

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:

```
1 import { test } from '@playwright/test';
2
3 test.describe('suite', () => {
4   test.beforeAll(async () => { /* ... */ });
5   test('first good', async ({ page }) => { /* ... */ });
6   test('second flaky', async ({ page }) => { /* ... */ });
7   test('third good', async ({ page }) => { /* ... */ });
8   test.afterAll(async () => { /* ... */ });
9 });
```

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 zusammen mit dem Browser 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 und fährt von da an fort.

- 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:



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

Oder in der Konfiguration:



```
1 import { defineConfig } from '@playwright/test';
2
3 export default defineConfig({
4   // Give failing tests 3 retry attempts
5   retries: 3,
6 });
```


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 Sie Ihre Tests in kleinere Teile, sogenannte „Shards“, aufteilen.

Jeder Shard ist wie ein separater Job, der unabhängig ausgeführt werden kann. Das Ziel ist es, Ihre Tests aufzuteilen, um die Testlaufzeit zu beschleunigen.

Sharding

Wenn Sie Ihre Tests in Shards aufteilen, 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?



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

Sharding

Als Github Action:

```
1 name: Playwright Tests
2 on:
3   push:
4     branches: [ main, master ]
5   pull_request:
6     branches: [ main, master ]
7 jobs:
8   playwright-tests:
9     timeout-minutes: 60
10    runs-on: ubuntu-latest
11    strategy:
12      fail-fast: false
13      matrix:
14        shardIndex: [1, 2, 3, 4]
15        shardTotal: [4]
16    steps:
17      - uses: actions/checkout@v4
18      - uses: actions/setup-node@v4
19        with:
20          node-version: lts/*
21      - name: Install dependencies
22        run: npm ci
23      - name: Install Playwright browsers
24        run: npx playwright install --with-deps
25
26      - name: Run Playwright tests
27        run: npx playwright test --shard=${{ matrix.shardIndex }}/${{ matrix.shardTotal }}
28
29      - name: Upload blob report to GitHub Actions Artifacts
30        if: ${{ !cancelled() }}
31        uses: actions/upload-artifact@v4
32        with:
33          name: blob-report-${{ matrix.shardIndex }}
34          path: blob-report
35          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 <code>{ timeout: 60_000 }</code> OVERRIDE IN TEST <code>test.setTimeout(120_000)</code> |
| Expect timeout | 5_000 ms | Timeout for each assertion SET IN CONFIG <code>{ expect: { timeout: 10_000 } }</code> OVERRIDE IN TEST <code>expect(locator).toBeVisible({ timeout: 10_000 })</code> |

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 <code>{ use: { actionTimeout: 10_000 } }</code> OVERRIDE IN TEST <code>locator.click({ timeout: 10_000 })</code> |
| Navigation timeout | no timeout | Timeout for each navigation action SET IN CONFIG <code>{ use: { navigationTimeout: 30_000 } }</code> OVERRIDE IN TEST <code>page.goto('/', { timeout: 30_000 })</code> |
| Global timeout | no timeout | Global timeout for the whole test run SET IN CONFIG <code>{ globalTimeout: 3_600_000 }</code> |
| <code>beforeAll/afterAll</code> timeout | 30_000 ms | Timeout for the hook SET IN HOOK <code>test.setTimeout(60_000)</code> |
| Fixture timeout | no timeout | Timeout for an individual fixture SET IN FIXTURE <code>{ scope: 'test', timeout: 30_000 }</code> |

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 Prüfungen 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 |
|---|-----------------------------|
| <code>expect(locator).toBeAttached()</code> | Element is attached |
| <code>expect(locator).toBeChecked()</code> | Checkbox is checked |
| <code>expect(locator).toBeDisabled()</code> | Element is disabled |
| <code>expect(locator).toBeEditable()</code> | Element is editable |
| <code>expect(locator).toBeEmpty()</code> | Container is empty |
| <code>expect(locator).toBeEnabled()</code> | Element is enabled |
| <code>expect(locator).toBeFocused()</code> | Element is focused |
| <code>expect(locator).toBeHidden()</code> | Element is not visible |
| <code>expect(locator).toBeInViewport()</code> | Element intersects viewport |
| <code>expect(locator).toBeVisible()</code> | 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
