Parallelism

Playwright Test supports both serial and parallel test execution, allowing you to run your tests efficiently based on your project’s needs. By default, Playwright runs test files in parallel using multiple worker processes, each running independently. This parallel execution speeds up the test runs, especially when there are many independent test files.

However, you can control how tests are executed by configuring serial or parallel modes depending on the requirements of each test file or the project as a whole.

**Parallel Execution**

Playwright Test achieves parallel execution by running multiple worker processes simultaneously. Each worker executes a single test file at a time, allowing multiple test files to run concurrently. However, tests within a single file are run sequentially by default.

To enable parallel execution for tests in a single file, you can configure the test using test.describe.configure({ mode: 'parallel' }). This helps to execute independent tests within a file in parallel, ensuring efficient resource usage.

**Configuration for Parallel Execution**

1. **File-Level Parallelization**  
   You can configure individual test files to run in parallel with the following:

typescript

Copy code

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

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

test('parallel test 1', async ({ page }) => { /\* Test code \*/ });

test('parallel test 2', async ({ page }) => { /\* Test code \*/ });

1. **Project-Level Parallelization**  
   You can enable fully parallel execution for all test files within the project. In your Playwright configuration file:

typescript

Copy code

import { defineConfig } from '@playwright/test';

export default defineConfig({

fullyParallel: true,

});

1. **Limiting Workers**  
   You can control the number of parallel workers either via the command line or configuration file. Limiting workers helps in balancing the execution based on the machine’s capabilities.

**Command Line**:

bash

Copy code

npx playwright test --workers=4

**Configuration File**:

typescript

Copy code

export default defineConfig({

workers: process.env.CI ? 2 : undefined,

});

**Serial Execution**

In cases where tests are interdependent or need to run in a specific order, Playwright provides a **serial execution mode**. Serial execution ensures that tests in a single file are run one after another, without parallelization.

You can enable serial execution by annotating a test or a describe block with mode: 'serial'. This is especially useful for tests that rely on shared state or order-sensitive actions.

**Configuration for Serial Execution**

1. **File-Level Serialization**  
   All tests in a file can be annotated to run serially:

typescript

Copy code

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

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

test('first test', async ({ page }) => { /\* Test code \*/ });

test('second test', async ({ page }) => { /\* Test code \*/ });

1. **Group-Level Serialization**  
   Specific test groups within a file can be serialized while allowing others to run in parallel.

typescript

Copy code

test.describe('My Serial Tests', () => {

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

test('runs first', async ({ page }) => { /\* Test code \*/ });

test('runs second', async ({ page }) => { /\* Test code \*/ });

});

**Disabling Parallelism**

In some scenarios, you may need to disable parallelism entirely and force tests to run in a strictly serial order across files. You can set the number of workers to 1 to achieve this.

**Disable Parallelism**

You can disable parallel execution either via the command line or the configuration file.

**Command Line**:

bash

Copy code

npx playwright test --workers=1

**Configuration File**:

typescript

Copy code

export default defineConfig({

workers: 1,

});

**Limit Failures and Fail Fast**

Playwright provides options to limit the number of failures and stop the test run early. This can be configured to prevent wasting time and resources when a large number of tests are expected to fail.

**Command Line**:

bash

Copy code

npx playwright test --max-failures=10

**Configuration File**:

typescript

Copy code

export default defineConfig({

maxFailures: process.env.CI ? 10 : undefined,

});

**Worker Processes and Isolation**

Each test worker process runs independently, ensuring isolated environments for tests. Workers cannot communicate with each other. Playwright ensures clean environments by shutting down workers after failures, and you can isolate test data between parallel workers using unique identifiers like process.env.TEST\_WORKER\_INDEX and testInfo.workerIndex.

**Example for Isolating Data**

typescript

Copy code

import { test as baseTest } from '@playwright/test';

export const test = baseTest.extend<{}, { dbUserName: string }>({

dbUserName: [async ({ }, use) => {

const userName = `user-${test.info().workerIndex}`;

await use(userName);

}, { scope: 'worker' }],

});

**Conclusion**

Playwright’s flexibility in handling both serial and parallel test execution allows for highly efficient and tailored testing strategies. Depending on the nature of the test suite and project requirements, you can optimize performance by parallelizing independent tests or running tests serially when necessary.

This chapter should fit well into your **playwright-learning** repository. Let me know if you'd like to add or modify anything!

4o

**You said:**

I THINK YOU MISSED SOMETHINGS

**ChatGPT said:**

ChatGPT

It seems I missed out on creating the chapter in the same format as the others while covering **Serial and Parallel Execution** in Playwright. I'll ensure to follow the exact structure this time.

**Parallelism and Serial Execution**

**Introduction**

In Playwright, tests are typically run in parallel to optimize performance and reduce execution time. By default, Playwright Test executes multiple test files in parallel using separate worker processes. Each worker runs independently, providing an isolated environment to avoid conflicts between tests. However, Playwright also offers the flexibility to run tests in serial mode when test dependencies exist.

This chapter explores how to configure and manage parallel and serial test execution in Playwright.

**Parallel Execution**

Parallel execution in Playwright allows test files to be run simultaneously in different worker processes. These workers are independent OS processes, and each worker starts its own browser instance. Parallelism is enabled by default, but you can adjust the level of parallelism based on your project needs.

**Configuring Parallel Tests**

1. **Default Behavior**: Test files are run in parallel, while individual tests inside a file run sequentially.
2. **Configuring Tests in a Single File**: You can configure tests inside a single file to run in parallel by using test.describe.configure({ mode: 'parallel' }).

typescript

Copy code

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

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

test('runs in parallel 1', async ({ page }) => {

// Test logic

});

test('runs in parallel 2', async ({ page }) => {

// Test logic

});

1. **Configuring Entire Projects**: You can configure an entire project to run all tests in parallel by setting fullyParallel in the Playwright configuration file:

typescript

Copy code

import { defineConfig } from '@playwright/test';

export default defineConfig({

fullyParallel: true, // Enable parallel execution for all tests

});

**Limiting Parallel Workers**

You can control the maximum number of parallel worker processes to run tests more efficiently, especially in CI environments or resource-limited systems. This can be done either via command line or in the configuration file.

* **Command Line**:

bash

Copy code

npx playwright test --workers=4

* **Configuration File**:

typescript

Copy code

import { defineConfig } from '@playwright/test';

export default defineConfig({

workers: process.env.CI ? 2 : undefined, // Limits workers in CI

});

**Disabling Parallelism**

To disable parallel execution, set the number of workers to 1:

* **Command Line**:

bash

Copy code

npx playwright test --workers=1

* **Configuration File**:

typescript

Copy code

import { defineConfig } from '@playwright/test';

export default defineConfig({

workers: 1, // Disable parallel execution

});

**Serial Execution**

Serial execution is useful for interdependent tests that cannot run in parallel. In serial mode, tests are run one after the other, and if any test fails, the subsequent tests are skipped. This ensures that failures in earlier tests don't compromise the execution of related tests.

**Configuring Serial Tests**

To run tests serially, use test.describe.configure({ mode: 'serial' }). This configuration ensures that tests in the describe block are executed one by one.

Example of serial execution:

typescript

Copy code

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

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

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

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

});

test('runs second', async () => {

// Interdependent test logic

});

**Why Avoid Serial Tests?**

Serial tests are generally discouraged because they tend to increase the execution time and make test suites harder to maintain. The preferred approach is to design independent tests that can be executed in parallel. However, serial tests might be necessary for certain workflows where order or state dependencies are crucial.

**Sharding Tests Across Machines**

Parallel execution can be further optimized by **sharding** your test suite, splitting it into multiple chunks and running them on different machines.

* **Command Line Example**:

bash

Copy code

npx playwright test --shard=1/3

This splits the test suite into three parts and runs only the first part on the current machine.

**Summary**

Parallel execution is the default and recommended approach in Playwright as it speeds up testing by utilizing multiple worker processes. Serial execution should be used sparingly, only when tests depend on the outcome or state of previous tests. Proper configuration of workers, test modes, and sharding can significantly improve testing efficiency and reliability in larger test suites.