Parallelism

Introduction

Playwright Test runs tests in parallel. In order to achieve that, it runs several worker processes that run at the same time. By default, **test files** are run in parallel. Tests in a single file are run in order, in the same worker process.

- You can configure tests using `test.describe.configure` to run **tests in a single file** in parallel.
- You can configure **entire project** to have all tests in all files to run in parallel using testProject.fullyParallel or testConfig.fullyParallel.
- To **disable** parallelism limit the number of workers to one.

You can control the number of parallel worker processes and limit the number of failures in the whole test suite for efficiency.

Worker processes

All tests run in worker processes. These processes are OS processes, running independently, orchestrated by the test runner. All workers have identical environments and each starts its own browser.

You can't communicate between the workers. Playwright Test reuses a single worker as much as it can to make testing faster, so multiple test files are usually run in a single worker one after another.

Workers are always shutdown after a test failure to guarantee pristine environment for following tests.

Limit workers

You can control the maximum number of parallel worker processes via command line or in the configuration file.

From the command line:

```
npx playwright test --workers 4
```

In the configuration file:

playwright.config.ts
```
import { defineConfig } from '@playwright/test';

export default defineConfig({
  // Limit the number of workers on CI, use default locally
  workers: process.env.CI ? 2 : undefined,
});
```

Disable parallelism

You can disable any parallelism by allowing just a single worker at any time. Either set `workers: 1` option in the configuration file or pass `--workers=1` to the command line.

```
npx playwright test --workers=1
```

Parallelize tests in a single file

By default, tests in a single file are run in order. If you have many independent tests in a single file, you might want to run them in parallel with test.describe.configure().

Note that parallel tests are executed in separate worker processes and cannot share any state or global variables. Each test executes all relevant hooks just for itself, including `beforeAll` and `afterAll`.

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

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

test('runs in parallel 1', async ({ page }) => { /* ... */ });
test('runs in parallel 2', async ({ page }) => { /* ... */ });
```

Alternatively, you can opt-in all tests into this fully-parallel mode in the configuration file:

playwright.config.ts
```
import { defineConfig } from '@playwright/test';

export default defineConfig({
  fullyParallel: true,
});
```

You can also opt in for fully-parallel mode for just a few projects:

playwright.config.ts
```
import { defineConfig } from '@playwright/test';

export default defineConfig({
  // runs all tests in all files of a specific project in parallel
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
      fullyParallel: true,
```

```
    },
  ]
});
```

Serial mode

You can annotate inter-dependent tests as serial. If one of the serial tests fails, all subsequent tests are skipped. All tests in a group are retried together.

NOTE

Using serial is not recommended. It is usually better to make your tests isolated, so they can be run independently.

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

// Annotate entire file as serial.
test.describe.configure({ mode: 'serial' });

let page: Page;

test.beforeAll(async ({ browser }) => {
  page = await browser.newPage();
});

test.afterAll(async () => {
  await page.close();
});

test('runs first', async () => {
  await page.goto('https://playwright.dev/');
});

test('runs second', async () => {
  await page.getByText('Get Started').click();
});
```

Shard tests between multiple machines

Playwright Test can shard a test suite, so that it can be executed on multiple machines. See sharding guide for more details.

```
npx playwright test --shard=2/3
```

Limit failures and fail fast

You can limit the number of failed tests in the whole test suite by setting `maxFailures` config option or passing `--max-failures` command line flag.

When running with "max failures" set, Playwright Test will stop after reaching this number of failed tests and skip any tests that were not executed yet. This is useful to avoid wasting resources on broken test suites.

Passing command line option:

```
npx playwright test --max-failures=10
```

Setting in the configuration file:

playwright.config.ts

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

export default defineConfig({
  // Limit the number of failures on CI to save resources
  maxFailures: process.env.CI ? 10 : undefined,
});
```

Worker index and parallel index

Each worker process is assigned two ids: a unique worker index that starts with 1, and a parallel index that is between `0` and `workers - 1`. When a worker is restarted, for example after a failure, the new worker process has the same `parallelIndex` and a new `workerIndex`.

You can read an index from environment variables `process.env.TEST_WORKER_INDEX` and `process.env.TEST_PARALLEL_INDEX`, or access them through testInfo.workerIndex and testInfo.parallelIndex.

Isolate test data between parallel workers

You can leverage `process.env.TEST_WORKER_INDEX` or testInfo.workerIndex mentioned above to isolate user data in the database between tests running on different workers. All tests run by the worker reuse the same user.

Create `playwright/fixtures.ts` file that will create `dbUserName` fixture and initialize a new user in the test database. Use testInfo.workerIndex to differentiate between workers.

playwright/fixtures.ts

```
import { test as baseTest, expect } from '@playwright/test';
// Import project utils for managing users in the test database.
import { createUserInTestDatabase, deleteUserFromTestDatabase } from './my-db-utils';

export * from '@playwright/test';
export const test = baseTest.extend<{}, { dbUserName: string }>({
  // Returns db user name unique for the worker.
  dbUserName: [async ({ }, use) => {
    // Use workerIndex as a unique identifier for each worker.
    const userName = `user-${test.info().workerIndex}`;
    // Initialize user in the database.
    await createUserInTestDatabase(userName);
    await use(userName);
    // Clean up after the tests are done.
    await deleteUserFromTestDatabase(userName);
  }, { scope: 'worker' }],
});
```

Now, each test file should import `test` from our fixtures file instead of `@playwright/test`.

tests/example.spec.ts

```
// Important: import our fixtures.
import { test, expect } from '../playwright/fixtures';

test('test', async ({ dbUserName }) => {
  // Use the user name in the test.
});
```

Control test order

Playwright Test runs tests from a single file in the order of declaration, unless you parallelize tests in a single file.

There is no guarantee about the order of test execution across the files, because Playwright Test runs test files in parallel by default. However, if you disable parallelism, you can control test order by either naming your files in alphabetical order or using a "test list" file.

Sort test files alphabetically

When you **disable parallel test execution**, Playwright Test runs test files in alphabetical order. You can use some naming convention to control the test order, for example `001-user-signin-flow.spec.ts`, `002-create-new-document.spec.ts` and so on.

Use a "test list" file

WARNING

Tests lists are discouraged and supported as a best-effort only. Some features such as VS Code Extension and tracing may not work properly with test lists.

You can put your tests in helper functions in multiple files. Consider the following example where tests are not defined directly in the file, but rather in a wrapper function.

feature-a.spec.ts

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

export default function createTests() {
  test('feature-a example test', async ({ page }) => {
    // ... test goes here
  });
}
```

feature-b.spec.ts

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

export default function createTests() {
  test.use({ viewport: { width: 500, height: 500 } });

  test('feature-b example test', async ({ page }) => {
    // ... test goes here
```

```
  });
}
```

You can create a test list file that will control the order of tests - first run `feature-b` tests, then `feature-a` tests. Note how each test file is wrapped in a `test.describe()` block that calls the function where tests are defined. This way `test.use()` calls only affect tests from a single file.

test.list.ts

```
import { test } from '@playwright/test';
import featureBTests from './feature-b.spec.ts';
import featureATests from './feature-a.spec.ts';

test.describe(featureBTests);
test.describe(featureATests);
```

Now **disable parallel execution** by setting workers to one, and specify your test list file.

playwright.config.ts

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

export default defineConfig({
  workers: 1,
  testMatch: 'test.list.ts',
});
```

NOTE

Do not define your tests directly in a helper file. This could lead to unexpected results because your tests are now dependent on the order of `import`/`require` statements. Instead, wrap tests in a function that will be explicitly called by a test list file, as in the example above.

Previous
Global setup and teardown

Next
Parameterize tests

    - Use a "test list" file

Learn

- Getting started
- Playwright Training
- Learn Videos
- Feature Videos

Community

- Stack Overflow
- Discord
- Twitter
- LinkedIn

More

- GitHub
- YouTube
- Blog
- Ambassadors