**PlayWright (Day 3)**

**1. Fixtures in Playwright and TypeScript**

Fixtures in Playwright allow you to set up **reusable test environments or components**. By defining fixtures, you can share setup code (like logging into a site or filling out a form) across multiple tests, improving code reusability and reducing duplication.

**Example:**

import { TodoPage } from "./todoPage";

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

type myFixtures = { todoPage: TodoPage }

export const test = base.extend<myFixtures>({

todoPage: async ({ page }, use) => {

const todoPage = new TodoPage(page);

await todoPage.goto();

await todoPage.addTodo('item 1');

await use(todoPage);

}

});

export { expect } from '@playwright/test';

* **Explanation**: The todoPage fixture initializes the TodoPage class and sets up initial tasks (item 1, item 2) for each test. This fixture is shared across different tests.

**2. Parallel Execution of Tests**

Playwright allows you to run tests in parallel, which significantly reduces test execution time. By configuring the test.describe.configure({ mode: 'parallel' }) setting, you can execute tests simultaneously across multiple browsers.

**Example:**

test.describe.configure({ mode: 'parallel' });

test('has title', async ({ page }) => {

await page.goto('https://playwright.dev/');

await expect(page).toHaveTitle(/Playwright/);

});

import { performLogin} from './sauceDemoMyFixture';

test('Login Test', async ({ page }) => {

  page.goto('https://www.saucedemo.com/');

  await performLogin(page, 'standard\_user', 'secret\_sauce');

});

* **Explanation**: The test.describe.configure({ mode: 'parallel' }) command tells Playwright to run the tests in parallel, improving the speed of test execution.

**3. Parameterized Test Cases**

Parameterized tests allow you to run the same test logic with different input data. This is useful for testing multiple scenarios with minimal code repetition.

**Example:**

import test from "@playwright/test";

test.describe.configure({ mode: 'parallel' });

[ { name: 'Naveen', msg: 'Hello' },

{ name: 'Deepak', msg: 'Hey' },

{ name: 'Vishal', msg: 'Hola' }

].forEach(({ name, msg }) => {

test(`testing with ${name}`, async ({ page }) => {

console.log(`Message for ${name}: ${msg}`);

// Your test logic here, for example: await page.goto(`http://example.com/${name}`);

});

});

* **Explanation**: This example runs the same test with different names and messages, which can be useful for testing multiple user interactions or different content on a page.

**4. npx Command for Parallel Execution**

To run tests in parallel using Playwright, you can use the npx playwright test command with configuration options to specify the number of workers and parallel execution.

**Example Command:**

npx playwright test --workers=4

* **Explanation**: This command runs the tests in parallel across 4 workers, distributing the test load to speed up the testing process.

**5. Reusable Login Function with Fixtures**

You can also create reusable login functions to simplify login operations across multiple tests. This approach is useful for tests requiring user authentication.

**Example:**

import { LoginPage } from "./sauceDemoPage";

export async function performLogin(page: any, username: string, password: string) {

const loginPage = new LoginPage();

await loginPage.login(page, username, password);

}

* **Explanation**: The performLogin function takes the page, username, and password as inputs and performs the login action. This is reusable across tests, saving time and reducing code duplication.