures that login and logout are only done once for the entire suite of tests, reducing redundancy.

Example:

```
beforeAll(async () => {
  // Code for logging in
});

afterAll(async () => {
  // Code for logging out
});
```

Summary of Hooks Usage:

Hook Type	When It Runs	Example Use Case
beforeEach	Before each individual test	Login before every test
afterEach	After each individual test	Logout after every test
beforeAll	Before all tests start	Login before all tests
afterAll	After all tests finish	Logout after all tests

Code Without using Hooks:

Test Cases

TC_001: Home Page Test

- **Test Description**: This test verifies the functionality of logging in, checking the number of products on the homepage, and logging out.
- **Expected Result**: The user should be able to log in successfully, view 8 product links on the homepage and log out successfully.

TC_002: Add to Cart Page Test

- **Test Description**: This test verifies the functionality of logging in, adding a product (Spiderman) to the cart, handling the confirmation dialog and checking for the success message after adding to the cart.
- **Expected Result**: The user should be able to log in, add the Spiderman product to the cart, confirm the dialog and see the success message.

Example DOM:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Example Page</title>
</head>
<body>
  <!-- Login Form -->
  <form id="loginForm">
    <input id="username" type="text" placeholder="Username" />
    <input id="password" type="password" placeholder="Password" />
    <button class="clickButton" type="submit">Login</button>
  </form>

  <!-- Home Page Products -->
  <div id="homePage">
    <a href="/product/spiderman" class="productLink">Spiderman</a>
    <a href="/product/batman" class="productLink">Batman</a>
    <a href="/product/superman" class="productLink">Superman</a>
    <!-- More products here -->
  </div>

  <!-- Add to Cart Page -->
  <div id="addToCartPage">
    <button id="addToCart" type="button">Add to Cart</button>
  </div>

  <!-- Dialog Box -->
  <dialog id="confirmationDialog">
    <p>Are you sure you want to add this item to your cart?</p>
    <button class="confirmButton" type="button">Yes</button>
    <button class="cancelButton" type="button">No</button>
  </dialog>

  <!-- Message after success -->
  <div id="successMessage" style="display:none">
    <p>Successfully Added to Cart</p>
  </div>

  <!-- Logout Button -->
  <button class="logoutButton">Logout</button>
</body>
</html>
```

Playwright Test Code:

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

// Test case 1: Home Page Test
test('Home Page Test', async ({ page }) => {
  // Navigate to the home page
  await page.goto('https://example.com');

  // Login
  await page.locator('#username').fill('Yogesh');
  await page.locator('#password').fill('testLife@123');
  await page.locator('.clickButton').click();

  // Verify the number of products on the homepage
  const productLinks = await page.$$('.productLink');
  await expect(productLinks).toHaveLength(8); // Checking if there are 8
product links

  // Logout
  await page.locator('.logoutButton').click();
});

// Test case 2: Add to Cart Page Test
test('Add to Cart Page Test', async ({ page }) => {
  // Navigate to the home page
  await page.goto('https://example.com');

  // Login
  await page.locator('#username').fill('Yogesh');
  await page.locator('#password').fill('testLife@123');
  await page.locator('.clickButton').click();

  // Find the product link for 'Spiderman' and click it
  const productLinks = await page.$$('.productLink');
  for (const productLink of productLinks) {
    if (await productLink.textContent() === 'Spiderman') {
      await productLink.click();
      break;
    }
  }

  await page.locator('#addToCart').click();

  // Handle the confirmation dialog
  await page.on('dialog', async (dialog) => {
    await expect(dialog.message()).toContain('Are you sure'); // Checking
the message in the dialog
    await dialog.accept(); // Accept the dialog (simulating 'Yes')
  });

  // Verify the success message
  await expect(page.locator('#successMessage')).toHaveText('Successfully
Added to Cart');

  // Logout
  await page.locator('.logoutButton').click();
});
```

Problem in the above code:

Reusability is missing because both tests have common login and logout functionality. By using hooks, we can optimize this by adding a precondition to execute before each test and a postcondition to execute after each test.
In the above code, the login function executes before each test, and the logout function executes after each test.
If you have 10 tests, the `BeforeEach` block will execute before each of the 10 tests, and the `AfterEach` block will execute after each of the 10 tests.

Code With Using Hooks:

In Playwright, hooks such as `beforeEach`, `afterEach`, `beforeAll`, and `afterAll` are used to perform setup and teardown operations around your tests. By using these hooks, you can avoid repeating code (like login and logout) across multiple test cases.

Handling Page Fixture in Hooks (Playwright - TypeScript)
Problem Statement

Issue with Page Fixture in Hooks

When writing tests in Playwright, we often use the **page fixture** to interact with the browser. However, when using **beforeEach** and **afterEach** hooks, the page fixture is not directly accessible. This creates challenges:

- **Page fixture is available in tests, but not in hooks.**
- **Code duplication** in test cases due to repeated login and logout steps.
- **Harder test maintenance**, as common setup and teardown steps are manually added in each test.

Solutions

There are **two approaches** to resolve this issue:

Solution 1: Using Browser Fixture to Create a Page in Hooks

Instead of using the **page fixture** directly in the hooks, we can use the **browser fixture** in the `beforeEach` hook to create a new page instance.

Solution 2: Using a Separate Page Fixture File

We create a **custom page fixture** file that:

1. **Manages the browser instance** (open and close actions).
2. **Exports the page object** so it can be used in both tests and hooks.
3. **Ensures reusability and clean code** by centralizing login and logout operations.

Implementation

Solution 1: Using Browser Fixture in Hooks

Step 1: Creating the Page Fixture in Hooks

To handle the missing **page fixture** in hooks, we use the **browser fixture** in the `beforeEach` hook and create a new page instance.

Code Example (TypeScript)

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

let page; // Declare a common page variable that will be used in multiple
tests

test.beforeEach(async ({ browser }) => {
  // Create a new page instance using the browser fixture
  page = await browser.newPage();

  // Perform login
  await page.locator('#username').fill('Yogesh');
  await page.locator('#password').fill('testLife@123');
  await page.locator('.clickButton').click();
});

test.afterEach(async () => {
  // Perform logout
  await page.locator('.logoutButton').click();
});
```

Why This Approach?

- **Reusability**: Avoids repeating login/logout logic in each test.
- **Efficiency**: Reduces code duplication.
- **Global Page**: The page variable is shared across tests.

Solution 2: Using a Separate Page Fixture File

Steps to Implement This Solution

Step 1: Create a `pageFixture.ts` file to manage the browser and page instance.
Step 2: Create a `hooks.ts` file to handle `beforeEach` and `afterEach` logic.
Step 3: Write test files (`homePageTest.spec.ts` & `addToCartTest.spec.ts`) that use the shared page instance.

Step 1: Create `pageFixture.ts` (Manages Page Instance)

This file handles browser launch and page creation.

```
import { Browser, Page, chromium } from '@playwright/test';

let browser: Browser;
let page: Page;

// Function to launch browser and create a new page
export const launchBrowser = async (): Promise<Page> => {
    browser = await chromium.launch({ headless: false });
    page = await browser.newPage();
    return page;
};

// Function to close the browser
export const closeBrowser = async (): Promise<void> => {
    if (browser) {
        await browser.close();
    }
};

// Exporting the page object
export { page };
```

Step 2: Create `hooks.ts` (Handles Hooks Setup & Cleanup)

```
import { launchBrowser, closeBrowser } from './pageFixture';

let page: any;

// Runs before each test
beforeEach(async () => {
    page = await launchBrowser();
    await page.goto('https://example.com/login');
});

// Runs after each test
afterEach(async () => {
    await closeBrowser();
});

export { page };
```

Step 3.1: Create `homePageTest.spec.ts` (Home Page Test Case)

```
import { test, expect } from '@playwright/test';
import { page } from './hooks';

test('TC_001: Home Page Test', async () => {
    await page.fill('#username', 'testuser');
    await page.fill('#password', 'password');
    await page.click('#loginButton');

    // Verify 8 product links on homepage
    const products = await page.$$('a.product-link');
    expect(products.length).toBe(8);

    await page.click('#logoutButton');
});
```

Step 3.2: Create `addToCartTest.spec.ts` (Add to Cart Test Case)

```
import { test, expect } from '@playwright/test';
import { page } from './hooks';

test('TC_002: Add to Cart Page Test', async () => {
    await page.fill('#username', 'testuser');
    await page.fill('#password', 'password');
    await page.click('#loginButton');

    // Add Spiderman product to the cart
    await page.click('text=Spiderman');
    await page.click('#addToCartButton');

    // Handle confirmation dialog
    page.on('dialog', async (dialog) => {
        await dialog.accept();
    });

    // Verify success message
    const successMessage = await page.textContent('#successMessage');
    expect(successMessage).toContain('Product added to cart successfully');
});
```

Summary & Benefits

Why Use a Separate Page Fixture & Hooks?

- **Reusability**: The page object is shared across tests, avoiding redundancy.
- **Clean Code**: Common steps (browser launch, navigation, cleanup) are centralized.
- **Scalability**: Easy to add more test cases without modifying setup logic.
- **Maintainability**: If changes are needed, they are made in one place (`pageFixture.ts`).
