Testing

- Oberflächentests mit Playwright

Oberflächentests mit Playwright

Einleitung

Oberflächentests (UI-Tests) stellen sicher, dass Webanwendungen sich so verhalten, wie es Benutzer erwarten – unabhängig vom internen Code. Mit **Playwright** steht ein modernes, mächtiges Test-Framework zur Verfügung, das speziell für dynamische Web-UIs entwickelt wurde. Dieser Artikel stellt Playwright vor, zeigt seinen Einsatz in einem typischen Projekt mit SvelteKit und vergleicht es mit Alternativen wie Cypress und Selenium. Darüber hinaus werden praktische Strategien zur Teststruktur, Fehlervermeidung und Debugging vermittelt.

Was ist Playwright?

Playwright ist ein von Microsoft entwickeltes Open-Source-Tool für **UI-Tests** von Webanwendungen. Tests werden in echten Browserinstanzen (Chromium, Firefox, WebKit) ausgeführt und über DevTools-Protokolle gesteuert. Besonders hervorzuheben sind:

- Unterstützung mehrerer Browser
- Automatische Warte-Mechanismen
- Netzwerk-Interception, Screenshot, Tracing, Debugging

Typischer Playwright-Workflow

Playwright erkennt standardmäßig alle Testdateien mit den Endungen `.spec.ts` oder `.test.ts`. Es ist also üblich, Testdateien nach dem Schema `xyz.spec.ts` zu benennen. Dies ist keine Pflicht, aber eine empfohlene Konvention, da sie vom Test-Runner automatisch erkannt wird.

Ein vollständiger Testdurchlauf mit Playwright besteht in der Regel aus drei Schritten:

1. **Test schreiben**
   - Der Test wird als JavaScript- oder TypeScript-Datei im `e2e/`-Verzeichnis abgelegt.
   - Beispiel:

```
test('zeigt Loginformular', async ({ page }) => {
  await page.goto('/login');
  await expect(page.locator('form')).toBeVisible();
});
```

2. **Test ausführen**
   - Im Terminal ausführen:

     ```
     npx playwright test
     ```

   - Optional: Nur bestimmte Tests oder Dateien ausführen:

     ```
     npx playwright test login.spec.ts
     ```

3. **Ergebnisse analysieren**
   - Playwright gibt im Terminal an, ob Tests bestanden oder fehlgeschlagen sind.
   - Bei Fehlern:
     - Trace-Dateien oder Screenshots analysieren
     - Optional: `--debug` oder `--trace on` verwenden

     Beispiel:

     ```
     npx playwright show-trace trace.zip
     ```

Dieser Workflow ist die Grundlage für einfache lokale Tests, funktioniert aber auch identisch in CI/CD-Umgebungen (z. B. GitHub Actions).

Automatische Warte-Mechanismen

Playwright wartet **intelligent und automatisch** auf Ereignisse im DOM. Das bedeutet: Wenn du z. B. ein Element anklickst oder eine Seite neu lädst, wartet Playwright so lange, bis die Seite bereit ist (Ladezustand, Sichtbarkeit, Interaktivität), **ohne dass du explizit `wait`-Befehle schreiben musst**. Das reduziert Fehler durch Timing-Probleme erheblich und macht Tests stabiler als bei älteren Tools wie Selenium.

Beispiel:

```
await page.click('text=Speichern'); // wartet intern, bis der Button sichtbar & klickbar ist
```

Netzwerk-Interception

Mit Playwright kannst du **HTTP-Anfragen abfangen und manipulieren**, bevor sie den Server erreichen oder nachdem sie empfangen wurden. Das ermöglicht z. B.:

- das **Mocken von Backend-Antworten**

- das **Blockieren von Requests** (z. B. zu externen APIs)
- das **Simulieren von Fehlerzuständen** (z. B. 500er-Fehler)

Beispiel:

```
await page.route('**/api/user', route => {
  route.fulfill({ status: 200, body: JSON.stringify({ name: 'Testuser' }) });
});
```

Screenshot

Playwright kann jederzeit einen Screenshot vom aktuellen Zustand der Seite aufnehmen – z. B. zur **Fehlerdokumentation**, zur visuellen Regression oder einfach als Debugging-Hilfe. Das ist besonders nützlich bei CI-Testläufen, wenn ein Test fehlschlägt.

Beispiel:

```
await page.screenshot({ path: 'screenshot.png', fullPage: true });
```

Tracing

Das **Tracing-System von Playwright** erstellt eine vollständige Aufzeichnung eines Testlaufs. Die generierte Datei (`trace.zip`) kann mit dem Trace Viewer analysiert werden. Dieser zeigt:

- DOM-Zustände und Screenshots nach jeder Aktion
- Klickpfade und Zeitverlauf
- Netzwerkanfragen und -antworten
- Konsolenausgaben und Fehler

Der Trace Viewer eignet sich ideal zum Debuggen, wenn Tests unerwartet fehlschlagen oder das Timing von UI-Interaktionen unklar ist.

Verwendung:

```
npx playwright test --trace on
npx playwright show-trace trace.zip
```

Das Tool ist interaktiv und CI-fähig – es gehört zu den zentralen Debugging-Funktionen von Playwright.

HTML-Report

Playwright enthält standardmäßig einen HTML-Reporter, der nach einem Testlauf einen interaktiven Bericht erzeugen kann.

Verwendung:

```
npx playwright test --reporter=html
npx playwright show-report
```

Dabei entsteht ein Verzeichnis `playwright-report/`, in dem die Testergebnisse als durchklickbarer HTML-Bericht gespeichert werden. Der Report enthält:

- Status jedes Tests (✔ / ✖)
- Detaillierte Fehlerbeschreibung (Stacktrace)
- Screenshots bei Fehlern
- Trace-Links, sofern aktiviert

Hinweis: Reporter lassen sich auch dauerhaft in `playwright.config.ts` setzen:

```
reporter: ['list', 'html']
```

Der HTML-Report eignet sich besonders gut für Reviews, Testprotokolle oder zur Weitergabe an Stakeholder.

Das **Tracing-System von Playwright** erstellt eine vollständige Aufzeichnung eines Testlaufs. Die generierte Datei (`trace.zip`) kann mit dem Trace Viewer analysiert werden. Dieser zeigt:

- DOM-Zustände und Screenshots nach jeder Aktion
- Klickpfade und Zeitverlauf
- Netzwerkanfragen und -antworten
- Konsolenausgaben und Fehler

Der Trace Viewer eignet sich ideal zum Debuggen, wenn Tests unerwartet fehlschlagen oder das Timing von UI-Interaktionen unklar ist.

Verwendung:

```
npx playwright test --trace on
npx playwright show-trace trace.zip
```

Das Tool ist interaktiv und CI-fähig – es gehört zu den zentralen Debugging-Funktionen von Playwright.

Das **Tracing-System von Playwright** erstellt einen kompletten „Film“ eines fehlgeschlagenen Testlaufs: Es enthält Informationen über:

- alle DOM-Änderungen
- Netzwerk-Anfragen
- Konsolenmeldungen

- Screenshots und Zeitleisten

Diese Daten können im **Playwright Trace Viewer** visuell nachvollzogen werden – ideal zum Debuggen komplexer Fehler in CI/CD.

```
npx playwright test --trace on
npx playwright show-trace trace.zip
```

Debugging

Zusätzlich zum Tracing bietet Playwright u. a.:

- einen **Inspector** (GUI zum „Step-by-Step-Debuggen“ im Browser)
- `codegen` zur **Aufzeichnung von Tests**
- **Pause-Funktion** für interaktives Testen (`page.pause()`)

Beispiel:

```
await page.pause(); // öffnet Debugger-Fenster mit DOM-Explorer
```

Integration in moderne Frameworks

Ein großer Pluspunkt ist die **nahtlose Integration in moderne Toolchains**. Bei der Installation eines SvelteKit-Projekts wird Playwright beispielsweise direkt mit angeboten. Es ist kein manueller Konfigurationsaufwand nötig.

Typische Teststruktur

Playwright kann Tests in mehreren Browsern ausführen – etwa in Chromium, Firefox oder WebKit. Wird in der `playwright.config.ts` jedoch kein `projects`-Abschnitt definiert, wie z. B. im SvelteKit-Starter, verwendet Playwright **standardmäßig nur Chromium**. Das reicht für viele Szenarien – schränkt aber die Cross-Browser-Testbarkeit ein.

Möchte man Tests zusätzlich in Firefox und WebKit durchführen, lässt sich die Konfiguration um `projects` erweitern:

```
projects: [
  { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
  { name: 'firefox',  use: { ...devices['Desktop Firefox'] } },
  { name: 'webkit',   use: { ...devices['Desktop Safari'] } },
```

```
]
```

Playwright führt die Tests dann automatisch für alle angegebenen Browser durch – entweder nacheinander oder parallel, je nach Konfiguration. So lassen sich UI-Komponenten und Workflows direkt in allen wichtigen Rendering-Engines verifizieren.

Playwright führt die Tests dann automatisch für alle angegebenen Browser durch – entweder nacheinander oder parallel, je nach Konfiguration. So lassen sich UI-Komponenten und Workflows direkt in allen wichtigen Rendering-Engines verifizieren.

Playwright-Konfigurationen werden in der Datei `playwright.config.ts` definiert. Besonders praktisch ist dabei die Möglichkeit, den lokalen Server automatisch zu starten, bevor die Tests ausgeführt werden:

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

export default defineConfig({
  webServer: {
    command: 'npm run build && npm run preview',
    port: 4173
  },
  testDir: 'e2e'
});
```

- **`command`** startet den lokalen Server, z. B. den SvelteKit-Preview-Modus.
- **`port`** legt fest, wo der Test-Runner auf die Anwendung wartet.
- **`testDir`** gibt das Verzeichnis an, in dem die Tests gespeichert sind.

Diese automatische Startlogik ist besonders hilfreich für CI/CD, da kein separater Serverlauf erforderlich ist. Wer möchte, kann hier auch eigene Kommandos oder Ports verwenden – etwa um eine andere Umgebung zu testen oder eine API-Instanz parallel zu starten.

Playwright sucht standardmäßig in dem Ordner nach Tests, in dem sich deine `playwright.config.ts` befindet. Zudem erkennt es von sich aus alle Dateien mit den Endungen `.spec.ts`, `.test.ts`, `.spec.js`, `.test.js`.

Viele Framework-Vorlagen – z. B. SvelteKit – legen hierfür das Verzeichnis **`e2e/`** an. Dies ist aber keine Vorgabe seitens Playwright, sondern eine Empfehlung, um **Frontend- und End-to-End-Tests (E2E)** strukturiert zu trennen.

Du kannst stattdessen auch beliebige andere Verzeichnisse nutzen (z. B. `tests/`, `ui-tests/`), solange sie sich im selben Verzeichnis wie `playwright.config.ts` befinden – oder du `testDir` in der Konfiguration entsprechend anpasst.

Nach dem Setup legt Playwright ein `e2e/`-Verzeichnis an (z. B. durch SvelteKit). Ein einfacher Test könnte so aussehen:

```
test('zeigt ein leeres Board mit 6x7 Zellen', async ({ page }) => {
  await page.goto('/');
  const cells = await page.locator('[data-test^=cell-]');
  await expect(cells).toHaveCount(6 * 7);
});
```

Häufig verwendete Befehle:

- `page.click(selector)` – Klick auf ein Element
- `page.fill(selector, text)` – Eingabe simulieren
- `expect(locator).toHaveText()` – Zustand prüfen
- `page.screenshot()` – Fehler sichtbar machen
- `test.beforeEach()` – Setup vor jedem Test

Nach dem Setup legt Playwright ein `e2e/`-Verzeichnis an. Ein einfacher Test könnte so aussehen:

```
test('zeigt ein leeres Board mit 6x7 Zellen', async ({ page }) => {
  await page.goto('/');
  const cells = await page.locator('[data-test^=cell-]');
  await expect(cells).toHaveCount(6 * 7);
});
```

Häufig verwendete Befehle:

- `page.click(selector)` – Klick auf ein Element
- `page.fill(selector, text)` – Eingabe simulieren
- `expect(locator).toHaveText()` – Zustand prüfen
- `page.screenshot()` – Fehler sichtbar machen
- `test.beforeEach()` – Setup vor jedem Test

Vergleich: Playwright, Cypress, Selenium

Merkmal	Playwright	Cypress	Selenium
Ausführungskontext	Außerhalb des Browsers über native DevTools-Protokolle	Im Browser	Extern via WebDriver
Sprachen	JS/TS, Python, C#, Java	Nur JS/TS	Viele (Java, Python, etc.)

Merkmal	Playwright	Cypress	Selenium
Browser-Support	Chromium, Firefox, WebKit	Chromium-basiert, Firefox (beta)	Alle, inkl. IE
Multi-Tab, Multi-Window	Voll unterstützt	Eingeschränkt	Möglich, aber komplex
Netzwerk-Interception	Einfach via `page.route()`	Eingeschränkt	Aufwendig
Debugging	Trace Viewer, Screenshots, CLI	Zeitreise im Browser, GUI	Schwerfällig
Setup-Aufwand	Minimal (bei modernen Frameworks)	Mittel	Hoch

Geeignete Einsatzzwecke

Playwright eignet sich besonders für:

- SPAs und dynamische Benutzeroberflächen
- Tests in mehreren Browsern (inkl. WebKit)
- CI/CD-Systeme mit paralleler Ausführung
- Komplexe User-Flows mit mehreren Tabs oder Netzwerkaktionen

Weniger geeignet ist Playwright für klassische Unit-Tests oder Alt-Systeme mit Internet Explorer.

Erweiterte Konzepte & Best Practices

1. Locator-Strategien

Robuste Selektoren sind entscheidend:

- `data-test="..."` als Attribut-Selektoren
- `getByRole('button', { name: 'Senden' })` für semantisch korrekte Auswahl
- CSS-Selektoren vermeiden

2. Teststruktur

- **Isolierte Tests:** Jeder Test startet mit `beforeEach()` in einer eigenen Umgebung.
- **Page Object Pattern:** UI-Interaktionen auslagern, um die Tests lesbar und wartbar zu halten.
- **Fixtures:** Wiederverwendbare Setup-Schritte in `test.extend()` kapseln.

3. Parallelisierung & Stabilität

Playwright führt Tests standardmäßig parallel aus, um die Laufzeit zu optimieren – lokal ebenso wie in CI/CD. Das bringt große Vorteile, erfordert aber, dass die Tests entsprechend gestaltet sind.

Grundprinzip:

> *Schreibe Tests so, als würden sie gleichzeitig laufen.*

Das bedeutet:

- Jeder Test sollte **unabhängig** von anderen Tests funktionieren.
- Es darf keine geteilten Daten, Zustände oder Seiteneffekte zwischen Tests geben.
- Testdaten sollten innerhalb des Tests erzeugt und genutzt werden.
- Falls Setup nötig ist: mit `beforeEach()` arbeiten, nicht mit geteilten Variablen.

Typisches Problem: Zwei Tests nutzen denselben Nutzer oder Datensatz – je nach Reihenfolge oder Timing schlägt einer davon fehl. Das wirkt dann wie ein „flaky test“ – ist aber ein Architekturproblem.

Hilfreiche Optionen in der Konfiguration oder beim Aufruf:

- `--shard=1/3` – teilt die Tests in gleichmäßige Gruppen für parallele Ausführung
- `--retries=2` – Wiederholung fehlgeschlagener Tests (z. B. bei Netzwerkaussetzern)
- `--workers=4` – legt explizit die Anzahl paralleler Worker fest

Fazit: Parallele Tests sparen Zeit – aber nur, wenn sie wirklich unabhängig voneinander sind. Genau das sollte beim Schreiben konsequent beachtet werden.

- `--shard`, `--retries`, `--workers` nutzen für schnelle, stabile CI-Runs.
- Traces (`--trace on`) für Fehleranalyse aktivieren.

4. Netzwerkinterception & Mocking

- Mit `page.route()` lassen sich APIs mocken oder blockieren.
- Reduziert Abhängigkeiten zu externen Diensten und beschleunigt die Tests.

Weiterführende Links

- Playwright Dokumentation
- Trace Viewer
- Playwright vs Cypress
- SvelteKit Testing Guide