**PlayWright (Dialogs, Clock, Sharding)**

**Alert Dialogs**

**Description:**  
Alert dialogs display a message to the user with an OK button. In Playwright, these dialogs are auto-dismissed unless explicitly handled.

**Code Example:**

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

test('alert dialog handling', async ({ page }: { page: Page }) => {

page.on('dialog', async dialog => {

console.log(dialog.message()); // Logs the alert message

await dialog.accept(); // Accepts the alert (dismisses it)

});

await page.evaluate(() => {

alert('This is an alert!');

});

});

**Confirm Dialogs**

**Description:**  
Confirm dialogs ask the user to confirm or cancel an action. Playwright allows handling the confirmation with .accept() or .dismiss().

**Code Example:**

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

test('confirm dialog handling', async ({ page }: { page: Page }) => {

page.on('dialog', async dialog => {

console.log(dialog.message()); // Logs the confirmation message

await dialog.accept(); // Clicks "OK"

});

await page.evaluate(() => {

if (confirm('Do you want to proceed?')) {

console.log('User clicked OK');

} else {

console.log('User clicked Cancel');

}

});

});

**Prompt Dialogs**

**Description:**  
Prompt dialogs allow the user to input text. You can handle the input by either accepting with .accept() and providing a value or dismissing with .dismiss().

**Code Example:**

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

test('prompt dialog handling', async ({ page }: { page: Page }) => {

page.on('dialog', async dialog => {

console.log(dialog.message()); // Logs the prompt message

await dialog.accept('Sample Input'); // Accepts with input value

});

await page.evaluate(() => {

const userInput = prompt('Please enter something:');

console.log(userInput);

});

});

**Default Behaviour of Dialogs**

**Description:**  
By default, Playwright auto-dismisses alert, confirm, and prompt dialogs. You don't need to handle them unless you want to perform specific actions like logging or passing values.

**Key Point:**  
To prevent automatic dismissal, you must register a dialog handler before triggering any dialog. Use dialog.accept() or dialog.dismiss() as needed.

**setFixedTime()**

**Description:**  
setFixedTime() allows you to set a specific fixed time for the page’s JavaScript execution, making it useful for testing date/time-dependent functionality without relying on the real current time.

**Code Example**

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

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

await page.context().setFixedTime(new Date('2024-02-02T10:00:00'));

await page.goto('http://127.0.0.1:5500/tests/index.html');

await expect(page.locator('#date')).toHaveText('2/2/2024');

await page.waitForTimeout(5000);

});

**Sharding**

**What is Sharding?**  
Sharding in Playwright (and generally in testing) is a technique used to split tests into smaller parts, called "shards", and run them in parallel across multiple workers (or processes). This allows faster execution of tests by distributing the workload across multiple resources, rather than running all tests sequentially on a single machine.

**Advantages of Shards**

* **Faster Test Execution:** Running tests in parallel (across different shards) can drastically reduce the total time to run the tests.
* **Better Resource Utilization:** You can use multiple machines or processes to distribute the test execution load.
* **Scalable:** As your test suite grows, you can add more shards to scale the parallelism.

**Disadvantages of Shards**

* **Complex Setup:** Setting up parallel execution across multiple machines or processes can be complex and require extra resources (e.g., machines or virtual environments).
* **Flaky Tests:** Some tests might fail intermittently when run in parallel, which could be caused by shared state or other concurrency issues.
* **Overhead:** There is overhead in managing and coordinating the multiple parallel executions.

**Shards Concept - Simple Diagram**

Here’s a simple arrow diagram to represent how sharding works:

[Sharded Tests]

/ | \

[Shard 1] [Shard 2] [Shard 3] -> Parallel Execution

| | |

[Test 1] [Test 2] [Test 3]

**GitHub Actions for Shards**

GitHub Actions can be used to automate the execution of tests in parallel by defining multiple jobs. Here’s an example of how you might set up sharding in a GitHub Actions workflow for Playwright tests:

**Example:**

name: Playwright Sharding Tests

on: [push, pull\_request]

jobs:

shard1:

runs-on: ubuntu-latest

strategy:

matrix:

testShard: [1]

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Install dependencies

run: npm install

- name: Run tests (Shard 1)

run: npx playwright test --shard=1

shard2:

runs-on: ubuntu-latest

strategy:

matrix:

testShard: [2]

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Install dependencies

run: npm install

- name: Run tests (Shard 2)

run: npx playwright test --shard=2

In this example, two shards are defined (shard1 and shard2), each running a subset of tests in parallel. You can adjust the number of shards by adding more jobs or modifying the strategy matrix.