Playwright

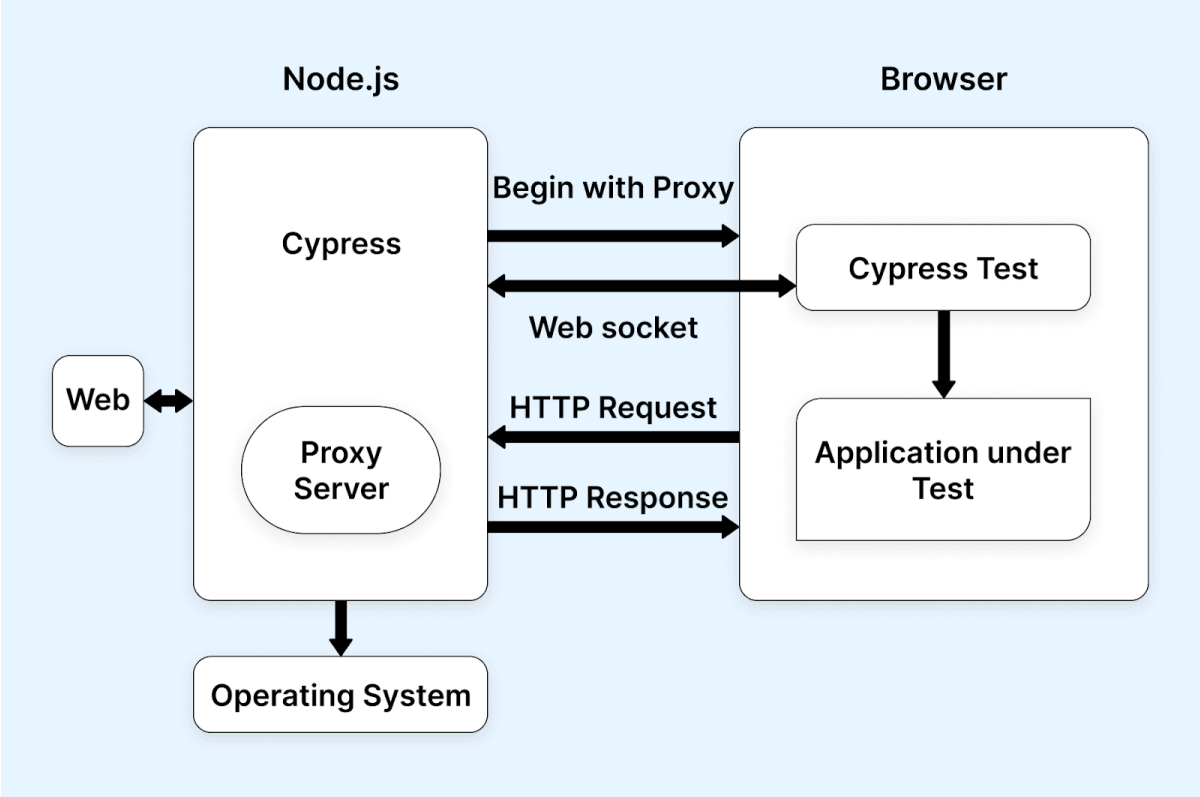
Playwright ist ein Open Source Cross-browser Testing Framework, welches von Microsoft entwickelt wird. Dabei unterstützt Playwright neben Chromium auch Firefox und Webkit. Neben Typescript werden auch Javascript, Python, .Net und Java unterstützt, um Tests zu schreiben. Tests können dabei parallel ausgeführt werden, was die Laufzeit gegenüber Cypress deutlich verringert.

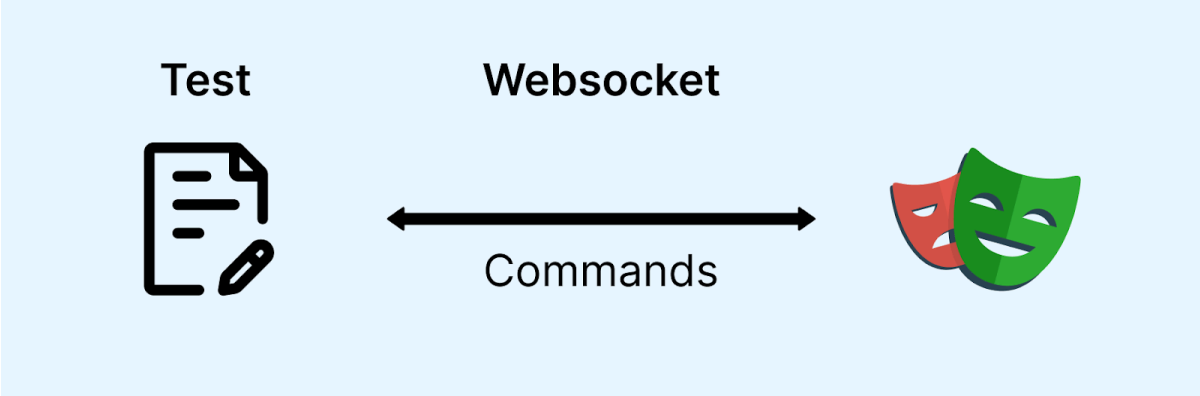
▼ Playwright VS Cypress

	Playwright	Cypress
Test Ausführung	Outside the browser mit Hilfe von Chrome Devtools Protocol (CDP)	Inside the browser in dem der Test-Code direkt in den Browser injected wird und diese in einem iFrame ausführt
Unterstütze Browser	Chromium, Firefox, Webkit	Chromium, Firefox, Webkit
Unterstütze Sprachen	JS, TS, .NET, Java, Python	JS, TS
Auto-wait	async/await	Declarative Asynchronität
Selection von Elementen	Über Locator	Über Queries (jquery)
Assertions	async / await	Explicit Assertions
Paralellisierung	for free	kostenpflichtig mit dem Cypress Dashboard
Entwickelt von	Microsoft	OS

	Playwright	Cypress
Downloads	Überholt Cypress	Abgelöst von Playwright

Architekturvergleich





▼ Setup von Playwright

Systemanforderungen

Die aktuellen Systemanforderungen können der Dokumentation entnommen werden, da sich diese ständig Ändern können:
https://playwright.dev/docs/intro#system-requirements

Installation von Playwright

Playwright stellt ein Setup-Script bereit, welches per npm ausgeführt wird. Das Script installiert die benötigten Dependencies und erzeugt einiger Beispieltests

```
npm init playwright@latest
```

Anschließend kann ein npm-Script definert werden um die Tests auszuführen

```
...
"e2e": "npx playwright test",
...
```

ESLint-Regeln

Um Playwright Tests nach den best practice Regeln zu schreiben, empfiehlt es sich das Community ESLint-Plugin zu verwenden:
https://github.com/playwright-community/eslint-plugin-playwright

Playwright selbst definiert empfohlene Regeln in der Dokumentation:
https://playwright.dev/docs/best-practices#lint-your-tests

Update der Browser

Um die Browser zu aktualisieren, gegen welche die Tests ausgeführt werden sollten,

lässt sich folgender Befehl ausführen

```
npm install -D @playwright/test@latest
# Also download new browser binaries and their dependencies:
npx playwright install --with-deps
```

Konfiguration

Playwright erzeugt nach der Initialisierung eine `playwright.config.ts` . In dieser werden alle nötigen Konfigurationen wie Browser, Testumgebungen oder Reporter konfiguriert. In diesem Abschnitt werden die wichtigsten Konfigurationen zusammengefasst. Alle weiteren Optionen können der Dokumentation entnommen werden: https://playwright.dev/docs/test-configuration

Worker

Playwright zeichnet sich vor allem durch seine Fähigkeit aus, Tests parallel auszuführen. Dabei können diese nicht nur auf unterschiedliche Threads sondern auch auf mehreren Maschinen verteilt ausgeführt werden. Standardmäßig führt Playwright Tests nur auf einem Worker aus.

Playwright führt dabei Test-Files parallel aus. Ist dies nicht gewünscht, kann diese in den entsprechenden Files angepasst werden. **Die Tests werden dabei in der im File definierten Reihenfolge durchgeführt.** Ebenfalls werden die Spec-Files in der Reihenfolge ausgeführt wie diese im Filesystem abgelegt werden.

Um Testdaten oder die Authentifizierung zwischen den einzelnen Workern zu isolieren, müssen entsprechende Anpassungen durchgeführt werden. Diese können der Dokumentation entnommen werden: https://playwright.dev/docs/test-parallel#isolate-test-data-between-parallel-workers

Reporter

Playwright ermöglicht es unterschiedliche Reports zu erzeugen, je nach Anwendungsfall (CI-Report oder lesbar für Menschen als HTML-Report). Es lassen sich mehrere Reports erzeugen oder je nach Umgebung anpassen. Standardmäßig wird ein HTML-Report erzeugt. Eigene Reporter können ebenfalls bequem in TypeScript geschrieben werden: https://playwright.dev/docs/test-reporters#custom-reporters

In CI-Umgebungen sollte der Output reduziert werden. Ein für Menschen lesbarer Report kann anschließend als Artifact ausgegeben werden. So kann unter anderem GitHub oder Azure DevOps die erzeugten Reports grafisch

darstellen: https://ultimateqa.com/playwright-reporters-how-to-integrate-with-azure-devops-pipelines/

Base URL & Proxy

Die Base-URL gibt vor, gegen welche URL die Tests letztendlich ausgeführt werden sollen. Dies kann der localhost sein als auch eine Remote-URL. In manchen Fällen wird ein Proxy benötigt um z.B. API-Requests entsprechend ausführen zu können. Dieser kann ebenfalls definiert werden.

Info: wird ein Proxy verwendet ist es wichtig den localhost nicht durch den Proxy zu Routen. Dies kann mit Hilfe des bypass Parameters konfiguriert werden.

```
proxy: {
    server: process.env['http_proxy'] ?? process.env['HTTP_PROXY'] ?? '',
    bypass: 'localhost',
},
```

Web Server

Soll vor der Ausführung der Tests die lokale Testumgebung mit z.B. der Angular-CLI gestartet werden, lässt sich dies in der webServer Konfiguration anpassen. Wichtig ist neben dem auszuführenden Script die URL oder der Port auf welchen Playwright hören soll. Ist die Webseite unter der URL oder Port erreichbar, startet Playwright mit der Ausführung der Tests.

Info: Nach meiner Erfahrung macht es mehr Sinn nach dem Port zu schauen statt der URL da dies schon mehrfach Lokal fehlgeschlagen ist.

Projects

In der Projects Konfiguration lassen sich neben den zu verwendeten Browsern Testumgebungen definieren. So ist es mögliche Tests wie z.B. Smoke-Tests oder Mobile-Tests zu gruppieren und mit angepassten Konfigurationen (Endpoint, Retry etc.) starten.

Zusätzlich lassen sich Abhängigkeiten wie Setup-Prozesse, aufsetzen einer Test-DB usw. definieren, welche parallel vor den eigentlichen Tests ausgeführt

werden.

Weitere Beispiele finden sich in der Dokumentation:
https://playwright.dev/docs/test-projects

Info: Neben der Gruppierung von Tests mit Hilfe der Projects, lassen diese sich auch Taggen.

Timezones und Locals

Beim Testen von Daten ist es empfehlenswert eine entsprechende Zeitzone zu definieren. Ebenfalls kann die zu verwendende Sprache des Browsers definiert werden

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

export default defineConfig({
  use: {
    // Emulates the user locale.
    locale: 'en-GB',

    // Emulates the user timezone.
    timezoneId: 'Europe/Paris',
  },
});
```

▼ Schreiben von Tests

In diesem Abschnitt wird beschrieben, wie man Tests in Playwright definiert und auf welche Dinge man dabei achten sollte

▼ Testaufbau

Genau wie Unit-Tests werden Playwright Tests in drei Schritten definiert. Arrange, Act, Assert. Ebenfalls ist eine Gruppierung von Tests möglich

▼ Arrange

Wie in Jest/Jasmine lassen sich vor allen Tests jeweils Before bzw. After Schritte definieren. Dies ist vor allem dann hilfreich, wenn vor jedem Test

z.B. auf eine entsprechende Seite verwiesen wird oder nach einem Test die Daten in der Datenbank zurückgesetzt werden müssen.

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

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

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

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

▼ Actions

▼ Navigation

Ein Tests fängt klassischerweise damit an, dass auf eine entsprechende Seite navigiert werden soll. Abhängig von der in der Konfiguration definierten Base-URL, wird lediglich der Pfad ausgehend davon angegeben

```
await page.goto('/home');
```

▼ Interactions

Um mit der Web-App zu interagieren, bietet Playwright unterschiedliche APIs an. Letztendlich wird hierfür die locator Funktion aufgerufen bzw. APIs welche darauf aufbauen.

```
await page.locator('text=Search').click();
await page.getByTestId('search-button').click();
```

```
await page.getByLabel('User Name').fill('John');
await page.getByLabel('Password').fill('secret-password');
await page.getByRole('button', { name: 'Sign in' }).click();
```

Playwright folgt der Philosophie, test so zu schreiben, wie ein User mit der Web-App auch interagieren würde. Deshalb empfiehlt es sich in der Praxis mit Hilfe der vorgebenen Selektoren, Elemente anhand ihrer Rolle sowie von Texten zu finden und mit diesen zu interagieren.

Die Suche nur nach einem Test wird nicht empfohlen. Lediglich bei nicht interaktiven Elemente.

> We recommend using text locators to find non interactive elements like `div`, `span`, `p`, etc. For interactive elements like `button`, `a`, `input`, etc. use role locators.

Info: Nach jeder ausgeführten Aktion, führt Playwright die Selektion des definierten Elements erneut aus um sicher zu stellen, dass das Element neu gerendert wurde und der entsprechende State gesetzt ist. Dies unterscheidet Playwright von Cypress.

Info: Meiner Meinung nach ist es ebenfalls okay, Tests mit Hilfe von Test-Ids zu definieren. Dies hat sich zumindest in der Vergangenheit in Cypress oder Integrationstest als nützlich erwiesen. Playwright selbst führt aber an, dass Tests dadurch nicht mehr einem User-Driven Ansatz folgen.

Workarounds mit CSS und XPath

Es ist dennoch möglich, Elemente anhand von CSS-Klassen oder einem XPath anzusprechen. Dies sollte aber nur im absoluten Notfall verwendet werden: https://playwright.dev/docs/locators#locate-by-css-or-xpath

Filter

Um ein Element aus einer Menge von Elementen herauszufiltern ist es möglich, filter auf die Locator anzuwenden. Playwright bietet hierfür eine Vielzahl von Filtern an.

```
await page
    .getByRole('listitem')
    .filter({ has: page.getByRole('heading', { name: 'Product 2' }) })
    .getByRole('button', { name: 'Add to cart' })
    .click();
```

Locator Operatoren

Locator können miteinander verknüpft werden um Elemente zu finden.

```
const button = page.getByRole('button').and(page.getByTitle('Subs
```

Nur sichtbare Elemente finden

Es kann sinnvoll sein Elemente zu filtern, welche sichtbar sind, zum Beispiel falls eine UI-Lib Elemente lediglich in der Visibility ausblendet und nicht aus dem DOM entfernt.

```
await page.locator('button').locator('visible=true').click();
```

▼ Basic Actions

Playwright unterstützt alle benötigten Interaktionen mit Elementen wie Klicken, Tippen, Hovern usw. Eine Übersicht der Actions findet sich in der Dokumentation: https://playwright.dev/docs/writing-tests#basic-actions

▼ Assertions

Playwright verwendet wie auch Jasmin oder Jest eine Expect-Function und ein Set von Assertions, um Variablen, Locator usw. zu prüfen. Wichtig dabei ist der Unterschied zwischen Auto-retrying und Non-retrying Assertions.

```
expect(success).toBeTruthy();
await expect(page.getByTestId('status')).toHaveText('Submitted');
```

Auto-retrying Assertions

Die entsprechenden Assertions versuchen solange auf ein Element/Eine Bedingung zu warten, bis diese erfüllt wird. Wichtig ist hier, dass mit einem `await` auf die Ausführung gewartet wird. Das Timeout beträgt 5 Sekunden: https://playwright.dev/docs/test-assertions#auto-retrying-assertions

Non-retrying Assertions

Die entsprechenden Assertions warten nicht, bis die entsprechende Bedingung erfüllt wird. Eine Verwendung dieser kann zu flaky Tests führen und sollten daher vermieden werden: https://playwright.dev/docs/test-assertions#non-retrying-assertions

▼ Soft Assertion

Es ist möglich Assertions als soft zu definieren. Tests werden dadurch nicht abgebrochen jedoch weiterhin als Fehlerhaft markiert. Es ist möglich auf zwei Arten Tests als Soft zu definieren

```
// Flavor I
await expect.soft(page.getByText('Hello World')).toBeVisible();

// Flavor II
import { expect as baseExpect, test } from '@playwright/test';

const expect = baseExpect.configure({ soft: true });
await expect.soft(page.getByText('Hello World')).toBeVisible();
```

▼ Test Gruppierung, Tags und Annotations

Playwright ermöglicht es Tests zu gruppieren als auch zu tagen. Dies macht Tests nicht nur lesbarer sondern hilft auch, Tests anhand der Tags durchzuführen. Dies ist vor allem Dann sinnvoll, wenn in der CI z.B. nur Smoke-Tests oder fast-failing Tests ausgeführt werden sollten, um die Laufzeit so kurz wie möglich zu halten

Gruppierung

Die Gruppierung von Tests erfolgt ähnlich wie in Jest oder Jasmine mit Hilfe von describe-Blocks

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

test.describe('Search Page', () => {
  test.describe('smoke', () => {
  });

  test.describe('navigation', () => {
  });

  test.describe('api interaction', () => {
  });
});
```

Tag Tests

Durch das Vergeben von Tags, lassen sich Tests später gezielt starten. Dies ist vor allem Sinnvoll um PRs später schnell prüfen zu können (fail fast).

```
test('smoke', {
  tag: ['@fast', '@smoke'],
}, async ({ page }) => {
  // ...
});

test('api interaction', {
  tag: ['@slow', '@api'],
}, async ({ page }) => {
```

```
  // ...
});
```

Anschließend können die Test wie folgt ausgeführt werden

```
npx playwright test --grep-invert @fast
npx playwright test --grep "@fast|@slow"
npx playwright test --grep "(?=.*@fast)(?=.*@slow)"
```

Annotations

Wird beispielsweise ein Test geschrieben, welcher einen gemeldeten Bug testet um Regressionen zu vermeiden, kann dieser mit einer Annotation versehen werden.

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

test('test login page', {
  annotation: {
    type: 'issue',
    description: 'https://github.com/microsoft/playwright/issues/23180',
  },
}, async ({ page }) => {
  // ...
});
```

▼ Page Object Models

Page Object Models (POM) sind wiederverwendbare Klassen einer isolierten Seite. Sie helfen dabei Tests zu vereinfachen und vor allem wartungsarm zu gestallten. So lassen sich Selektoren als Variablen definieren und einfach aufrufen.

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

export class PlaywrightDevPage {
  readonly page: Page;
```

```
  readonly getStartedLink: Locator;
  readonly gettingStartedHeader: Locator;

  constructor(page: Page) {
    this.page = page;
    this.getStartedLink = page.locator('a', { hasText: 'Get started' });
    this.gettingStartedHeader = page.locator('h1', { hasText: 'Installation' });
  }

  async goto() {
    await this.page.goto('https://playwright.dev');
  }

  async getStarted() {
    await this.getStartedLink.first().click();
    await expect(this.gettingStartedHeader).toBeVisible();
  }
}
```

▼ Fixtures

Fixtures helfen eine Testumgebung für Tests zu definieren und um wiederverwendeten Code zu reduzieren, ähnlich wie die Custom Commands in Cypress. Fixtures lassen sich zudem mit POMs verbinden um diese in einem Test aufzurufen.

```
import { test as base } from '@playwright/test';
import { LicenceListPage } from './licences/licence-list.pom';
import { LicenceCreatePage } from './licences/license-create.pom';

interface MyFixtures {
  licencesPage: LicenceListPage;
  licencesCreatePage: LicenceCreatePage;
}

export const test = base.extend<MyFixtures>({
```

```
  licencesPage: async ({ page }, use) => {
    await use(new LicenceListPage(page));
  },
  licencesCreatePage: async ({ page }, use) => {
    await use(new LicenceCreatePage(page));
  },
});

export { expect } from '@playwright/test';
```

Anschließen wird das Fixture in den Tests importiert um diese zu definieren. In komplexeren Projekten empfiehlt es sich, einzelne Fixtures für Teilbereiche einer Anwendung zu erstellen um die Übersichtlichkeit zu gewährleisten.

▼ Netzwerk Interactions

Playwright verfolgt das Ziel User-Interaktionen und den damit verbundenen Output der Web-App zu testen. Dennoch ist es teilweise nötig, Requests an die API oder 3rd Party Abhängigkeiten ebenfalls mit in die Testfälle aufzunehmen. Playwright ermöglicht hierbei nicht nur auf Requests zu warten sondern diese auch zu Intercepten. Damit ist es möglich, Testdaten bereitzustellen, falls dies nicht mit Hilfe einer Test-DB möglich sein sollte.

Mocking von Requests

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

test.beforeEach(async ({ context }) => {
  await context.route(/.css$/, route => route.abort());
});

test('loads page without css', async ({ page }) => {
  await page.goto('https://playwright.dev');
});
```

Warten auf Requests/Responses

```
test('list should render data', async ({ page }) => {
  const request = page.waitForResponse((response) => {
    return response.url() === 'https://api.foo.com/bar';
  });
  await page.goto('');
  await expect(page.getByTestId('card')).toHaveCount(9);
  await request;
});
```

Intercepten und modifien von Requests

```
test('it should render data', async ({ page }) => {
  await page.route("https://api.foo.com/bar", (route) =>
    route.fulfill({
      status: 200,
      json: [
        {
          ...
        },
      ],
    })
  );
});
```

▼ Authentication / Session Storage

Playwright bietet ähnlich wie Cypress die Möglichkeit, einen Auth-Flow zu definieren und die im Browser abgelegten Cookies sowie Localstorage Properties zu speichern. Somit muss der Login einmalig vor der Ausführung der eigentlichen Tests durchgeführt und nicht mehr wiederholt werden. Da der Flow recht ausführlich beschrieben ist, kann dieser der Dokumentation entnommen werden: https://playwright.dev/docs/auth

Es können im Setup bereits mehrere User angemeldet und die Ergebnisse gespeichert werden. Dies ist vor allem dann sinnvoll, wenn es Benutzer mit unterschiedlichen Rollen gibt.

Info: Der Inhalt der dabei erzeugten Datei ist immer auf das Origin gescoped, unter welchem der Login stattfand. Anderenfalls muss die Origin händisch angepasst werde.

Authentication per API-Request

Gibt es eine API um den User anzumelden und einen Token abzurufen, lässt sich der Auth-Flow vereinfachen. Mit Hilfe der Request API lässt sich der Login ausführen und den Response speichern.

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

const authFile = 'playwright/.auth/user.json';

setup('authenticate', async ({ request }) => {
  // Send authentication request. Replace with your own.
  await request.post('https://github.com/login', {
    form: {
      'user': 'user',
      'password': 'password'
    }
  });
  await request.storageState({ path: authFile });
});
```

Tests ohne Authentifizierung

Wenn die Authentifizierung in einzelnen Test ignoriert werden soll, z.B. um die Login-Maske gesondert zu testen, kann dies ebenfalls definiert werden

```
test.use({ storageState: { cookies: [], origins: [] } });
```

▼ Test Debugging

Playwright Tests lassen sich auf unterschiedliche Weisen debuggen. Die einfachste Art ist über das Terminal den Debug Modus zu starten. Dabei öffnet sich ein Browser Fenster in welchem man die Tests Schritt für Schritt durchlaufen lassen kann: https://playwright.dev/docs/debug

▼ Debug Modus

Playwright Tests lassen sich auf unterschiedliche Weisen debuggen. Die einfachste Art ist über das Terminal den Debug Modus zu starten. Dabei öffnet sich ein Browser Fenster in welchem man die Tests Schritt für Schritt durchlaufen lassen kann.

```
npx playwright test --debug
```

Um eine ähnliche UI wie in Cypress anzuzeigen, kann folgender Befehl im Terminal ausgeführt werden

```
npx playwright test --ui
```

▼ VSCode Debugger

Playwright selbst empfiehlt das entsprechende VSCode Plugin zu verwenden um direkt in der IDE Test zu debuggen.

▼ Trace Viewer

Die Trace View ist das Debugging Tool von Playwright. Über dieses lassen sich Locator, Snapshots, Screenshots, Netzwerk sowie die Console betrachten und einzelne Tests debuggen. Letzendlich handelt es sich dabei um den UI-Mode.

```
npx playwright test --trace on
```

▼ Visuelle Regression & Screenshots

Playwright bietet die Möglichkeit Screenshot-Tests auszuführen. Diese sind besonders bei der Verwendung von CSS-Frameworks oder eigens

entwickelten UI-Komponenten sinnvoll. Ein großer Vorteil ist die Geschwindigkeit solcher Tests.

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

test('example test', async ({ page }) => {
  await page.goto('https://playwright.dev');
  await expect(page).toHaveScreenshot();
});
```

Dabei funktionieren die Screenshots ähnlich wie Snapshot-Tests mit Jests. Screenshots müssen explizit aktualisiert werden, falls sich die UI geändert hat und man die Tests entsprechend anpassen muss.

```
npx playwright test --update-snapshots
```

Screenshots können außerdem dazu verwendet werden, Reports zu ergänzen und die Testergebnisse zu validieren.

```
// page
await page.screenshot({ path: 'screenshot.png' });
// full page
await page.screenshot({ path: 'screenshot.png', fullPage: true });
// one specific Element
await page.locator('.header').screenshot({ path: 'screenshot.png' });
```

▼ Test Generator

Um schnell Tests zu definieren, kann der Testgenerator verwendet werden. Es wird jedoch nicht empfohlen damit generierte Tests ohne Anpassung zu verwenden: https://playwright.dev/docs/codegen

Chrome Recorder

Playwright stellt ein Chrome Plugin bereit, mit welchem man Tests direkt in Chrome aufzeichnen und als Playwright Test exportieren kann:

https://chromewebstore.google.com/detail/playwright-chrome-recorde/bfnbgoehgplaehdceponclakmhlgjlpd?pli=1

CI

Die CI Konfiguration kann der Dokumentation entnommen werden. Letztendlich muss entschieden werden, wie die Tests parallel ausgeführt werden sollen und welche Reporter die jeweilige CI Umgebung unterstützt:
https://playwright.dev/docs/ci