Parameterize tests

Introduction

You can either parameterize tests on a test level or on a project level.

Parameterized Tests

example.spec.ts

```
[
  { name: 'Alice', expected: 'Hello, Alice!' },
  { name: 'Bob', expected: 'Hello, Bob!' },
  { name: 'Charlie', expected: 'Hello, Charlie!' },
].forEach(({ name, expected }) => {
  // You can also do it with test.describe() or with multiple tests as long the test name is unique.
  test(`testing with ${name}`, async ({ page }) => {
    await page.goto(`https://example.com/greet?name=${name}`);
    await expect(page.getByRole('heading')).toHaveText(expected);
  });
});
```

Before and after hooks

Most of the time you should put `beforeEach`, `beforeAll`, `afterEach` and `afterAll` hooks outside of `forEach`, so that hooks are executed just once:

example.spec.ts

```
test.beforeEach(async ({ page }) => {
  // ...
});

test.afterEach(async ({ page }) => {
  // ...
});

[
  { name: 'Alice', expected: 'Hello, Alice!' },
  { name: 'Bob', expected: 'Hello, Bob!' },
  { name: 'Charlie', expected: 'Hello, Charlie!' },
].forEach(({ name, expected }) => {
  test(`testing with ${name}`, async ({ page }) => {
    await page.goto(`https://example.com/greet?name=${name}`);
    await expect(page.getByRole('heading')).toHaveText(expected);
```

```
  });
});
```

If you want to have hooks for each test, you can put them inside a `describe()` - so they are executed for each iteration / each individual test:

example.spec.ts

```
[
  { name: 'Alice', expected: 'Hello, Alice!' },
  { name: 'Bob', expected: 'Hello, Bob!' },
  { name: 'Charlie', expected: 'Hello, Charlie!' },
].forEach(({ name, expected }) => {
  test.describe(() => {
    test.beforeEach(async ({ page }) => {
      await page.goto(`https://example.com/greet?name=${name}`);
    });
    test(`testing with ${expected}`, async ({ page }) => {
      await expect(page.getByRole('heading')).toHaveText(expected);
    });
  });
});
```

Parameterized Projects

Playwright Test supports running multiple test projects at the same time. In the following example, we'll run two projects with different options.

We declare the option `person` and set the value in the config. The first project runs with the value `Alice` and the second with the value `Bob`.

- TypeScript
- JavaScript

my-test.ts

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

export type TestOptions = {
  person: string;
};

export const test = base.extend<TestOptions>({
  // Define an option and provide a default value.
  // We can later override it in the config.
  person: ['John', { option: true }],
});
```

my-test.js

```
const base = require('@playwright/test');

exports.test = base.test.extend({
  // Define an option and provide a default value.
  // We can later override it in the config.
  person: ['John', { option: true }],
});
```

We can use this option in the test, similarly to fixtures.

example.spec.ts

```
import { test } from './my-test';

test('test 1', async ({ page, person }) => {
  await page.goto(`/index.html`);
  await expect(page.locator('#node')).toContainText(person);
  // ...
});
```

Now, we can run tests in multiple configurations by using projects.

- TypeScript
- JavaScript

playwright.config.ts

```
import { defineConfig } from '@playwright/test';
import type { TestOptions } from './my-test';

export default defineConfig<TestOptions>({
  projects: [
    {
      name: 'alice',
      use: { person: 'Alice' },
    },
    {
      name: 'bob',
      use: { person: 'Bob' },
    },
  ]
});
```

playwright.config.ts

```
// @ts-check

module.exports = defineConfig({
  projects: [
    {
      name: 'alice',
      use: { person: 'Alice' },
    },
    {
      name: 'bob',
      use: { person: 'Bob' },
    },
  ]
});
```

We can also use the option in a fixture. Learn more about fixtures.

- TypeScript
- JavaScript

my-test.ts

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

export type TestOptions = {
  person: string;
};
```

```
export const test = base.extend<TestOptions>({
  // Define an option and provide a default value.
  // We can later override it in the config.
  person: ['John', { option: true }],

  // Override default "page" fixture.
  page: async ({ page, person }, use) => {
    await page.goto('/chat');
    // We use "person" parameter as a "name" for the chat room.
    await page.getByLabel('User Name').fill(person);
    await page.getByText('Enter chat room').click();
    // Each test will get a "page" that already has the person name.
    await use(page);
  },
});
```

my-test.js

```
const base = require('@playwright/test');

exports.test = base.test.extend({
  // Define an option and provide a default value.
  // We can later override it in the config.
  person: ['John', { option: true }],

  // Override default "page" fixture.
  page: async ({ page, person }, use) => {
    await page.goto('/chat');
    // We use "person" parameter as a "name" for the chat room.
    await page.getByLabel('User Name').fill(person);
    await page.getByText('Enter chat room').click();
    // Each test will get a "page" that already has the person name.
    await use(page);
  },
});
```

NOTE

Parameterized projects behavior has changed in version 1.18. Learn more.

Passing Environment Variables

You can use environment variables to configure tests from the command line.

For example, consider the following test file that needs a username and a password. It is usually a good idea not to store your secrets in the source code, so we'll need a way to pass secrets from outside.

example.spec.ts

```
test(`example test`, async ({ page }) => {
  // ...
  await page.getByLabel('User Name').fill(process.env.USER_NAME);
  await page.getByLabel('Password').fill(process.env.PASSWORD);
});
```

You can run this test with your secret username and password set in the command line.

- Bash
- PowerShell

- Batch

```
USER_NAME=me PASSWORD=secret npx playwright test
$env:USER_NAME=me
$env:PASSWORD=secret
npx playwright test
set USER_NAME=me
set PASSWORD=secret
npx playwright test
```

Similarly, configuration file can also read environment variables passed through the command line.

playwright.config.ts

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

export default defineConfig({
  use: {
    baseURL: process.env.STAGING === '1' ? 'http://staging.example.test/' :
'http://example.test/',
  }
});
```

Now, you can run tests against a staging or a production environment:

- Bash
- PowerShell
- Batch

```
STAGING=1 npx playwright test
$env:STAGING=1
npx playwright test
set STAGING=1
npx playwright test
```

.env files

To make environment variables easier to manage, consider something like `.env` files. Here is an example that uses [dotenv] package to read environment variables directly in the configuration file.

playwright.config.ts

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

// Read from ".env" file.
dotenv.config({ path: path.resolve(__dirname, '.env') });

// Alternatively, read from "../my.env" file.
dotenv.config({ path: path.resolve(__dirname, '..', 'my.env') });

export default defineConfig({
  use: {
    baseURL: process.env.STAGING === '1' ? 'http://staging.example.test/' :
'http://example.test/',
```

```
  }
});
```

Now, you can just edit `.env` file to set any variables you'd like.

```
# .env file
STAGING=0
USER_NAME=me
PASSWORD=secret
```

Run tests as usual, your environment variables should be picked up.

```
npx playwright test
```

Create tests via a CSV file

The Playwright test-runner runs in Node.js, this means you can directly read files from the file system and parse them with your preferred CSV library.

See for example this CSV file, in our example `input.csv`:

```
"test_case","some_value","some_other_value"
"value 1","value 11","foobar1"
"value 2","value 22","foobar21"
"value 3","value 33","foobar321"
"value 4","value 44","foobar4321"
```

Based on this we'll generate some tests by using the csv-parse library from NPM:

test.spec.ts

```
import fs from 'fs';
import path from 'path';
import { test } from '@playwright/test';
import { parse } from 'csv-parse/sync';

const records = parse(fs.readFileSync(path.join(__dirname, 'input.csv')), {
  columns: true,
  skip_empty_lines: true
});

for (const record of records) {
  test(`foo: ${record.test_case}`, async ({ page }) => {
    console.log(record.test_case, record.some_value,
record.some_other_value);
  });
}
```

Previous
Parallelism

Next
Projects

Learn

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

Community

- Stack Overflow
- Discord
- Twitter
- LinkedIn

More

- GitHub
- YouTube
- Blog
- Ambassadors