Retries, Sharding, Timeout and Waiting

Retries

Testwiederholungen sind eine Möglichkeit, einen Test automatisch zu wiederholen, wenn er fehlschlägt. Dies ist nützlich, wenn ein Test unzuverlässig ist und gelegentlich fehlschlägt. Testwiederholungen werden in der Konfigurationsdatei konfiguriert.

Retries

Hier ein erstes Beispiel:

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

test.describe('suite', () => {
  test.beforeAll(async () => { /* ... */ });
  test('first good', async ({ page }) => { /* ... */ });
  test('second flaky', async ({ page }) => { /* ... */ });
  test('third good', async ({ page }) => { /* ... */ });
  test.afterAll(async () => { /* ... */ });
});
```

Retries

Wenn alle Tests bestanden sind, werden sie nacheinander in demselben Arbeitsprozess ausgeführt:

- Worker process starts
  - `beforeAll` hook runs
  - `first good` passes
  - `second flaky` passes
  - `third good` passes
  - `afterAll` hook runs

Retries

Sollte ein Test fehlschlagen, verwirft Playwright Test den gesamten Arbeitsprozess und startet einen neuen.

Die Tests werden im neuen Worker-Prozess fortgesetzt und beginnen mit dem nächsten Test.

- Worker process #1 starts
  - `beforeAll` hook runs
  - `first good` passes
  - `second flaky` fails
  - `afterAll` hook runs
- Worker process #2 starts
  - `beforeAll` hook runs again
  - `third good` passes
  - `afterAll` hook runs

Retries

Wenn man Wiederholungen aktiviert, beginnt der zweite Arbeitsprozess mit der Wiederholung des fehlgeschlagenen Tests.

- Worker process #1 starts
  - `beforeAll` hook runs
  - `first good` passes
  - `second flaky` fails
  - `afterAll` hook runs
- Worker process #2 starts
  - `beforeAll` hook runs again
  - `second flaky` is retried and passes
  - `third good` passes
  - `afterAll` hook runs

Retries

Dieses Schema funktioniert perfekt für unabhängige Tests und garantiert, dass sich fehlgeschlagene Tests nicht auf gesunde Tests auswirken können.

Retries

Wir können dies auch bequem über die Konsole konfigurieren:

```
# Give failing tests 3 retry attempts
npx playwright test --retries=3
```

Oder in der Konfiguration:

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

export default defineConfig({
  // Give failing tests 3 retry attempts
  retries: 3,
});
```

Retries

Playwright Test kategorisiert die Tests wie folgt:

- „passed“ - Tests, die beim ersten Durchlauf bestanden haben; (bestanden)
- „flaky“ - Tests, die beim ersten Durchlauf fehlgeschlagen sind, aber bei der Wiederholung bestanden haben; (fehlerhaft)
- „failed“ - Tests, die beim ersten Durchlauf fehlgeschlagen sind und alle Wiederholungsversuche nicht bestanden haben. (fehlgeschlagen)

Serial Mode

Man verwendet **`test.describe.serial()`**, um abhängige Tests zu gruppieren, damit sie immer zusammen und in der richtigen Reihenfolge ausgeführt werden.

Wenn einer der Tests fehlschlägt, werden alle nachfolgenden Tests übersprungen. Alle Tests in der Gruppe werden gemeinsam erneut durchgeführt.

Sharding

Standardmässig führt Playwright Testdateien parallel aus und bemüht sich um eine optimale Auslastung der CPU-Kerne auf dem Rechner.

Um eine noch stärkere Parallelisierung zu erreichen, kann man die Playwright-Testausführung weiter skalieren, indem man Tests auf mehreren Rechnern gleichzeitig ausführt.

Sharding

Man nennt diese Betriebsart „Sharding“. Sharding in Playwright bedeutet, dass man seine Tests in kleinere Teile, sogenannte „Shards“, aufteilt.

Jeder Shard ist wie ein separater Job, der unabhängig ausgeführt werden kann. Ziel ist es Zeit zu sparen.

Sharding

Wenn man die Tests in Shards aufteilt, kann jeder Shard für sich ausgeführt werden, wobei die verfügbaren CPU-Kerne genutzt werden. Dies trägt zur Beschleunigung des Testprozesses bei, da Aufgaben gleichzeitig ausgeführt werden.

In einer CI-Pipeline kann jeder Shard als separater Job ausgeführt werden, wobei die in der CI-Pipeline verfügbaren Hardwareressourcen, wie z. B. CPU-Kerne, genutzt werden, um die Tests schneller auszuführen.

Sharding

Wie sieht das aus?

```
npx playwright test --shard=1/4
npx playwright test --shard=2/4
npx playwright test --shard=3/4
npx playwright test --shard=4/4
```

Sharding

Als Github Action:

```yaml
name: Playwright Tests
on:
  push:
    branches: [ main, master ]
  pull_request:
    branches: [ main, master ]
jobs:
  playwright-tests:
    timeout-minutes: 60
    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        shardIndex: [1, 2, 3, 4]
        shardTotal: [4]
    steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: lts/*
    - name: Install dependencies
      run: npm ci
    - name: Install Playwright browsers
      run: npx playwright install --with-deps

    - name: Run Playwright tests
      run: npx playwright test --shard=${{ matrix.shardIndex }}/${{ matrix.shardTotal }}

    - name: Upload blob report to GitHub Actions Artifacts
      if: ${{ !cancelled() }}
      uses: actions/upload-artifact@v4
      with:
        name: blob-report-${{ matrix.shardIndex }}
        path: blob-report
        retention-days: 1
```

Timeouts

Playwright Test verfügt über mehrere konfigurierbare Zeitüberschreitungen für verschiedene Aufgaben.

Timeout	Default	Description

Test timeout	30_000 ms	Timeout for each test
		SET IN CONFIG
		`{ timeout: 60_000 }`
		OVERRIDE IN TEST
		`test.setTimeout(120_000)`

Expect timeout	5_000 ms	Timeout for each assertion
		SET IN CONFIG
		`{ expect: { timeout: 10_000 } }`
		OVERRIDE IN TEST
		`expect(locator).toBeVisible({ timeout: 10_000 })`

Timeouts

Timeouts können ganz leicht gesetzt werden (Konfig):

```
1 import { defineConfig } from '@playwright/test';
2
3 export default defineConfig({
4   timeout: 120_000,
5 });
```

Low Level Timeouts (Advanced)

Timeout	Default	Description

Action timeout	no timeout	Timeout for each action
		SET IN CONFIG
		`{ use: { actionTimeout: 10_000 } }`
		OVERRIDE IN TEST
		`locator.click({ timeout: 10_000 })`

Navigation timeout	no timeout	Timeout for each navigation action
		SET IN CONFIG
		`{ use: { navigationTimeout: 30_000 } }`
		OVERRIDE IN TEST
		`page.goto('/', { timeout: 30_000 })`

Global timeout	no timeout	Global timeout for the whole test run SET IN CONFIG `{ globalTimeout: 3_600_000 }`
`beforeAll`/`afterAll` timeout	30_000 ms	Timeout for the hook SET IN HOOK `test.setTimeout(60_000)`
Fixture timeout	no timeout	Timeout for an individual fixture SET IN FIXTURE `{ scope: 'test', timeout: 30_000 }`

Timeouts

Timeouts können ganz leicht gesetzt werden (Test):

```
1
2  import { test, expect } from '@playwright/test';
3
4  test('slow test', async ({ page }) => {
5    test.slow(); // Easy way to triple the default timeout
6    // ...
7  });
8
9  test('very slow test', async ({ page }) => {
10   test.setTimeout(120_000);
11   // ...
12 });
```

Waiting

Playwright führt vor der Ausführung von Aktionen eine Reihe von Prüfungen der **`Aktionsfähigkeit`** der Elemente durch, um sicherzustellen, dass sich diese Aktionen wie erwartet verhalten.

Es wartet automatisch, bis alle relevanten Tests bestanden sind, und führt erst dann die angeforderte Aktion aus.

Wenn die erforderlichen Prüfungen nicht innerhalb der angegebenen Zeitspanne erfolgreich sind, schlägt die Aktion mit dem **`TimeoutError`** fehl.

Waiting

For example, for locator.click(), Playwright will ensure that:

- locator resolves to exactly one element
- element is Visible
- element is Stable, as in not animating or completed animation
- element Receives Events, as in not obscured by other elements
- element is Enabled

Waiting

Assertions

Playwright includes auto-retrying assertions that remove flakiness by waiting until the condition is met, similarly to auto-waiting before actions.

Assertion	Description
expect(locator).toBeAttached()	Element is attached
expect(locator).toBeChecked()	Checkbox is checked
expect(locator).toBeDisabled()	Element is disabled
expect(locator).toBeEditable()	Element is editable
expect(locator).toBeEmpty()	Container is empty
expect(locator).toBeEnabled()	Element is enabled
expect(locator).toBeFocused()	Element is focused
expect(locator).toBeHidden()	Element is not visible
expect(locator).toBeInViewport()	Element intersects viewport
expect(locator).toBeVisible()	Element is visible

nur ein Ausschnitt...

Waiting

Visible

Element is considered visible when it has non-empty bounding box and does not have `visibility:hidden` computed style.

Note that according to this definition:

- Elements of zero size **are not** considered visible.
- Elements with `display:none` **are not** considered visible.
- Elements with `opacity:0` **are** considered visible.

Stable

Element is considered stable when it has maintained the same bounding box for at least two consecutive animation frames.

Enabled

Element is considered enabled unless it is a `<button>`, `<select>`, `<input>` or `<textarea>` with a `disabled` property.

Editable

Element is considered editable when it is enabled and does not have `readonly` property set.

Ende

Das war alles für dieses Kapitel