

Playwright Workshop

Modernes E2E-Testing leicht gemacht



Daniel Sogl - linktr.ee/daniel_sogl

Über den Trainer

Daniel Sogl

Consultant & Conference Speaker bei
Thinktecture AG

- 🎯 Spezialist für Angular und Generative AI
- ♦ MVP für Developer & Web Technologies
- 🎤 Tech Speaker & Workshop Trainer
- 📚 Regelmäßiger Konferenz-Speaker



think
tecture

Workshop Details

Organisatorisches

- **Dauer:** 3 Tage, täglich von 9:00 bis 16:00 Uhr
- **Pausen:**
 - Vormittagspause: 10:30 - 10:45 Uhr
 - Mittagspause: 12:00 - 13:00 Uhr
 - Nachmittagspause: 14:30 - 14:45 Uhr
- **Format:** Mix aus Theorie, Live-Coding und praktischen Übungen
- **Interaktion:** Fragen können jederzeit gestellt werden

Ziele des Workshops

- Verständnis von E2E-Testing und dessen Bedeutung
- Beherrschung der Playwright-API und Testmethodik
- Automatisierung komplexer Testszenarien
- Integration von Tests in CI/CD-Pipelines
- Entwicklung stabiler, wartbarer Testsuiten
- Praktische Erfahrung durch hands-on Übungen

Was Sie mitnehmen

-  Fundiertes Wissen über moderne E2E-Testing-Strategien
-  Praktische Erfahrung mit Playwright
-  Best Practices für Test-Automatisierung
-  Tipps und Tricks aus der Praxis
-  Beispiel-Code und Übungsmaterialien
-  Strategien zur CI/CD-Integration

Agenda Tag 1

Vormittag

- Einführung in Testing
 - Warum testen wir?
 - Testpyramide & moderne Teststrategien
 - End-to-End Testing im Detail
- Playwright Grundlagen
 - Was ist Playwright?
 - Installation & Setup
 - Erste Tests schreiben

Nachmittag

- Playwright in der Praxis
 - Locators & Selektoren
 - Interaktionen & Aktionen
 - Assertions & Validierung
- Debugging & Fehlerbehebung
 - Playwright Inspector
 - Trace Viewer
 - IDE-Integration
 - Übungen & praktische Anwendung

Grundlagen des Testings

Warum ist Testen wichtig?

Daniel Sogl - linktr.ee/daniel_sogl



Warum Software-Testing?

- **Qualitätssicherung:** Erkennung von Fehlern und Gewährleistung der Funktionalität
- **Risikominimierung:** Frühzeitige Identifizierung kritischer Probleme
- **Kosteneffizienz:** Günstigere Fehlerbeseitigung in frühen Entwicklungsphasen
- **Kundenzufriedenheit:** Sicherstellung einer positiven Nutzererfahrung
- **Vertrauen:** Aufbau von Vertrauen in die Anwendung
- **Kontinuierliche Verbesserung:** Basis für iterative Weiterentwicklung

Testarten im Überblick

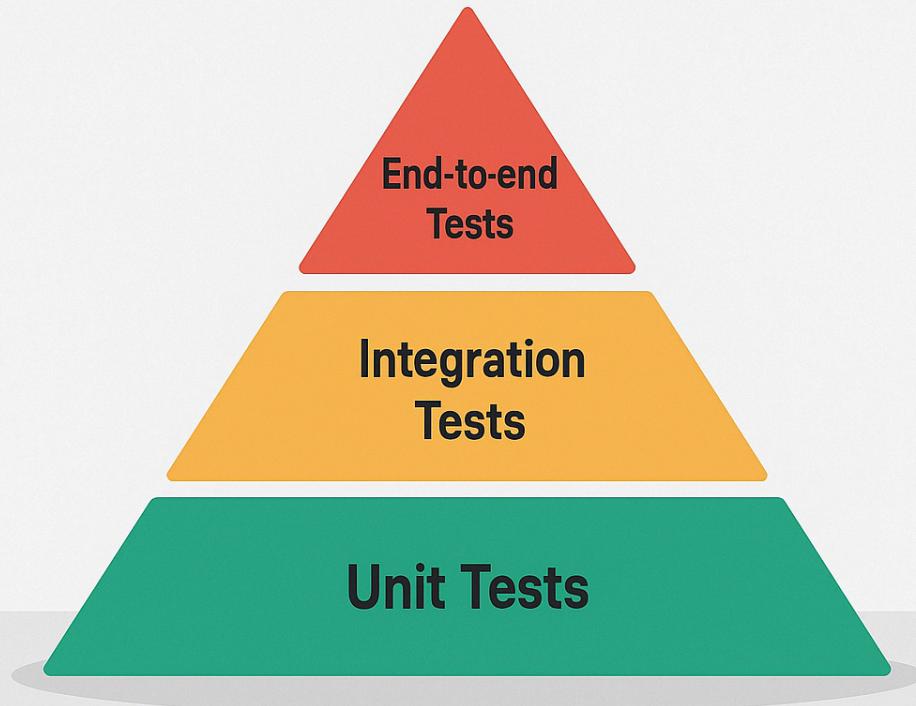
Nach Ebene

- **Unit-Tests:** Testen einzelner Komponenten/Funktionen
- **Integrations-Tests:** Testen des Zusammenspiels von Komponenten
- **System-Tests:** Testen des Gesamtsystems
- **End-to-End-Tests:** Testen ganzer Anwendungsabläufe

Nach Fokus

- **Funktional:** Überprüfung der Funktionalität
- **Nicht-funktional:** Performance, Sicherheit, Usability
- **Regression:** Sicherstellung, dass neue Änderungen keine bestehenden Funktionen beeinträchtigen
- **Akzeptanz:** Überprüfung der Anforderungserfüllung

TESTING PYRAMID



Grenzen klassischer Teststrategien

- Nicht alle Systeme passen zur Pyramidenlogik
- Mobile und Microservice-Architekturen benötigen oft andere Gewichtung
- Fokus zu stark auf Isolierung kann Integrationsfehler übersehen
- UI-Tests oft zu wenig berücksichtigt

End-to-End Testing

- Simuliert reale Benutzerinteraktionen
- Testet die gesamte Anwendung inkl. aller Systeme und Abhängigkeiten
- Validiert den gesamten Funktionsfluss der Anwendung
- **Genau hier setzt Playwright an!**

Vorteil

E2E-Tests validieren die Anwendung aus Nutzerperspektive, genau wie echte Benutzer damit interagieren würden.

Herausforderung

E2E-Tests können langsamer und anfälliger für Instabilitäten sein.

Herausforderungen beim manuellen Testing

- **Zeitaufwand:** Wiederholtes Testen ist zeitintensiv
- **Konsistenz:** Schwierigkeiten, Tests immer identisch durchzuführen
- **Skalierbarkeit:** Begrenzte Kapazität für umfangreiche Testszenarien
- **Dokumentation:** Aufwändige Protokollierung von Testergebnissen
- **Regression:** Hoher Aufwand bei wiederholten Tests nach Änderungen
- **Menschlicher Faktor:** Anfälligkeit für Fehler und Übersehen von Details

Vorteile automatisierter Tests

- **Effizienz:** Schnellere Ausführung als manuelle Tests
- **Wiederholbarkeit:** Identische Ausführung in jeder Testumgebung
- **Konsistenz:** Gleiche Testbedingungen bei jedem Durchlauf
- **Abdeckung:** Möglichkeit, mehr Testszenarien abzudecken
- **Frühe Fehlererkennung:** Schnelles Feedback in der Entwicklung
- **CI/CD-Integration:** Automatische Ausführung als Teil der Pipeline
- **Dokumentation:** Tests dienen als lebende Dokumentation

Unit-Tests – Grundlagen

Ziel: Testen kleinster, isolierter Code-Einheiten (z.B. Funktionen, Methoden, Klassen).

Fokus: Überprüfung der Logik einer einzelnen Einheit ohne externe Abhängigkeiten (oft durch Mocks/Stubs ersetzt).

- Isolierte Tests einzelner Funktionalitäten
- Externe Abhängigkeiten werden durch Mocks/Stubs ersetzt
- Basis der Testpyramide mit der größten Testanzahl
- Schnelle Ausführung und einfaches Setup

Unit-Tests – Vor- und Nachteile

Vorteile

- **Schnell:** Sehr schnelle Ausführung
- **Präzise Fehlerlokalisierung:** Fehler können leicht einer spezifischen Einheit zugeordnet werden
- **Fördert gutes Design:** Zwingt Entwickler zu modularem, entkoppeltem Code
- **Frühes Feedback:** Schnelle Rückmeldung während der Entwicklung

Nachteile

- **Keine Integrationsgarantie:** Testen nicht das Zusammenspiel verschiedener Einheiten
- **"Mock Hell":** Kann zu übermäßigem Mocking führen, das von der Realität abweicht
- **Blind für Systemfehler:** Finden keine Fehler, die nur im Gesamtsystem auftreten

Integrations-Tests – Grundlagen

Ziel: Testen des Zusammenspiels mehrerer Komponenten oder Module.

Fokus: Überprüfung der Schnittstellen und Datenflüsse zwischen integrierten Teilen des Systems.

Beispiele: Interaktion zwischen Service-Schicht und Datenbank, Kommunikation zwischen Microservices.

- **Realitätsnäher als Unit-Tests:** Testen das tatsächliche Zusammenspiel von Komponenten
- **Finden Schnittstellenprobleme:** Decken Fehler in der Kommunikation zwischen Modulen auf
- **Höheres Vertrauen:** Geben mehr Sicherheit über die Korrektheit der Integration

Integrations-Tests – Vor- und Nachteile

Vorteile

- **Realitätsnäher als Unit-Tests:** Testen das tatsächliche Zusammenspiel von Komponenten
- **Finden Schnittstellenprobleme:** Decken Fehler in der Kommunikation zwischen Modulen auf
- **Höheres Vertrauen:** Geben mehr Sicherheit über die Korrektheit der Integration

Nachteile

- **Langsamer als Unit-Tests:** Benötigen oft mehr Setup und Laufzeit
- **Schwierigere Fehleranalyse:** Fehlerursachen können komplexer zu finden sein als bei Unit-Tests
- **Abhängigkeiten:** Erfordern oft laufende externe Systeme (Datenbanken, andere Services)

E2E Tests – Überblick

Ziel: Simulation vollständiger Benutzerabläufe durch die gesamte Anwendung.

Fokus: Validierung des gesamten Systemflusses aus der Perspektive des Endbenutzers, inklusive UI, Backend, Datenbanken und externer Integrationen.

Beispiele: Einloggen, Artikel zum Warenkorb hinzufügen, Kasse, Ausloggen.

- Testen der Anwendung aus Benutzerperspektive
- Validierung vollständiger Geschäftsprozesse
- Überprüfung aller Systemkomponenten im Zusammenspiel
- Simulation realer Nutzungsszenarien

E2E Tests – Vor- und Nachteile

Vorteile

- **Höchstes Vertrauen:** Testen die Anwendung so, wie ein Benutzer sie erleben würde
- **Validieren komplette Features:** Stellen sicher, dass ganze Geschäftsabläufe funktionieren
- **Decken systemweite Probleme auf:** Finden Fehler, die nur im Zusammenspiel aller Teile auftreten

Nachteile

- **Langsam:** Deutlich längere Ausführungszeiten als Unit- oder Integrationstests
- **Instabil ("Flaky"):** Anfällig für Probleme durch Timing, Netzwerk, UI-Änderungen oder Testdaten
- **Aufwändig:** Komplexer in der Erstellung und Wartung

Teststrategie: Nicht nur Theorie!

- **Warum das Ganze?** Die richtige Strategie spart Zeit & Nerven.
- **Ziel:** Hohe Testabdeckung mit optimalem Aufwand.
- **Playwright im Kontext:** Stärkt massiv die E2E-Säule eurer Tests.
- **Diskussion:** Welche Strategie passt zu eurem Projekt? Wie kann Playwright helfen?

Kritik an der Test-Pyramide – Übersicht

Obwohl ein wertvolles Konzept, stößt die klassische Pyramide an Grenzen:

- **Überbetonung von Unit-Tests?**: Können zu viele isolierte Tests ein falsches Sicherheitsgefühl geben, wenn Integrationspunkte fehlschlagen?
- **Vernachlässigung von Integrationstests?**: Das Zusammenspiel von Komponenten ist oft fehleranfällig und wird möglicherweise unterrepräsentiert.
- **Wird E2E unterschätzt?**: Werden E2E-Tests zu stark reduziert, obwohl sie das Nutzererlebnis am besten abbilden?

Kritik an der Test-Pyramide – Moderne Anforderungen

- **Moderne Architekturen:** Passt das Modell noch perfekt zu Microservices, Frontend-Frameworks oder stark vernetzten Systemen?
- **CI/CD ermöglicht häufige und schnelle Ausführung auch komplexerer Tests**

Hinweis

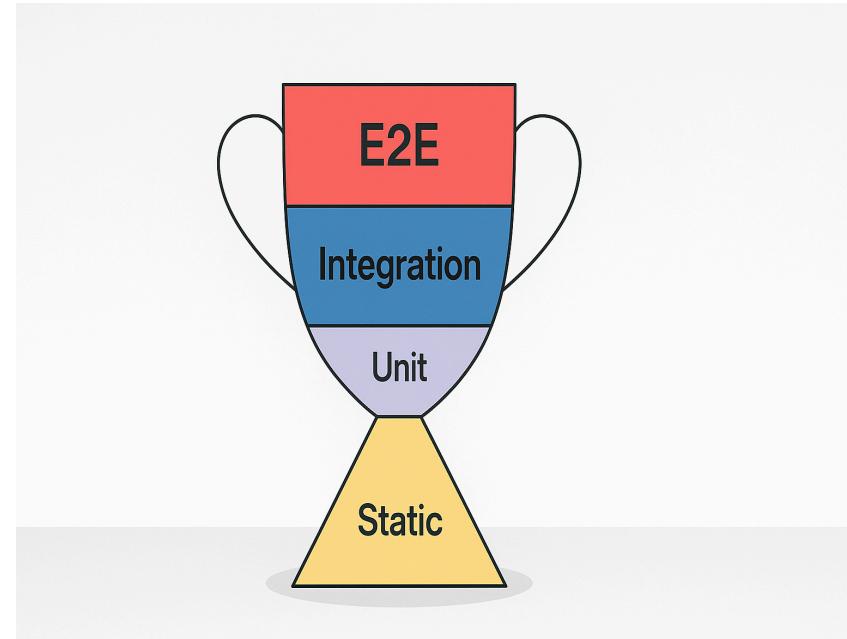
*Die Pyramide ist ein **Modell**, keine starre Regel. Der Kontext (Architektur, Team, Risiken) ist entscheidend.*

Moderne Ansätze: Testing Trophy – Struktur

Die Testing Trophy legt den Fokus auf Integrationstests als wertvollste Testart.

Struktur:

- **Statische Tests** (Basis): Linters, Typ-Checker
- **Unit Tests**: Weniger als in der Pyramide
- **Integration Tests** (Kern): Der größte und wichtigste Teil
- **E2E Tests** (Spitze): Wichtig, aber wenige

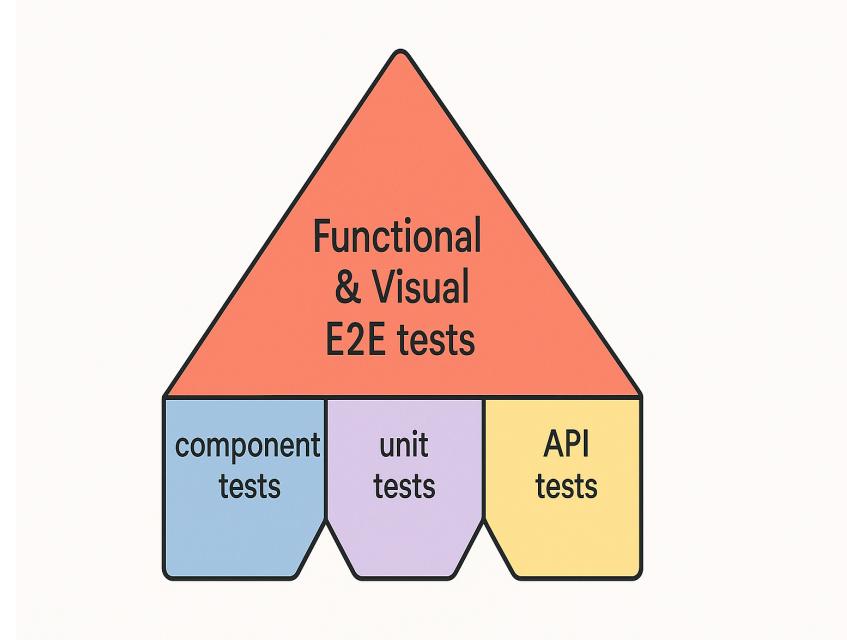


Testing Crab – Struktur & Merkmale

Von Gleb Bahmutov, betont eine **parallele** und
ausgewogene Verteilung:

Struktur (Beispiele):

- Component Tests
- API Tests
- Unit Tests
- Functional E2E Tests
- Visual E2E Tests
- Performance Tests
- Security Tests



Testing Crab – Philosophie

Philosophie:

- **Keine Hierarchie:** Alle Testarten sind wichtig und können parallel laufen.
- **Fokus auf Nutzererlebnis:** Starke Betonung von funktionalen und visuellen E2E-Tests.
- **Flexibilität:** Passt sich gut an komplexe UIs und agile Prozesse an.
- **Parallelisierung:** Schnelleres Feedback durch gleichzeitige Ausführung verschiedener Testsuiten.

Vergleich verschiedener Modelle

Modell	Stärke	Ideal für...
Pyramide	Starke Basis, schnell, gute Isolation	Modulare Systeme, Backend-Services
Trophy	Fokus auf Komponenten-Interaktion	Full-Stack-Apps, Systeme mit vielen Abhängigkeiten
Crab	Parallel, ausgewogen, starker E2E/Visual-Fokus	Komplexe UIs, Agilität, schnelles Feedback

Empfehlung: Verstehe die Prinzipien, passe sie an und **sei bereit zur Anpassung!**

Zusammenfassung – Testarten & Strategien

- Keine Teststrategie ist universell richtig
- Kombination aus Unit, Integration und E2E Tests ist meist sinnvoll
- Kontextabhängige Gewichtung ist entscheidend
- Moderne Tools wie Playwright erlauben produktive E2E-Automatisierung
- **Playwright ermöglicht es, E2E-Tests als starken Pfeiler eurer Qualitätssicherung zu etablieren.**

Playwright Grundlagen

Tag 1: Einstieg in Playwright und E2E-Testing



Daniel Sogl - linktr.ee/daniel_sogl

Was ist Playwright? – Überblick & Geschichte

- **Open-Source** End-to-End Testing Framework von **Microsoft** (seit 2020)
- Entwickelt vom Team, das zuvor an Puppeteer arbeitete
- Basierend auf dem Chrome DevTools Protocol
- **Cross-Browser** Testing für Chromium, Firefox, WebKit
- Unterstützt JavaScript, TypeScript, Python, .NET und Java
- Ziel: Einschränkungen bestehender Frameworks überwinden
- Schnelle Entwicklung, wachsende Community

Typische E2E-Testing Schmerzpunkte

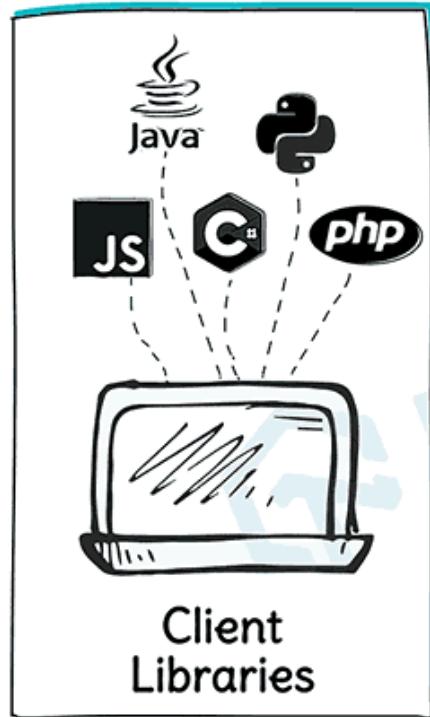
- **Instabile Tests (Flakiness)**: Tests schlagen ohne Code-Änderung fehl.
- **Langsames Feedback**: Lange Testausführungszeiten bremsen die Entwicklung.
- **Cross-Browser-Herausforderungen**: Unterschiedliches Verhalten in Browsern.
- **Komplexe Setups**: Schwierige Konfiguration und Wartung der Testumgebung.
- **Schwieriges Debugging**: Fehler in Tests sind schwer zu finden.
- **Mangelnde Zuverlässigkeit**: Tests finden nicht immer alle Fehler.

Kommen Ihnen diese bekannt vor? Playwright wurde entwickelt, um genau diese Probleme zu lösen!

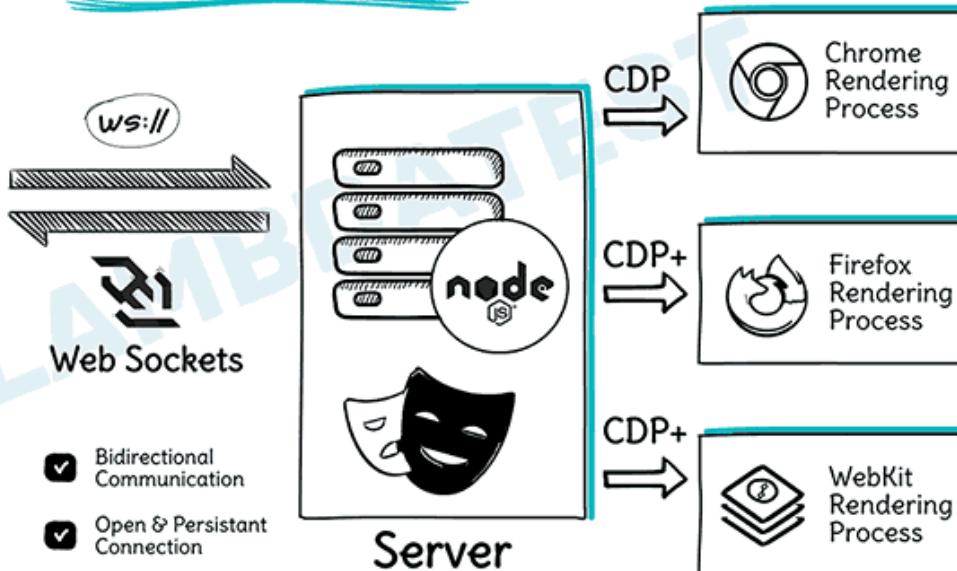
Warum Playwright? Die Lösung für Ihre Test-Herausforderungen

- **Multi-Browser-Support:** Testen Sie konsistent über Chromium, Firefox & WebKit.
- **Auto-Wait & Stabilität:** Intelligente Synchronisation verhindert Flakiness.
- **Geschwindigkeit:** Schnelle Ausführung für zügiges Feedback.
- **Network Interception:** Mocken Sie APIs und kontrollieren Sie das Netzwerkverhalten.
- **Mobile Emulation:** Testen Sie responsive Designs einfach und zuverlässig.
- **Isolation & Zuverlässigkeit:** Browser-Contexts sorgen für unabhängige Tests.
- **Moderne Architekturen:** SPAs, Shadow-DOM, iframes – kein Problem!
- **Traces & Debugging:** Umfangreiche Werkzeuge für schnelle Fehlersuche.

Playwright-Architektur im Überblick



PLAYWRIGHT ARCHITECTURE



Playwright vs. Cypress

- **Cypress:** Sehr beliebt, einfache Einrichtung, tolle DX, aber nur JS/TS, eingeschränkter Browser-Support, keine Multi-Tab/Window-Tests
- **Playwright:** Multi-Browser, Multi-Language, bessere Parallelisierung, umfangreichere API, schnellere Ausführung, Cross-Domain-Tests

Technische Unterschiede: Cypress vs. Playwright

- **Cypress:** Läuft im Browser, jQuery-ähnliche Syntax, automatische Wartezeiten, eingeschränkte Domain-Wechsel, gute Debug-Tools
- **Playwright:** Browser-übergreifende Architektur, moderne Test-APIs, smartes Auto-Waiting, volle Domain-Kontrolle, Trace-Funktionen, Netzwerk-Kontrolle

Installationsvoraussetzungen

- Node.js (v18 oder höher)
- NPM oder Yarn (neueste stabile Version)
- Für verschiedene Browser:
 - Windows: Keine zusätzlichen Anforderungen
 - macOS: Keine zusätzlichen Anforderungen
 - Linux: Eventuell fehlende Browser-Abhängigkeiten

Tipp

Playwright installiert automatisch alle erforderlichen Browser.

Installation von Playwright

```
1 # Mit npm  
2 npm init playwright@latest  
3  
4 # Mit yarn  
5 yarn create playwright  
6  
7 # Mit pnpm  
8 pnpm create playwright
```

Bei der Installation:

- TypeScript oder JavaScript wählen
- Testordner benennen
- GitHub Actions Workflow hinzufügen
- Browser installieren

 Lassen Sie uns das direkt einmal ausprobieren! (Live Demo)

Test Generator – Automatische Testgenerierung

- Was ist der Test Generator?
 - Ein Tool, das automatisch Playwright-Tests generiert, während Sie mit der Anwendung interagieren.
 - Starten Sie den Generator mit `npx playwright codegen <URL>`.
- Vorteile:
 - Spart Zeit bei der Erstellung von Tests.
 - Generiert sofort ausführbaren Code.
 - **Hilft bei der Auswahl stabiler, benutzerorientierter Locators.**
- Demo (noch einmal, weil es so wichtig ist!):
 - Öffnen Sie ein Terminal und führen Sie `npx playwright codegen IHRE_APP_URL` aus.
 - Interagieren Sie mit Ihrer Anwendung, um automatisch Testcode zu generieren.

Demo Projekt auschecken

1. Repository auschecken:

```
git clone https://github.com/danielsogl/playwright-workshop.git  
cd playwright-workshop
```

2. Projekt in VS Code öffnen:

```
code .
```

3. Abhängigkeiten installieren:

```
npm install
```

4. Demo Projekt starten:

```
npm run dev
```

 **Tipp:** Die Anwendung sollte nun unter <http://localhost:3000> erreichbar sein.

Schnellstart: Erster Test mit Codegen

1. Playwright Codegen starten:

```
npx playwright codegen demo.playwright.dev/todomvc
```

2. Interagieren Sie mit der Seite: Fügen Sie Todos hinzu, bearbeiten Sie sie, löschen Sie sie.
3. Beobachten Sie den generierten Code!
4. Kopieren Sie den Code in eine neue Datei (z.B. `e2e/todo.spec.ts`).
5. Test ausführen:

```
npx playwright test e2e/todo.spec.ts
```



Glückwunsch! Ihr erster Playwright Test lief erfolgreich!

Installationsoptionen im Detail

```
1 # Interaktive Installation mit Auswahlmöglichkeiten
2 npm init playwright@latest
3
4 # Installation mit vordefinierten Optionen
5 npm init playwright@latest --quiet
6   --browser=chromium,firefox,webkit
7   --gha
8   --typescript
9   --junit
```

- **-browser:** Zu installierende Browser (chromium, firefox, webkit)
- **-gha:** GitHub Actions Workflow hinzufügen
- **-typescript:** TypeScript-Unterstützung aktivieren
- **-junit:** JUnit-Reporter für CI-Integration

Übung 1 – Projekt-Setup

Ziel: Playwright-Projekt mit automatischem Server-Start einrichten.

Aufgaben:

1. **Dependencies:** `npm install & npx playwright install chromium .`
2. **.env Datei:** `RSS_OFFLINE_MODE=true` für konsistente Testdaten.
3. **playwright.config.ts :** webServer aktivieren für Auto-Start.
4. **Smoke-Test schreiben:** Titel & Navigation prüfen.
5. **Test ausführen:** `npx playwright test` (Server startet automatisch!).

Zeit: 10 Minuten

Projektstruktur

```
1 my-project/
2   └── e2e/          # Testdateien
3     ├── example.spec.ts    # Testspezifikation
4     └── fixtures/        # Testdaten
5   └── playwright.config.ts # Playwright-Konfiguration
6   └── package.json       # Projektabhängigkeiten
7   └── test-results/      # Testergebnisse
8     ├── traces/          # Aufgezeichnete Traces
9     └── screenshots/      # Testscreenshots
```

Wichtigste Dateien für den Anfang:

- `playwright.config.ts` : Das Herzstück Ihrer Testkonfiguration.
- `e2e/` : Hier leben Ihre Testskripte.

Konfigurationsdatei - Minimalbeispiel

```
1 // playwright.config.ts
2 import { defineConfig } from "@playwright/test";
3
4 export default defineConfig({
5   testDir: "./e2e", // Wo sind Ihre Tests?
6   use: {
7     baseURL: "http://localhost:3000", // Basis-URL für Ihre App
8   },
9   webServer: { // Optional: Lokalen Dev-Server starten
10     command: "npm run start",
11     port: 3000,
12     reuseExistingServer: !process.env.CI,
13   },
14   projects: [ // Welche Browser wollen Sie testen?
15     { name: "chromium" },
16     // { name: "firefox" },
17     // { name: "webkit" },
18   ],
19 });

});
```

Tipp: Starten Sie mit einer minimalen Konfiguration und erweitern Sie sie bei Bedarf.

Konfigurationsdatei - Wichtigste Optionen erklärt

- `testDir` : Verzeichnis mit Ihren Testdateien (z.B. `./e2e`).
- `use.baseURL` : Die Basis-URL Ihrer Anwendung (z.B. `http://localhost:3000`). `page.goto("/login")` wird dann zu `http://localhost:3000/login` .
- `projects` : Definiert, welche Browser und Geräte getestet werden sollen (z.B. Chromium, Firefox, mobile Geräte).
- `webServer` : Startet Ihren lokalen Entwicklungsserver automatisch vor den Tests (optional, aber sehr praktisch).

Weitere Optionen (`timeout` , `reporter` , `trace` etc.) werden wir später genauer betrachten.

Konfigurationsdatei - Grundeinstellungen

```
1 // playwright.config.ts
2 import { defineConfig, devices } from "@playwright/test";
3
4 export default defineConfig({
5     // Grundeinstellungen
6     testDir: "./e2e",
7     timeout: 30000,
8     expect: {
9         timeout: 5000,
10    },
11    fullyParallel: true,
12    forbidOnly: !!process.env.CI,
13    retries: process.env.CI ? 2 : 0,
14    workers: process.env.CI ? 1 : undefined,
15 });

```

Konfigurationsdatei - Reporter & Test-Einstellungen

```
1 // playwright.config.ts (Fortsetzung)
2 export default defineConfig({
3   // Reporter-Konfiguration
4   reporter: [["html"], ["junit", { outputFile: "results.xml" }]],
5
6   // Globale Test-Einstellungen
7   use: {
8     baseURL: "http://localhost:3000",
9     trace: "on-first-retry",
10    screenshot: "only-on-failure",
11    timezoneId: "America/New_York",
12    locale: "de-DE",
13  },
14});
```

Konfigurationsdatei - Browser & Webserver

```
1  export default defineConfig({
2    projects: [
3      {
4        name: "chromium",
5        use: { ...devices["Desktop Chrome"] },
6      },
7      {
8        name: "firefox",
9        use: { ...devices["Desktop Firefox"] },
10     },
11     {
12       name: "webkit",
13       use: { ...devices["Desktop Safari"] },
14     },
15   ],
16   // Lokaler Entwicklungsserver
17   webServer: {
18     command: "npm run start",
19     port: 3000,
```

Wichtige Konfigurationsoptionen

- **testDir**: Verzeichnis mit Testdateien
- **timeout**: Maximale Testlaufzeit in Millisekunden
- **expect.timeout**: Timeout für Assertions
- **fullyParallel**: Tests parallel ausführen
- **retries**: Anzahl der Wiederholungsversuche bei Fehlern
- **reporter**: Berichtsformate (HTML, JUnit, etc.)
- **use**: Globale Testeinstellungen
 - **baseURL**: Basis-URL für relative Pfade
 - **trace**: Wann Traces aufgezeichnet werden sollen
 - **screenshot**: Wann Screenshots erstellt werden sollen
- **projects**: Browser-Konfigurationen
- **webServer**: Lokalen Entwicklungsserver starten

Erster Test – Codeübersicht

```
1 import { test, expect } from "@playwright/test";
2
3 test("Startseite hat den richtigen Titel", async ({ page }) => {
4     // Seite öffnen
5     await page.goto("https://example.com");
6
7     // Titel prüfen
8     await expect(page).toHaveTitle(/Example Domain/);
9
10    // Text auf der Seite prüfen
11    const heading = page.locator("h1");
12    await expect(heading).toHaveText("Example Domain");
13});
```

Teststruktur im Detail

```
1 // Import der Playwright Test API
2 import { test, expect } from "@playwright/test";
3
4 // Testgruppe definieren
5 test.describe("Startseiten-Tests", () => {
6     // Setup vor jedem Test
7     test.beforeEach(async ({ page }) => {
8         await page.goto("https://example.com");
9     });
10
11     // Einzelner Test
12     test("hat den richtigen Titel", async ({ page }) => {
13         await expect(page).toHaveTitle(/Example Domain/);
14     });
15 });
```

Erster Test – Ergebnisse

```
1 # Test ausführen  
2 npx playwright test  
3  
4 # Test mit UI-Modus ausführen  
5 npx playwright test --ui  
6  
7 # Test in spezifischem Browser ausführen  
8 npx playwright test --project chromium
```

Test ausführen

- **CLI-Ausführung:** `npx playwright test`
- **UI-Modus:** `npx playwright test --ui`
- **Spezifischer Browser:** `npx playwright test --project chromium`
- **Headed-Modus:** `npx playwright test --headed`
- **Debug-Modus:** `npx playwright test --debug`

Test mit UI-Modus ausführen

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

- Interaktive Oberfläche zum Ausführen und Debuggen von Tests
- Live-Protokoll der Testausführung
- Visuelles Feedback mit Screenshots
- Time-Travel Debugging durch jeden Schritt
- Watch-Modus für automatisches Re-Run bei Änderungen

Test in spezifischem Browser ausführen

```
1  # Nur Chromium
2  npx playwright test --project chromium
3
4  # Nur Firefox
5  npx playwright test --project firefox
6
7  # Nur WebKit
8  npx playwright test --project webkit
9
10 # Mobile Chrome
11 npx playwright test --project "Mobile Chrome"
12
13 # Alle Browser
14 npx playwright test
15
16 # Mehrere spezifische Browser
17 npx playwright test --project chromium --project firefox
```

UI Mode – Interaktives Testing

- **Visueller Test Runner mit Live-Preview**
- **Time-Travel Debugging** - Schritt für Schritt durch Tests
- **Locator Playground** - Elemente finden und testen
- **Watch Mode** - Tests automatisch bei Änderungen ausführen
- **Trace Viewer Integration** - Detaillierte Test-Analyse

UI Mode starten

- **Befehl:** `npx playwright test --ui`
- **Features:**
 - Test-Auswahl per Klick
 - Live-Browser-Vorschau
 - Step-by-Step Ausführung
 - Time-Travel durch Aktionen
 - Locator-Playground
 - Watch-Modus für automatisches Re-Run

Testausführung – Erweiterte Optionen

```
1  # Bestimme Testdatei ausführen
2  npx playwright test tests/example.spec.ts
3
4  # Tests mit bestimmtem Namen ausführen
5  npx playwright test -g "Login"
6
7  # Tests im Watch-Modus ausführen (bei Änderungen neu starten)
8  npx playwright test --ui --watch
9
10 # Headed-Modus (sichtbare Browser)
11 npx playwright test --headed
12
13 # Debugging mit Inspector
14 npx playwright test --debug
15
16 # Anzahl paralleler Worker festlegen
17 npx playwright test --workers=4
18
19 # Tests in Serie ausführen (keine Parallelisierung)
20 npx playwright test --workers=1
```

Bestimmte Testdatei ausführen

```
1 npx playwright test tests/login.spec.ts
```

- **Einzelne Datei:** Nur Tests aus einer Datei
- **Glob-Pattern:** tests/**/*.spec.ts
- **Mehrere Dateien:** Leerzeichen-getrennt angeben
- **Schneller Feedback-Loop** während der Entwicklung

Tests mit bestimmtem Namen ausführen

```
1 npx playwright test -g "Login"
```

- **Grep-Filter:** Nur Tests mit passendem Namen
- **Regex-Support:** `-g "Login.*Success"`
- **Case-insensitive:** `-g "login" -i`
- **Negation:** `-g "^!(?.*Login).*$"` (alle außer Login)

Tests im Watch-Modus ausführen

```
1 npx playwright test --ui --watch
```

- Automatisches Re-Run bei Dateiänderungen
- Nur betroffene Tests werden neu ausgeführt
- Live-Feedback während der Entwicklung
- Perfekt für TDD (Test-Driven Development)

Headed-Modus (sichtbare Browser)

```
1 npx playwright test --headed
```

- Browser wird sichtbar ausgeführt
- Gut für Debugging und Entwicklung
- Langsamer als Headless-Modus
- Kombinierbar mit `--debug` für Pause

Debugging mit Inspector

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

- Pausiert bei jedem Schritt
- Inspector-UI für Schritt-für-Schritt
- Locator-Playground zum Testen
- Console für Live-Debugging
- Netzwerk-Tab für API-Calls

Playwright Debugging

Tests verstehen und Fehler finden



Daniel Sogl - linktr.ee/daniel_sogl

Debugging mit Playwright Inspector

- Visuelle Debugging-Oberfläche
- Step-by-Step Ausführung: Durch Tests navigieren
- DOM-Inspektor: Elemente untersuchen und Locators testen
- Konsole: JavaScript im Browserkontext ausführen
- Breakpoints: Test an bestimmten Stellen anhalten

```
1 # Mit Inspector starten
2 npx playwright test --debug
3
4 # UI-Modus für visuelles Debugging
5 npx playwright test --ui
```

Playwright Inspector im Detail

- **Action Explorer:** Zeigt alle Aktionen im Test
- **Watch Mode:** Automatische Aktualisierung bei Codeänderungen
- **Selector Explorer:** Hilft bei der Auswahl von Locators
- **Network Monitor:** Zeigt Netzwerkanfragen und -antworten
- **Console:** Zeigt Konsolenausgaben und Fehler
- **Source Panel:** Zeigt den Testcode mit Breakpoints

The screenshot shows the Playwright Inspector window. The top bar has icons for Record, Run, Stop, and Refresh, and a Target dropdown set to "example.spec.ts". The main area is divided into two panels: the Source Panel on the left and the Network Monitor on the right.

Source Panel:

```
1 import { test, expect } from '@playwright/test';
2
3 test('has title', async ({ page }) => {
4     await page.goto('https://playwright.dev/');
5
6     // Expect a title "to contain" a substring.
7     await expect(page).toHaveTitle(/Playwright/);
8 });
9
10 test('get started link', async ({ page }) => {
11     await page.goto('https://playwright.dev/');
12     // Click the get started link.
13     await page.getByRole('link', { name: 'Get started' }).click();
14
15     // Expects the URL to contain intro.
16     await expect(page).toHaveURL(/.*intro/);
17 });
```

Network Monitor:

- > browserContext.newPage ✓ — 222ms
- > page.goto(<https://playwright.dev/>) ✓ — 405ms
- > page.getByRole('link', { name: 'Get started' }).click() □
 - waiting for getByRole('link', { name: 'Get started' })
 - locator resolved to visible Get started
 - attempting click action
 - waiting for element to be visible, enabled and stable
 - element is visible, enabled and stable
 - scrolling into view if needed

Debugging-Strategien - Entwicklung

Während der Entwicklung

- `--debug` für interaktives Debugging
- `--ui` für UI-Modus mit Watch-Funktion
- `--headed` für sichtbare Browser
- `page.pause()` für manuelle Breakpoints
- `console.log()` für Variableninspektion

Bei fehlgeschlagenen Tests

- **Trace-Dateien** analysieren
- **Screenshots und Videos** prüfen
- **Netzwerkanfragen** untersuchen
- **Konsolenausgaben** überprüfen
- **Locators** mit Inspector testen

Debugging-Tipps im Code

Debugging-Tipps im Code

```
1 // Debugging-Tipps im Code
2 test("Debugging-Beispiel", async ({ page }) => {
3     // Testausführung pausieren
4     await page.pause();
5
6     // Variablen in der Konsole ausgeben
7     const url = page.url();
8     console.log("Aktuelle URL:", url);
9
10    // DOM-Zustand in der Konsole ausgeben
11    await page.evaluate(() => {
12        console.log("DOM-Struktur:", document.body.innerHTML);
13    });
14
15    // Locators testen
16    const buttonExists = await page
17        .getByRole("button", { name: "Submit" })
18        .isVisible();
19    console.log("Button sichtbar:", buttonExists);
```

Debugging – Einfache Techniken

Einfaches Debugging

```
1 // Testausführung pausieren
2 await page.pause();
3
4 // Screenshot erstellen
5 await page.screenshot({
6   path: "debug.png",
7 });
8
9 // Konsole nutzen
10 console.log("Current URL:", page.url());
```

Mit Trace Viewer

```
1 // In playwright.config.ts
2 use: {
3   trace: 'on-first-retry', // or 'on' or 'retain-on-failure'
4 }
5
6 // Trace für einen Test starten
7 await context.tracing.start({ screenshots: true });
8
9 // ... Test ausführen ...
10
11 // Trace speichern
12 await context.tracing.stop({ path: 'trace.zip' });
```

Trace Viewer – Analyse fehlgeschlagener Tests

The screenshot shows the Playwright Trace Viewer interface, which displays a timeline of actions taken during a test run. The timeline is at the top, showing a sequence of events from 0 to 5.5 seconds. Below the timeline is a detailed log of actions, their metadata, and a screenshot of the application state at the time of the failure.

Actions & Metadata

Action	Before	After
> Before Hooks	389ms	
locator.check <code>getByTestId('t...')</code>	20ms	
locator.click <code>getByRole('but...')</code>	66ms	
<code>expect.toHaveCount <code>getByTe...</code></code>	5.0s	
> After Hooks	0ms	

Screenshot

A screenshot of a web browser window titled "todos". The page lists three items: "buy some cheese", "book a doctors appointment", and "learn how to use the API". A tooltip above the first item says "This is a todo item for testing - it has no id". The "Completed" button for the first item is highlighted in blue. At the bottom of the page, there are tabs for "All", "Active", and "Completed".

Logs

- Locator: 480ms
- Call: 480ms `expect.toHaveCount getByTestId('todo-item')`
- Errors: 1
- Console: 1
- Network: 6
- Source: 1
- Attachments: 1

Failure Details

Timed out 5000ms waiting for `expect(received).toHaveCount(expected)` // deep equality

Expected: 3
Received: 2
Call log:

- `expect.toHaveCount` with timeout 5000ms
- waiting for `getByTestId('todo-item')`
- locator resolved to 2 elements
- unexpected value "2"

Trace-Funktionen im Detail

Was wird aufgezeichnet

- Screenshots (vor/nach jeder Aktion)
- DOM-Snapshots
- Netzwerkanfragen
- Konsolenausgaben
- Testschritte mit Zeitstempeln
- Browserkontext-Informationen

Konfigurationsoptionen

```
1 // In playwright.config.ts
2 use: {
3   trace: 'on-first-retry', // Standard
4   // Alternativen:
5   // 'on' - immer aufzeichnen
6   // 'off' - nie aufzeichnen
7   // 'retain-on-failure' - nur bei Fehlern behalten
8 }
```

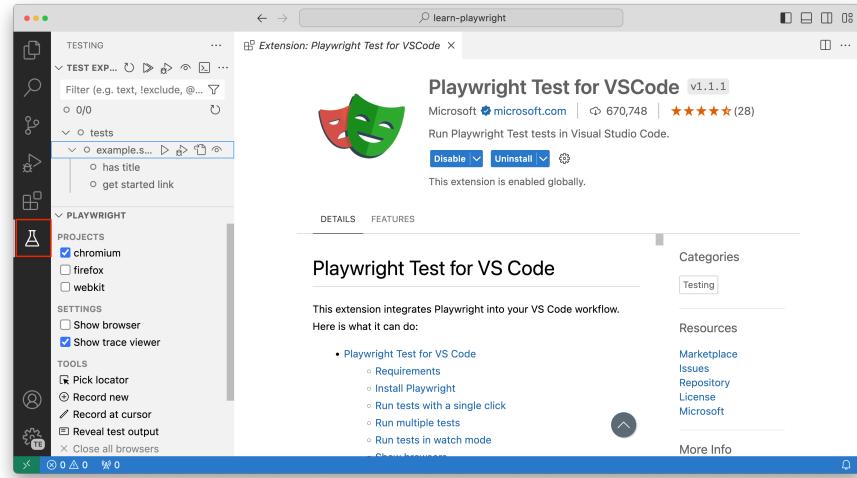
Trace Viewer – Aktionen gruppieren

```
1 // Aktionen im Trace Viewer visuell gruppieren
2 test('komplexer User-Flow', async ({ page }) => {
3     // Login-Gruppe
4     await page.context().tracing.group('Benutzer-Login');
5     await page.goto('/login');
6     await page.getLabel('Email').fill('user@test.com');
7     await page.getLabel('Passwort').fill('password');
8     await page.getRole('button', { name: 'Login' }).click();
9     await page.context().tracing.groupEnd();
10
11    // Daten-Abruf-Gruppe
12    await page.context().tracing.group('Daten laden');
13    await page.goto('/dashboard');
14    await page.getRole('button', { name: 'Daten aktualisieren' }).click();
15    await page.waitForResponse('**/api/data');
16    await page.context().tracing.groupEnd();
17});
```

Vorteile: Komplexe Testflows werden im Trace Viewer übersichtlich in logische Abschnitte unterteilt.

Playwright in der IDE: Visual Studio Code

- Offizielle Extension: **Playwright Test for VS Code** (ID: ms-playwright.playwright)
- **Test-Explorer:** Tests ausführen, debuggen und Ergebnisse direkt in der IDE sehen
- **Debugging:** Breakpoints setzen und interaktiv debuggen
- **Locator-Tools:**
 - Locators testen und inspizieren
 - Locators generieren ("Pick Locator")
- **Tests aufzeichnen** (Record new tests)
- **Trace-Viewer** und Screenshots direkt öffnen
- **Installation:** Im Extensions Marketplace nach "Playwright Test" suchen (oder ID: ms-



Playwright in der IDE: JetBrains (WebStorm, IntelliJ, ...)

- Native Playwright-Unterstützung ab WebStorm 2022.3
- Test-Explorer: Übersicht und Ausführung aller Tests
- Debugging mit Breakpoints und Step-by-Step
- Trace-Viewer Integration
- Automatische Erkennung von Playwright-Tests nach Installation als npm-Dependency
- Keine zusätzliche Extension notwendig

Console Events und Error Handling

- **Console-Nachrichten abfangen:**
 - Alle `console.log`, `console.error`, etc. aus der Webseite
 - Hilfreich zum Debuggen von JavaScript-Fehlern
 - Validierung von erwarteten/unerwarteten Konsolenausgaben
- **Page-Errors überwachen:**
 - Nicht abgefangene JavaScript-Exceptions
 - Netzwerkfehler und fehlgeschlagene Ressourcen
 - Kritische Fehler, die die Anwendung beeinträchtigen
- **Anwendungsfälle:**
 - CI/CD: Tests bei JavaScript-Fehlern fehlschlagen lassen
 - Performance-Monitoring: Warnungen überwachen
 - Fehleranalyse: Debugging-Informationen sammeln

Console Events - Beispiele

```
1 // Console-Nachrichten abfangen
2 test('Console-Ausgaben überwachen', async ({ page }) => {
3     // Handler für Console-Nachrichten
4     page.on('console', msg => {
5         console.log(`#${msg.type()}: ${msg.text()}`);
6
7         // Bei Errors Test fehlschlagen lassen
8         if (msg.type() === 'error') {
9             throw new Error(`Console error: ${msg.text()}`);
10        }
11    });
12
13     // Handler für unbehandelte Exceptions
14     page.on('pageerror', error => {
15         console.error('Page error:', error.message);
16         throw error;
17    });
18
19     await page.goto('/');
20
21     // Spezifische Console-Nachrichten validieren
22     const consoleMessages = [];
```

Erweiterte Console-Debugging-Techniken

```
1  test('Detaillierte Console-Analyse', async ({ page }) => {
2      const consoleLogs = [];
3
4      page.on('console', async msg => {
5          // Nachrichtentyp und Argumente extrahieren
6          const values = await Promise.all(
7              msg.args().map(arg => arg.jsonValue())
8          );
9
10         consoleLogs.push({
11             type: msg.type(),
12             text: msg.text(),
13             location: msg.location(),
14             values: values
15         });
16     });
17
18     await page.goto('/dashboard');
19
20     // Console-Logs nach Test analysieren
21     const errors = consoleLogs.filter(log => log.type === 'error');
22     const warnings = consoleLogs.filter(log => log.type === 'warning');
```

Page Crashes und Request Failures

```
1 // Seiten-Crashes behandeln
2 test('Page Crash Detection', async ({ page }) => {
3   page.on('crash', () => {
4     console.error('Page crashed!');
5     throw new Error('Page crashed during test');
6   });
7
8   await page.goto('/unstable-page');
9   // ... weitere Test-Schritte
10 });
11
12 // Fehlgeschlagene Requests überwachen
13 test('Network Error Monitoring', async ({ page }) => {
14   const failedRequests = [];
15
16   page.on('requestfailed', request => {
17     failedRequests.push({
18       url: request.url(),
19       failure: request.failure()
```

Locators

Was sind Locators?

- Locators sind Ausdrücke, mit denen Playwright Elemente auf einer Webseite findet und anspricht.
- Sie ermöglichen es, gezielt mit Buttons, Feldern, Texten usw. zu interagieren.
- Playwright unterstützt verschiedene Locator-Typen: Text, Rolle, Label, Test-ID, CSS, XPath u.v.m.
- Gute Locators machen Tests stabil, lesbar und wartbar.
- Sie sind die Grundlage für alle Aktionen und Assertions im Test.
- Playwright priorisiert benutzerorientierte Locators (z.B. sichtbarer Text, Rollen) für robuste Tests.

User-Facing Locators

User-Facing Locators sind Locators, die sich an den sichtbaren und zugänglichen Eigenschaften einer Benutzeroberfläche orientieren – also an dem, was echte Nutzer:innen sehen und womit sie interagieren. Dazu zählen beispielsweise sichtbarer Text, Rollen (wie `button`, `link`), Labels von Formularfeldern, Platzhaltertexte, Alt-Texte von Bildern oder Titel-Attribute.

Warum User-Facing Locators verwenden?

- **Robustheit:** Sie sind weniger anfällig für Änderungen im Code, da sie sich auf stabile, für Nutzer:innen sichtbare Eigenschaften stützen – nicht auf interne Implementierungsdetails wie CSS-Klassen oder verschachtelte Strukturen.
- **Lesbarkeit:** Tests mit User-Facing Locators sind verständlicher, da sie die tatsächliche Nutzerinteraktion widerspiegeln.
- **Barrierefreiheit:** Sie fördern gute Accessibility, da sie auf zugängliche Attribute wie Rollen und Labels setzen.
- **Wartbarkeit:** Änderungen am Styling oder an der Struktur der Seite führen seltener zu fehlschlagenden Tests, solange die Benutzeroberfläche gleich bleibt.

Beispiele für User-Facing Locators in Playwright

```
1 // Nach sichtbarem Text
2 await page.getByText("Anmelden").click();
3
4 // Nach Rolle (z.B. Button)
5 await page.getRole("button", { name: "Absenden" }).click();
6
7 // Nach Label (z.B. für Inputs)
8 await page.getLabel("Benutzername").fill("user123");
9
10 // Nach Platzhaltertext
11 await page.getPlaceholder("Passwort eingeben").fill("pass123");
12
13 // Nach Alt-Text (z.B. für Bilder)
14 await page.getAltText("Firmenlogo").isVisible();
```

Fazit: User-Facing Locators machen Tests stabiler, verständlicher und nachhaltiger. Sie sollten immer bevorzugt werden – Test-IDs oder CSS/XPath-Locators sind nur dann sinnvoll, wenn keine benutzerorientierten Locators möglich sind.

Alternative Locators in Playwright

Neben den benutzerorientierten Locators (wie `getByRole`, `getByText`, `getByLabel` usw.) bietet Playwright auch **alternative Locators** für spezielle Anwendungsfälle oder wenn User-Facing-Locators nicht ausreichen.

1. CSS-Locators

- Verwende Standard-CSS-Locators, um Elemente gezielt anzusprechen.
- Beispiel:

```
await page.locator('.my-class > input[type="email"]').fill('test@example.com');
```

2. XPath-Locators

- Beispiel:

```
await page.locator('//button[@type="submit"]').click();
```

3. Test-IDs

- Mit expliziten Attributen wie `data-testid` lassen sich stabile Locators definieren.
- Beispiel:

```
await page.getByTestId('checkout-payment-submit').click();
```

4. Kombination und Filter

- Playwright erlaubt das Kombinieren und Filtern von Locators:

```
await page.locator('button').filter({ hasText: 'Speichern' }).click();
```

Hinweis:

Alternative Locators sind nützlich, wenn keine benutzerorientierten Locators möglich sind. Sie sollten jedoch mit Bedacht eingesetzt werden, da sie oft weniger robust gegenüber UI-Änderungen sind.

Locators-Strategien und Best Practices

Locators: Empfohlene Reihenfolge

1. User-facing Attribute (Was der Benutzer sieht/interagiert):

- `page.getByRole()` (Zugänglichkeit: Buttons, Links, Headings etc.)
- `page.getText()` (Sichtbarer Text)
- `page.getByLabel()` (Verknüpfte Formular-Labels)
- `page.getPlaceholder()` (Platzhaltertext in Inputs)
- `page.getAltText()` (Alt-Text von Bildern)
- `page.getTitle()` (Titel-Attribut)

➡ Warum diese? Sie sind robust, lesbar & fördern barrierefreie Anwendungen!

2. Test-spezifische Attribute (Wenn nichts anderes geht):

- `page.getTestId()` (Explizite `data-testid` o.ä. für Tests)

💡 Wie finde ich diese? Chrome DevTools (Accessibility Tab) & Playwright Inspector/Codegen sind deine Werkzeuge! (Mehr dazu gleich!)

Locators: Zu vermeiden (wenn möglich)

- CSS-Locators, die auf Implementierungsdetails basieren (Styling-Klassen, komplexe Strukturen)
 - Beispiel: `.button-styles.primary > span`
- XPath-Locators (oft komplex und weniger lesbar)
- Klassen/IDs, die sich ändern könnten (z.B. durch CSS-Module generiert)
- Indizes ohne klare semantische Bedeutung (z.B. `li:nth-child(3)`)

Tipp

Verwende den Codegen (`npx playwright codegen`) oder den Inspector (`--debug`), um gute, resiliente Locators zu finden! Playwright priorisiert automatisch benutzerorientierte Locators.

Test-IDs – Best Practices

- **Konsistente Benennung:**
 - Verwende ein klares Namensschema
 - z.B. feature-component-action
 - Beispiel: checkout-payment-submit
- **Strukturierte Hierarchie:**
 - Eltern-Kind-Beziehungen abbilden
 - z.B. user-menu und user-menu-profile
- **Vermeidung von Duplikaten:**
 - Eindeutige IDs im gesamten Projekt
 - Automatisierte Prüfung auf Duplikate
- **Dokumentation:**
 - Test-IDs in Komponenten-Dokumentation aufnehmen

Wann Test-IDs verwenden?

Test-IDs: Vorteile

- **Stabile Locators:** Unabhängig von Textänderungen (Übersetzungen) oder UI-Strukturänderungen.
- **Explizit für Tests:** Macht klar, dass ein Element für die Automatisierung wichtig ist.
- **Entkopplung:** Trennt Testlogik von der genauen UI-Implementierung.
- **Klar dokumentierte Testbarkeit:** Signalisiert, welche Teile der UI testbar sein sollen.

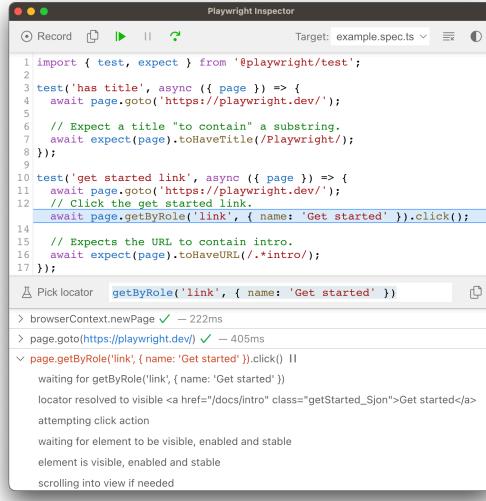
Test-IDs: Nachteile & Empfehlung

- **Zusätzlicher Markup-Overhead:** Müssen im Code hinzugefügt und gepflegt werden.
- **Nicht benutzerorientiert:** Testen nicht, wie ein echter Benutzer die Seite wahrnimmt (z.B. über sichtbaren Text oder Rollen).
- **Kann zu "Test-only"-Denken führen:** Elemente könnten nur für Tests hinzugefügt werden, ohne echten Nutzen.
- **Nicht für Barrierefreiheit relevant:** Im Gegensatz zu Rollen-Locators.

Empfehlung

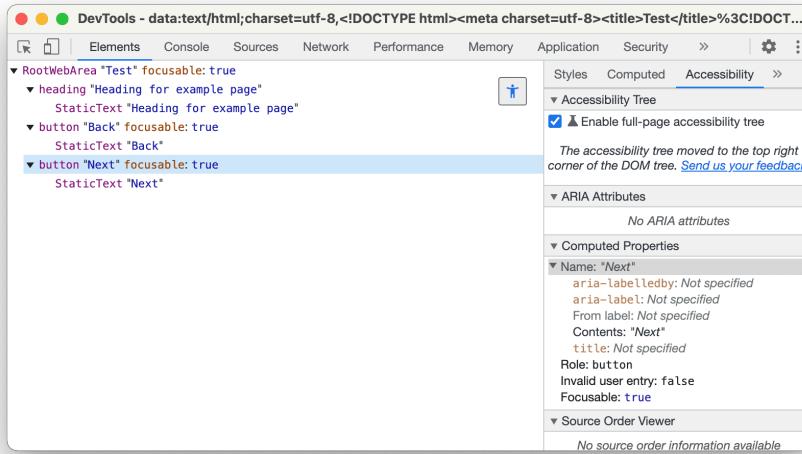
Verwende Test-IDs als ****Fallback****, wenn benutzerorientierte Locators (Rolle, Text, Label etc.) nicht eindeutig, stabil oder praktikabel sind. Priorisiere immer benutzerorientierte Locators!

Locators finden mit Playwright Inspector



- **Starten:** `npx playwright test --ui` oder `npx playwright test --debug`.
- **Inspector:** Interaktives Auswählen von Elementen im Browser.
- **Locator-Vorschläge:** Playwright schlägt automatisch gute Locators vor.
- **Testen & Verfeinern:** Locators direkt im Inspector ausprobieren.

Locators finden mit Chrome DevTools



Die Chrome DevTools bieten im "Accessibility"-Tab die Möglichkeit, den sogenannten **Accessibility Tree** einer Webseite einzusehen. Dieser Baum zeigt, wie Screenreader und andere Hilfsmittel die Seite wahrnehmen. Jedes Element im Accessibility Tree besitzt Eigenschaften wie **Rolle** (z.B. `button`, `link`), **Name** (sichtbarer Text), **Label**, **Beschreibung** und weitere zugängliche Attribute.

Locators finden mit Chrome DevTools

So nutzt du den Accessibility Tree:

1. Öffne die Chrome DevTools (F12 oder Rechtsklick → "Untersuchen").
2. Wähle das gewünschte Element im "Elements"-Tab aus.
3. Wechsle zum Tab "**Accessibility**" (ggf. unter ">>" versteckt).
4. Sieh dir Rolle, Name, Label und weitere Eigenschaften an.

Tipp:

Wenn ein Element im Accessibility Tree nicht wie erwartet erscheint, kann das auf fehlende oder fehlerhafte ARIA-Attribute oder Labels hindeuten – ein wichtiger Hinweis für die Verbesserung der Barrierefreiheit und für stabile Playwright-Locators!

Locators Filter

- **Filter nach Text:** `.filter({ hasText: 'Suchtext' })`
- **Filter nach Kindelementen:** `.filter({ has: page.getByRole('button') })`
- **Filter nach Sichtbarkeit:** `.filter({ visible: true })`
- **Filter nach Position:** `.first()` , `.last()` , `.nth(index)`
- **Kombinierte Filter:** Mehrere Filter können verkettet werden
- **Strikte Filterung:** Playwright wirft einen Fehler, wenn mehrere Elemente gefunden werden

Hinweis

Vermeide `.first()` , `.last()` und `.nth()` wenn möglich. Besser: Eindeutige Locators verwenden!

Locators Filter - Beispiele

```
1 // Nach Text filtern
2 await page.locator('li').filter({ hasText: 'News' }).click();
3
4 // Nach Kindelement filtern
5 await page.locator('article').filter({ has: page.getByRole('button', { name: 'Mehr' }) }).first().click();
6
7 // Nur sichtbare Elemente auswählen
8 const visibleItems = page.locator('.menu-item').filter({ hasText: 'Aktiv', visible: true });
9 await expect(visibleItems).toHaveLength(1);
10
11 // Erstes, letztes oder bestimmtes Element auswählen
12 await page.locator('.list-item').first().click();
13 await page.locator('.list-item').last().click();
14 await page.locator('.list-item').nth(2).click(); // drittes Element (0-basiert)
15
16 // Kombinierte Filter: Nach Text und Kindelement
17 await page.locator('section').filter({ hasText: 'Wichtig', has: page.getByRole('button', { name: 'Details' }) }).click();
18
19 // Strikte Filterung: Fehler, wenn mehrere Elemente gefunden werden
```

Übung 2 – Locators kennenlernen

Ziel: Verschiedene Locator-Strategien praktisch anwenden.

1. **Neuer Test:** `e2e/locators-practice.spec.ts` erstellen
2. **Navigation:** Zur Feed App (`http://localhost:3000`)
3. **Elemente finden mit verschiedenen Locators:**
 - "Public News" Link → `getByRole()`
 - "News Feed" Überschrift → `getByText()`
 - Suchfeld → `getByPlaceholder()`
 - Theme-Toggle → `getByLabel()`
4. **Bestätigung:** `console.log()` für gefundene Elemente
5. **Experiment:** Locator Playground im UI Mode (`-ui`)

Hinweis: Noch KEINE Assertions verwenden - nur Elemente finden und loggen!

Zeit: 15 Minuten

Playwright Actions

Was sind Playwright Actions?

- Aktionen, mit denen Playwright Benutzerinteraktionen imitiert
- Beispiele: Klicken, Tippen, Ausfüllen von Formularen, Auswählen, Drag & Drop
- Simulieren echtes Nutzerverhalten im Browser
- Können mit Maus, Tastatur oder Touch ausgeführt werden
- Grundlage für alle Interaktionen und Testszenarien
- Unterstützen Auto-Waiting für stabile Tests
- Ermöglichen komplexe Abläufe wie Datei-Upserts oder Mehrfachauswahl

Aktionen in Playwright

```
1 // Grundlegende Aktionen
2 await page.goto("https://example.com");
3 await page.getByRole("button").click();
4 await page.getLabel("Passwort").fill("12345");
5 await page.getByRole("checkbox").check();
6
7 // Erweiterte Aktionen
8 await page.keyboard.press("Enter");
9 await page.mouse.move(100, 200);
10 await page.getText("Über uns").hover();
11 await page.getLabel("Datei").setInputFiles("upload.jpg");
```

Erweiterte Interaktionen

```
1 // Drag & Drop
2 await page.getByTestId("draggable").dragTo(page.getByTestId("target"));
3
4 // Mehrere Tasten gleichzeitig drücken
5 await page.keyboard.press("Control+A");
6 await page.keyboard.press("Control+C");
7
8 // Mausaktionen
9 await page.mouse.down();
10 await page.mouse.move(100, 200);
11 await page.mouse.up();
12
13 // Mehrfachklick
14 await page.getText("Doppelklick").dblclick();
15
16 // Rechtsklick
17 await page.getText("Kontextmenü").click({ button: "right" });
18
19 // Mit Modifizieren klicken
```

Formular-Interaktionen

```
1 // Formular-Interaktionen
2 await page.getByLabel("Benutzername").fill("user123");
3 await page.getByLabel("Passwort").fill("password123");
4 await page.getByLabel("E-Mail").fill("user@example.com");
5
6 // Checkbox und Radio-Buttons
7 await page.getByLabel("Nutzungsbedingungen akzeptieren").check();
8 await page.getByLabel("Newsletter abonnieren").uncheck();
9 await page.getByLabel("Option 2").check(); // Radio-Button
10
11 // Dropdown-Menüs
12 await page.getByLabel("Land").selectOption("Deutschland");
13 await page.getByLabel("Sprache").selectOption({ label: "Deutsch" });
14 await page.getByLabel("Kategorie").selectOption({ value: "tech" });
15
16 // Datei-Upload
17 await page.getByLabel("Profilbild").setInputFiles("path/to/image.jpg");
18 await page.getByLabel("Dokumente").setInputFiles(["doc1.pdf", "doc2.pdf"]);
19
20 // Formular absenden
21 await page.getByRole("button", { name: "Absenden" }).click();
```

Erweiterte File Upload-Beispiele

```
1 // Datei-Upload mit verschiedenen Methoden
2 test('Verschiedene Upload-Szenarien', async ({ page }) => {
3     // 1. Einfacher Upload über Input-Element
4     await page.getLabel('Avatar').setInputFiles('path/to/avatar.png');
5
6     // 2. Mehrere Dateien gleichzeitig
7     await page.getLabel('Dokumente').setInputFiles([
8         'path/to/doc1.pdf',
9         'path/to/doc2.pdf',
10        'path/to/doc3.xlsx'
11    ]);
12
13     // 3. Datei per Drag & Drop (mitFileChooser)
14     const [fileChooser] = await Promise.all([
15         page.waitForEvent('filechooser'),
16         page.getText('Hier klicken oder Datei ablegen').click(),
17     ]);
18     await fileChooser.setFiles('path/to/dragdrop.zip');
19
20     // 4. Buffer/Memory Upload (ohne physische Datei)
21     await page.getLabel('CSV Import').setInputFiles({
22         name: 'data.csv',
```

File Upload - Drag & Drop Simulation

```
1  test('Drag & Drop File Upload', async ({ page }) => {
2      await page.goto('/upload');
3
4      // Datei vorbereiten
5      const buffer = Buffer.from('Test file content');
6      const dataTransfer = await page.evaluateHandle(() => new DataTransfer());
7
8      // Datei zum DataTransfer hinzufügen
9      await page.evaluateHandle(
10        ([dt, buffer]) => {
11            const file = new File([buffer], 'test.txt', { type: 'text/plain' });
12            dt.items.add(file);
13        },
14        [dataTransfer, buffer]
15    );
16
17      // Drag & Drop Events simulieren
18      const dropzone = page.locator('.dropzone');
19
20      await dropzone.dispatchEvent('dragenter', { dataTransfer });
21      await dropzone.dispatchEvent('dragover', { dataTransfer });
22      await dropzone.dispatchEvent('drop', { dataTransfer });
```

File Upload - Validierung und Fehlerbehandlung

```
1  test('File Upload mit Validierung', async ({ page }) => {
2    await page.goto('/profile');
3
4    // Dateigröße prüfen (Client-seitig simuliert)
5    const largeFile = {
6      name: 'large.zip',
7      mimeType: 'application/zip',
8      buffer: Buffer.alloc(10 * 1024 * 1024) // 10 MB
9    };
10
11   await page.getLabel('Avatar').setInputFiles(largeFile);
12   await expect(page.getText('File too large')).toBeVisible();
13
14   // Falscher Dateityp
15   await page.getLabel('Avatar').setInputFiles('document.pdf');
16   await expect(page.getText('Only images allowed')).toBeVisible();
17
18   // Erfolgreicher Upload mit Vorschau
19   await page.getLabel('Avatar').setInputFiles('avatar.jpg');
20
21   // Warte auf Vorschau
22   const preview = page.locator('img.preview');
```

Übung 3 – Erste Interaktionen

Ziel: Grundlegende Benutzer-Interaktionen ohne Assertions üben.

Aufgaben:

1. Klick-Interaktionen:

- Klicke auf den "Public News" Link
- Klicke auf den Theme-Toggle Button
- Beobachte die visuellen Änderungen im Browser

2. Tastatur-Eingaben:

- Tippe "Playwright" in die Suchleiste
- Drücke Enter
- Beobachte wie sich die Artikel-Liste ändert

3. Formular ausfüllen (optional):

- Navigiere zum Login-Bereich
- Fülle Username und Password Felder aus

Nachbesprechung: Interaktionen

Diskussion der Übungsergebnisse

Assertions

Assertions in Playwright – Was sind Assertions?

- Eine Assertion prüft, ob ein erwarteter Zustand im Test erfüllt ist.
- Sie ist die Grundlage für verlässliche und aussagekräftige Tests.
- Beispiel: Erwartung, dass ein Button sichtbar ist oder ein Text angezeigt wird.
- Ein Test ohne Assertions prüft nichts und ist wertlos!
- Playwright bietet mächtige und flexible Assertions.

Waiting & Timeouts

Assertions in Playwright – Auto-Retrying Assertions

- Auto-Retrying Assertions warten automatisch auf den gewünschten Zustand
- Typische Beispiele:
 - `toBeVisible`
 - `toHaveText`
 - `toHaveURL`
 - `toBeChecked`
- Reduziert flaky Tests, da Playwright mehrfach prüft, bis das Ergebnis stimmt oder ein Timeout erreicht ist
- Ideal für UI-Elemente, die asynchron erscheinen oder sich ändern

Assertions in Playwright – Auto-Retrying Assertions: Beispiele

```
1 // Sichtbarkeit prüfen
2 await expect(page.getByRole('button', { name: 'Login' })).toBeVisible();
3
4 // Textinhalt prüfen
5 await expect(page.getByTestId('headline')).toHaveText('Willkommen!');
6
7 // URL prüfen
8 await expect(page).toHaveURL(/.*dashboard/);
9
10 // Checkbox angehakt?
11 await expect(page.getLabel('AGB akzeptieren')).toBeChecked();
12
13 // Attribut prüfen
14 await expect(page.getByRole('img')).toHaveAttribute('alt', 'Logo');
15
16 // Anzahl der Elemente prüfen
17 const items = page.locator('.todo-item');
18 await expect(items).toHaveLength(3);
```

Assertions in Playwright – Non-Retrying Assertions

- Non-Retrying (generische) Assertions prüfen synchron einen Wert, ohne zu warten
- Typische Beispiele:
 - toBe
 - toEqual
 - toContain
 - toBeTruthy
- Typisch für Variablen, API-Responses, interne Logik

Assertions in Playwright – Non-Retrying Assertions: Beispiele

```
1 // Statuscode einer API-Response
2 expect(response.status()).toBe(200);
3
4 // Array enthält Wert
5 expect(items).toContain('News');
6
7 // Objektvergleich
8 expect(user).toEqual({ id: 1, name: 'Max' });
9
10 // Wahrheitswert
11 expect(isLoggedIn).toBeTruthy();
```

Assertions in Playwright – Soft Assertions

- Mit `expect.soft(...)` können mehrere Fehler gesammelt werden, ohne den Test sofort abzubrechen
- Am Ende des Tests werden alle Fehler gemeinsam gemeldet
- Nützlich, um mehrere Aspekte in einem Test zu prüfen

Assertions in Playwright – Soft Assertions: Beispiele

```
1 // Mehrere UI-Elemente prüfen, Fehler werden gesammelt
2 expect.soft(page.getByText('A')).toBeVisible();
3 expect.soft(page.getByText('B')).toBeVisible();
4 expect.soft(page.getByText('C')).toBeVisible();
5
6 // Auch kombinierbar mit generischen Assertions
7 expect.soft(apiResponse.status).toBe(200);
```

Assertions in Playwright – Best Practices

- Nutze auto-retry Assertions für UI-Checks
- Schreibe präzise, beschreibende Fehlermeldungen (zweites Argument von `expect`)
- Assertions gehören in die Tests, nicht ins Page Object
- Vermeide unnötige `waitFor`-Aufrufe – Playwright wartet automatisch
- Nutze `.not` für Negationen: `await expect(locator).not.toBeVisible();`
- Halte Assertions so spezifisch wie möglich, um Fehlerursachen schnell zu erkennen

Auto-Wait und Synchronisation

- Playwright **wartet automatisch** auf Elemente
- Keine expliziten Waits notwendig
- Wartet auf verschiedene Bedingungen:
 - Element ist sichtbar
 - Element ist aktiviert
 - Element ist stabil (keine Animation)
 - Netzwerkanfragen sind abgeschlossen

```
1 // Kein explizites Warten notwendig
2 await page.getByRole("button", { name: "Speichern" }).click();
3 // Playwright wartet, bis der Button klickbar ist
```

Übung 4 – Erster Test mit Assertions

Ziel: Locators, Interaktionen und Assertions kombinieren.

Aufgaben:

1. Test erstellen (`e2e/first-test.spec.ts`).
2. Navigation mit `getByRole('link', { name: /public news/i })` testen.
3. Auto-Waiting nutzen: Playwright wartet automatisch!
4. Mit Listen arbeiten: News-Artikel zählen.
5. Debug-Tools kennenlernen (`--debug` , `--ui`).

Zeit: 20 Minuten

Nachbesprechung

Übung 5 – News Feed Suche testen

Ziel: Suchfunktion mit Test-Organisation und dynamischen Inhalten testen.

Aufgaben:

1. Test-Suite mit `describe` und `beforeEach` erstellen.
2. Initiale Anzeige: 65 Items aus `feed.json` prüfen.
3. Suche testen: "Revelo" (1 Ergebnis), "XYZ123" (0 Ergebnisse).
4. Suche zurücksetzen und erneut prüfen.
5. Trace bei Fehler analysieren (`trace: 'retain-on-failure'`).

Zeit: 25 Minuten

Nachbesprechung

Test-Annotierungen in Playwright

Playwright bietet die Möglichkeit, Tests mit **Annotierungen** zu versehen, um sie gezielt zu steuern oder zu konfigurieren. Dies ist besonders nützlich, um Tests zu überspringen, nur unter bestimmten Bedingungen auszuführen oder zusätzliche Metadaten hinzuzufügen.

Arten von Annotierungen

- `test.step` : Unterteilt einen Test.
- `test.skip` : Überspringt einen Test.
- `test.only` : Führt nur diesen Test aus.
- `test.fixme` : Markiert einen Test als fehlerhaft, der später behoben werden soll.
- `test.slow` : Markiert einen Test als langsam, um ihm mehr Zeit zu geben.
- `test.describe.parallel` : Führt Tests innerhalb einer Gruppe parallel aus.

Beispiele für Annotierungen

```
1 // Test überspringen
2 import { test, expect } from '@playwright/test';
3
4 test('dieser Test wird übersprungen', async ({ page }) => {
5   test.skip(true, 'Feature noch nicht implementiert');
6   await page.goto('https://example.com');
7 });
8
9 // Test nur in Chromium ausführen
10 test('nur in Chromium', async ({ browserName }) => {
11   test.skip(browserName !== 'chromium', 'Nur für Chromium relevant');
12 });
13
14 // Langsamer Test
15 test('langsamer Test', async ({ page }) => {
16   test.slow();
17   await page.goto('https://example.com');
18 });
```

Beispiel: test.step() für bessere Testorganisation

```
1 import { test, expect } from '@playwright/test';
2
3 test('Checkout Flow', async ({ page }) => {
4     await test.step('Produkt hinzufügen', async () => {
5         await page.goto('/products');
6         await page.getByRole('button', { name: 'In den Warenkorb' }).click();
7     });
8
9     await test.step('Zur Kasse gehen', async () => {
10        await page.getByRole('link', { name: 'Warenkorb' }).click();
11        await page.getByRole('button', { name: 'Zur Kasse' }).click();
12    });
13
14    await test.step('Zahlung abschließen', async () => {
15        await page.getLabel('Kartennummer').fill('4242424242424242');
16        await page.getByRole('button', { name: 'Bezahlen' }).click();
17    });
18});
```

test.step() - Vorteile

- **Bessere Lesbarkeit:** Tests sind in logische Schritte unterteilt
- **Report Integration:** Schritte werden im HTML Report und Trace Viewer angezeigt
- **Einfacheres Debugging:** Sofort sichtbar, wo ein Test fehlschlägt
- **Verschachtelung möglich:** Steps können weitere Steps enthalten
- **Dokumentation:** Tests lesen sich wie eine Anleitung

Tipp

Nutze `test.step()` für längere Tests mit mehreren Aktionen. Jeder Schritt wird im Trace Viewer einzeln dargestellt.

Best Practices für Annotierungen

- Verwende Annotierungen sparsam, um Tests nicht unnötig zu fragmentieren.
- Dokumentiere den Grund für jede Annotierung, um die Nachvollziehbarkeit zu gewährleisten.
- Nutze `test.only` nur lokal, um versehentliches Einchecken in den Code zu vermeiden.

Teststruktur im Detail (Fortsetzung)

```
1 // Weiterer Test in der Gruppe
2 test("zeigt den richtigen Inhalt", async ({ page }) => {
3   const heading = page.locator("h1");
4   await expect(heading).toHaveText("Example Domain");
5   const paragraph = page.locator("p").first();
6   await expect(paragraph).toContainText("for illustrative examples");
7 });
8
9 // Test mit Bedingung überspringen
10 test("zeigt Benutzeroberfläche an", async ({ page, browserName }) => {
11   // Test nur in Chrome ausführen
12   test.skip(browserName !== "chromium", "Nur in Chrome getestet");
13
14   // Testlogik ...
15 });
16
17 // Weitere Tests ...
18 test("prüft Seitennavigation", async ({ page }) => {
19   await page.getByRole("link", { name: "More information" }).click();
20   await expect(page).toHaveURL(/.*iana.org/);
21   await expect(page).toHaveTitle(/IANA/);
22 });
```

Test-Hooks und Fixtures

```
1 test.describe("Produktseiten-Tests", () => {
2     // Vor allen Tests in dieser Gruppe ausführen
3     test.beforeAll(async ({ browser }) => {
4         // Gemeinsames Setup, z.B. Testdaten erstellen
5         const context = await browser.newContext();
6         const page = await context.newPage();
7         await page.goto("https://example.com/admin");
8         // Testprodukte anlegen...
9         await context.close();
10    });
11
12    // Nach allen Tests in dieser Gruppe ausführen
13    test.afterAll(async () => {
14        // Aufräumen, z.B. Testdaten löschen
15    });
16});
```

Test-Hooks und Fixtures (Fortsetzung)

```
1 // Vor jedem Test ausführen
2 test.beforeEach(async ({ page }) => {
3   await page.goto("https://example.com/products");
4   // Weitere Setup-Schritte ...
5 });
6
7 // Nach jedem Test ausführen
8 test.afterEach(async ({ page }) => {
9   // Screenshots erstellen
10  await page.screenshot({ path: "test-results/screenshot.png" });
11  // Logs sammeln, etc.
12 });
13
14 // Einzelne Tests ...
15 test("Produkt ist sichtbar", async ({ page }) => {
16   await expect(page.getText("Produkt 1")).toBeVisible();
17 });
```

Accessibility Testing mit Playwright

- Barrierefreiheit (Accessibility, a11y): Digitale Produkte für alle Menschen nutzbar machen
- Warum testen?
 - Gesetzliche Vorgaben (z.B. BITV, WCAG)
 - Bessere Usability für alle
 - Frühzeitiges Erkennen von Problemen spart Kosten
- Automatisierte Tests: Finden viele, aber nicht alle Probleme

Warum Accessibility wichtig ist

- **Gesetzliche Anforderungen:**
 - BITV 2.0 (Barrierefreie-Informationstechnik-Verordnung)
 - Barrierefreiheitsstärkungsgesetz (BFSG) ab 28. Juni 2025
 - Internationale WCAG-Standards
- **Wirtschaftliche Vorteile:**
 - Erweiterung der Zielgruppe
 - Verbesserte SEO durch semantische Struktur
 - Höhere Kundenzufriedenheit und -bindung
- **Technische Benefits:**
 - Robustere und wartbarere Code-Struktur
 - Bessere Testbarkeit durch semantische Locators

Accessibility-Checks mit Playwright & axe-core

- `@axe-core/playwright`: Integration des beliebten axe Accessibility-Scanners
- **Typische Checks:**
 - Fehlende Labels, schlechte Kontraste, doppelte IDs, etc.
- **Integration:**
 - `npm install --save-dev @axe-core/playwright`
 - Im Test importieren und verwenden
- **Bonus:** Erkenntnisse aus Accessibility-Tools (wie `axe-core` oder Chrome DevTools) helfen nicht nur Barrierefreiheitsprobleme zu beheben, sondern auch die **besten und robustesten Playwright-Locators** zu finden!

Accessibility Regeln

- WCAG 2.0, 2.1, 2.2 Konformität
 - Level A (Basis-Anforderungen)
 - Level AA (Wichtige Verbesserungen)
 - Level AAA (Optimale Zugänglichkeit)
- Häufige Testfälle:
 - Fehlende Alt-Texte für Bilder
 - Unzureichende Farbkontraste
 - Fehlende ARIA-Labels
 - Fehlende Überschriften-Hierarchie
 - Tastatur-Navigation
 - Formular-Labels und Fehlermeldungen

Beispiel: Accessibility-Test

```
1 import { test, expect } from "@playwright/test";
2 import AxeBuilder from "@axe-core/playwright";
3
4 test("Seite ist barrierefrei", async ({ page }) => {
5   await page.goto("https://example.com");
6   const results = await new AxeBuilder({ page }).analyze();
7   expect(results.violations).toEqual([]);
8 });
9
10 test('', async ({ page }) => {
11   test('Seite erfüllt WCAG 2.1 AA Standards', async ({ page }) => {
12     await page.goto("https://example.com");
13     const accessibilityScanResults = await new AxeBuilder({ page })
14       .withTags(['wcag2a', 'wcag2aa', 'wcag21a', 'wcag21aa'])
15       .analyze();
16     expect(accessibilityScanResults.violations).toEqual([]);
17   });
});
```

DevTools Recap: A11y & Locators

Erinnerung an Tag 1:

- Der "**Accessibility**"-Tab in den Chrome DevTools ist entscheidend.
- Er zeigt den **Accessible Name** und die **ARIA-Rolle** eines Elements.
- Diese Informationen sind die Basis für Playwrights User-Faced Selectors:
 - `getByRole()`
 - `getByLabel()`
 - `getByText()`
 - `getByPlaceholder()`
 - `getByAltText()`
 - `getTitle()`

Übung 6 – Accessibility Testing

Ziel: Automatisierte Barrierefreiheits-Tests mit Axe-Core implementieren.

Basis A11y Tests

- Homepage und News Feed Accessibility
- WCAG 2.1 Level AA Compliance
- Detaillierte Violation Reports

Komponenten & Navigation

- Navigation und Forms prüfen
- Dark Mode Kontraste
- Mobile Touch Target Sizes
- Tab-Order testen
- Focus-Indikatoren prüfen
- Screen Reader Kompatibilität

Zeit: 30 Minuten

Nachbesprechung: Accessibility

Diskussion der Übungsergebnisse

Komplexe UI-Interaktionen

Vertiefung: Komplexe UI-Interaktionen

- **Web-Apps sind komplexer geworden:**
 - Dynamische Inhalte, die erst bei bestimmten Interaktionen sichtbar werden
 - Verschachtelte und komplexe Formularelemente
 - Hover-Menüs, Tooltips, Dropdowns, Dialoge
 - Tabellen mit sortierbaren Spalten und editierbaren Zellen
 - Drag & Drop, Slider, Multi-Select-Listen
- **Diese Interaktionen erfordern spezielle Techniken:**
 - Gezielte Hover-Aktionen für Interaktionen mit dynamisch erscheinenden Elementen
 - Präzise Tastatur-Events für Shortcuts oder spezielle Eingaben
 - Explizite Wartestrategien, wenn Auto-Wait nicht ausreicht

Beispiel: Hover und dynamisch erscheinende Elemente

```
1  test("Löschen eines Todo-Items mit Hover-Aktion", async ({ page }) => {
2      await page.goto("https://demo.playwright.dev/todomvc/");
3      await page.getByPlaceholder("What needs to be done?").fill("Hover-Test");
4      await page.keyboard.press("Enter");
5
6      // Prüfen, dass Item existiert
7      await expect(page.getByText("Hover-Test")).toBeVisible();
8
9      // Hover über dem Element, um den Lösch-Button sichtbar zu machen
10     await pageByText("Hover-Test").hover();
11
12     // Klick auf den Lösch-Button (der erst nach Hover sichtbar ist)
13     // In TodoMVC ist der Lösch-Button ein Element mit der Klasse "destroy"
14     await page.locator('.todo-list li').filter({ hasText: 'Hover-Test' })
15         .getByRole('button', { name: 'Delete' })
16         .click();
17
18     // Prüfen, dass Item gelöscht wurde
19     await expect(page.getByText("Hover-Test")).not.toBeVisible();
```

Beispiel: Tabellen-Interaktionen

```
1  test("Tabelleninteraktionen", async ({ page }) => {
2      await page.goto('https://demoqa.com/webtables');
3
4      // Suche eine bestimmte Zelle in der Tabelle mit Hilfe von Rollen-Lokatoren
5      const emailCell = page.getByRole('gridcell', { name: 'alden@example.com' });
6      await expect(emailCell).toBeVisible();
7
8      // Bearbeite einen Benutzer durch Klicken des Bearbeiten-Buttons in einer bestimmten Zeile
9      await page
10         .getByRole('row', { name: '/alden@example.com/' })
11         .locator('[id^="edit-record"]')
12         .click();
13
14      await page.locator('[id^="edit-record"]').first().click();
15
16      // Füllt das Formular mit neuen Daten aus
17      await page
18         .getByRole('textbox', { name: 'First Name' })
19         .fill('Geänderter Name');
```

Beispiel: Tastatur-Interaktionen

```
1 // In vielen Web-Apps sind Tastaturkürzel wichtige Interaktionsmöglichkeiten
2 test("Tastaturinteraktionen mit Todo-Items", async ({ page }) => {
3     // Setup: Todo-Item erstellen
4     await page.goto("https://demo.playwright.dev/todomvc/");
5     await page.getByPlaceholder("What needs to be done?").fill("Tastatur-Test");
6     await page.keyboard.press("Enter");
7
8     // Mit Tab zum Todo-Item navigieren
9     await page.keyboard.press("Tab");
10
11    // Mit Leertaste das Checkbox-Element aktivieren
12    await page.keyboard.press("Space");
13
14    // Bearbeiten eines Todos durch Doppelklick
15    const todoItem = page.getText("Tastatur-Test");
16    await todoItem dblclick();
17
18    // Text im Bearbeitungsmodus ändern
19    await page.keyboard.press("Control+A"); // Alles auswählen
```

Frames und iFrames

Arbeiten mit Frames und iFrames

- **Was sind Frames/iFrames?**
 - Eingebettete HTML-Dokumente innerhalb einer Seite
 - Eigener DOM-Kontext und JavaScript-Scope
 - Häufig für Widgets, Werbung, Payment-Gateways
- **Herausforderungen beim Testen:**
 - Locators funktionieren nicht über Frame-Grenzen
 - Jeder Frame hat seinen eigenen Kontext
 - Cross-Origin Frames haben Sicherheitsbeschränkungen
- **Playwright's Frame-APIs:**
 - Nahtlose Navigation zwischen Frames
 - Automatisches Warten auf Frame-Verfügbarkeit
 - Unterstützung für verschachtelte Frames

Frame Locators verwenden

```
1 // Frame über Name oder URL finden
2 const frame = page.frame('frame-name');
3 const frame = page.frame({ url: /.*/domain.*/ });
4
5 // FrameLocator für bessere Stabilität
6 const frameLocator = page.frameLocator('#my-iframe');
7 await frameLocator.getByRole('button', { name: 'Submit' }).click();
8
9 // Verschachtelte Frames
10 const nestedFrame = page
11   .frameLocator('#outer-frame')
12   .frameLocator('#inner-frame');
13 await nestedFrame.getText('Nested content').click();
14
15 // Frame-Inhalt prüfen
16 await expect(frameLocator.getByRole('heading')).toHaveText('Frame Title');
```

Praktisches Frame-Beispiel: Payment Gateway

```
1 test('Payment mit iFrame Gateway', async ({ page }) => {
2   await page.goto('/checkout');
3
4   // Auf Payment-Frame warten und damit interagieren
5   const paymentFrame = page.frameLocator('#payment-gateway');
6
7   // Kreditkartendaten eingeben (im Frame)
8   await paymentFrame.getByLabel('Card Number').fill('4111111111111111');
9   await paymentFrame.getByLabel('Expiry').fill('12/25');
10  await paymentFrame.getByLabel('CVV').fill('123');
11
12  // Submit im Frame
13  await paymentFrame.getByRole('button', { name: 'Pay Now' }).click();
14
15  // Zurück zum Hauptframe für Bestätigung
16  await expect(page.getText('Payment successful')).toBeVisible();
17});
```

Dialog-Handling (Alerts, Confirms, Prompts)

- **Browser-Dialoge:**
 - `alert()` - Einfache Benachrichtigung
 - `confirm()` - Ja/Nein Entscheidung
 - `prompt()` - Texteingabe vom Benutzer
 - `beforeunload` - Seite verlassen Warnung
- **Automatische Behandlung:**
 - Playwright kann Dialoge automatisch akzeptieren/ablehnen
 - Standard: Dialoge werden automatisch dismissed
- **Explizite Kontrolle:**
 - Event-Listener für Dialog-Ereignisse
 - Programmatische Entscheidung pro Dialog
 - Prompt-Texte eingeben

Dialog Event Handler

```
1 // Dialog automatisch akzeptieren
2 page.on('dialog', async dialog => {
3   console.log(`Dialog message: ${dialog.message()}`);
4   await dialog.accept();
5 });
6
7 // Confirm-Dialog mit Bedingung
8 page.on('dialog', async dialog => {
9   if (dialog.type() === 'confirm') {
10     if (dialog.message().includes('delete')) {
11       await dialog.dismiss(); // Abbrechen bei Löschung
12     } else {
13       await dialog.accept();
14     }
15   }
16 });
17
18 // Prompt mit Texteingabe
19 page.on('dialog', async dialog => {
```

Dialog-Handling Best Practices

```
1  test('Dialog-Handling mit Assertions', async ({ page }) => {
2      // Dialog-Handler vor der Aktion setzen
3      let dialogMessage = '';
4
5      page.once('dialog', async dialog => {
6          dialogMessage = dialog.message();
7          await dialog.accept();
8      });
9
10     // Aktion, die Dialog triggert
11     await page.getByRole('button', { name: 'Delete' }).click();
12
13     // Dialog-Nachricht prüfen
14     expect(dialogMessage).toBe('Are you sure you want to delete?');
15
16     // UI-Zustand nach Dialog prüfen
17     await expect(page.getText('Item deleted')).toBeVisible();
18 });
19
20 // BeforeUnload Event
21 test('Ungespeicherte Änderungen Warnung', async ({ context }) => {
22     // BeforeUnload-Dialoge aktivieren (standardmäßig deaktiviert)
```

File Downloads

- **Download-Events:**
 - Automatisches Warten auf Download-Start
 - Zugriff auf Download-Metadaten
 - Download-Pfad und Dateiname
- **Download-Handling:**
 - Downloads werden in temporärem Verzeichnis gespeichert
 - Automatische Bereinigung nach Test
 - Manuelles Speichern an gewünschtem Ort
- **Anwendungsfälle:**
 - Export-Funktionen testen
 - Datei-Inhalt validieren
 - Download-Performance messen

Download-Beispiele

```
1 // Einfacher Download
2 test('PDF Download', async ({ page }) => {
3     // Download-Promise vor dem Klick starten
4     const downloadPromise = page.waitForEvent('download');
5
6     // Download triggern
7     await page.getByRole('button', { name: 'Download PDF' }).click();
8
9     // Auf Download warten
10    const download = await downloadPromise;
11
12    // Download-Informationen
13    console.log('Dateiname:', await download.suggestedFilename());
14    console.log('URL:', download.url());
15
16    // Download speichern
17    await download.saveAs('./downloads/' + download.suggestedFilename());
18
19    // Oder als Buffer für Validierung
```

Download-Validierung

```
1  test('CSV Export validieren', async ({ page }) => {
2    // Download starten
3    const downloadPromise = page.waitForEvent('download');
4    await page.getByRole('button', { name: 'Export to CSV' }).click();
5    const download = await downloadPromise;
6
7    // Dateiname prüfen
8    expect(download.suggestedFilename()).toMatch(/export-\d{4}-\d{2}-\d{2}\.csv/);
9
10   // Inhalt lesen und validieren
11   const path = await download.path();
12   const content = await fs.readFile(path, 'utf-8');
13
14   // CSV-Inhalt prüfen
15   const lines = content.split('\n');
16   expect(lines[0]).toBe('Name,Email,Date'); // Header
17   expect(lines.length).toBeGreaterThan(1); // Hat Daten
18
19   // Download-Fehler behandeln
```

Session Handling

Authentifizierungsstrategien im Vergleich

- UI-basierte Authentifizierung
 - Testet den vollständigen Login-Flow
 - Prüft UI-Elemente und Validierungen
 - Langsamer und fehleranfälliger
 - Realistischste Benutzererfahrung
- API-basierte Authentifizierung:
 - Direkter API-Aufruf zum Login
 - Umgeht UI-Interaktionen
 - Schneller und zuverlässiger
 - Weniger realistisch, aber effizienter

Authentifizierungsmethoden

- **Token-basierte Auth:**
 - JWT (JSON Web Tokens)
 - Access & Refresh Tokens
 - Stateless Authentication
- **Session-basierte Auth:**
 - Cookie-basierte Sessions
 - Server-seitige Speicherung
 - Stateful Authentication
- **OAuth & OpenID Connect:**
 - Delegierte Authentifizierung
 - Single Sign-On (SSO)
 - Autorisierungscode-Flow

Herausforderungen beim Login Testing

- Wiederholtes Login in jedem Test ist ineffizient
- Sitzungsdauer: Tokens können ablaufen
- UI-Login ist langsam und fehleranfällig
- Abhängigkeiten: Tests hängen vom Auth-Provider ab
- Sicherheit: Testen mit echten Anmeldedaten

Tipp: Arbeiten mit Umgebungsvariablen

- Umgebungsvariablen sind ideal für:
 - Base URLs (`process.env.BASE_URL`)
 - Test-Zugangsdaten (`process.env.TEST_USER` , `process.env.TEST_PASSWORD`)
 - Konfigurationsunterschiede zwischen Umgebungen
- In `playwright.config.ts` verwenden:

```
// Nutzung der BASE_URL aus einer Umgebungsvariable
use: {
  baseURL: process.env.BASE_URL || 'https://demo.playwright.dev/todomvc',
}
```

- In Tests verwenden:

```
test('Login mit Env-Credentials', async ({ page }) => {
  await page.goto('https://www.saucedemo.com/');
  await page.getLabel('user-name').fill(process.env.TEST_USER || 'standard_user');
  await page.getLabel('password').fill(process.env.TEST_PASSWORD || 'secret_sauce');
});
```

Tipp: Umgebungsvariablen mit dotenv

- **Mit dotenv für lokale Entwicklung:**
 - Installieren: `npm install --save-dev dotenv`
 - In `playwright.config.ts` : `require('dotenv').config()`
 - `.env` Datei erstellen mit `TEST_USER=myuser`
 - `.env` Datei zu `.gitignore` hinzufügen
- **Best Practices:**
 - Verwende Fallback-Werte für lokale Entwicklung
 - Dokumentiere alle benötigten Umgebungsvariablen
 - Verwende aussagekräftige Variablennamen
 - Trenne Test- und Produktions-Credentials

Authentifizierungsstrategien für Tests

Für E2E-Tests

- Storage State für wiederholte Tests
- API-basierte Auth für Effizienz
- Mock Auth Provider für Isolation
- Test-spezifische Benutzer

Für Komponententests

- Auth-Context mocken
- Direkte Zustandsmanipulation
- Geschützte Routen umgehen
- Berechtigungen simulieren

Sicherheitsaspekte beim Auth-Testing

- **Niemals produktive Credentials in Tests:**
 - Separate Testbenutzer verwenden
 - Testumgebung isolieren
- **Sichere Speicherung von Testdaten:**
 - Umgebungsvariablen für Credentials
 - Secrets-Management in CI/CD
- **Datenschutz beachten:**
 - Keine echten Benutzerdaten in Tests
 - Testdaten nach Testlauf bereinigen

Session-Handling mit Playwright – Theorie

- **Browser-Kontext:** Isolierte Browser-Umgebung
- **Storage State:** Speichern von Cookies und LocalStorage
- **Wiederverwendung:** State in neue Kontexte laden
- **Fixture-basiert:** Integration in Playwright's Testmodell

Storage State im Detail

```
1  // Struktur eines Storage State
2  {
3      "cookies": [
4          {
5              "name": "session",
6              "value": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 ... ",
7              "domain": "example.com",
8              "path": "/",
9              "expires": 1661201000,
10             "httpOnly": true,
11             "secure": true,
12             "sameSite": "Lax"
13         }
14     ],
15     "origins": [
16         {
17             "origin": "https://example.com",
18             "localStorage": [
19                 {
20                     "name": "token",
21                     "value": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 ... "
22                 },
23             ]
24         }
25     ]
26 }
```

UI-Auth Flow

```
1 // Login durchführen und Storage State speichern
2 import { test as setup, expect } from '@playwright/test';
3 import path from 'path';
4
5 const authFile = path.join(__dirname, '../playwright/.auth/user.json');
6
7 setup('authenticate', async ({ page }) => {
8     await page.goto('https://github.com/login');
9
10    await page.getLabel('Username or email address').fill('username');
11    await page.getLabel('Password').fill('password');
12    await page.getRole('button', { name: 'Sign in' }).click();
13
14    await page.waitForURL('https://github.com/');
15    await expect(page.getRole('button', { name: 'View profile and more' })).toBeVisible();
16
17    await page.context().storageState({ path: authFile });
18});
```

API-basierte Authentifizierung

```
1 // Effizienter Login über API statt UI
2 import { test as setup } from '@playwright/test';
3
4 const authFile = 'playwright/.auth/user.json';
5
6 setup('authenticate', async ({ request }) => {
7     // Send authentication request. Replace with your own.
8     await request.post('https://github.com/login', {
9         form: {
10             'user': 'user',
11             'password': 'password'
12         }
13     });
14     await request.storageState({ path: authFile });
15 });


```

Storage State eibinden

```
1 // In playwright.config.ts
2 import { defineConfig, devices } from '@playwright/test';
3
4 export default defineConfig({
5   projects: [
6     { name: 'setup', testMatch: /\.+\.\.setup\.ts/ },
7     {
8       name: 'chromium',
9       use: {
10         ...devices['Desktop Chrome'],
11         storageState: 'playwright/.auth/user.json',
12       },
13       dependencies: ['setup'],
14     },
15   ],
16 });

```

Mehrere Benutzerrollen in Projekten

```
1 // In playwright.config.ts
2 import { defineConfig } from "@playwright/test";
3
4 export default defineConfig({
5   projects: [
6     // Setup-Projekte für verschiedene Rollen
7     {
8       name: "setup-admin",
9       testMatch: /admin-setup\.\ts/,
10    },
11    {
12      name: "setup-editor",
13      testMatch: /editor-setup\.\ts|,
14    },
15
16     // Test-Projekte für verschiedene Rollen
17     {
18       name: "admin-tests",
19       dependencies: ["setup-admin"],
20       use: {
21         storageState: "auth-admin.json",
22       },
23     },
24   ],
25 }
```

Browser Context: Grundlagen

- **Browser Context:** Ermöglicht isolierte Browser-Sitzungen
 - Jeder Context hat eigene Cookies, Storage & Auth-Status
 - Perfekt für parallele Tests mit verschiedenen Benutzerrollen
 - Verhindert Interferenzen zwischen Testfällen
- **Auth-Vorteile:**
 - Separate Login-Sessions pro Context
 - Storage State kann pro Context gespeichert werden
 - Einfaches Testen von Multi-User-Szenarien
- **Best Practices:**
 - Context nach Test beenden mit `context.close()`
 - Storage State für Auth-Status wiederverwenden
 - Isolierte Tests durch separate Contexts

Browser Context: Multi-Role Implementation

```
1 // tests/multi-role.spec.ts
2 import { test, expect } from '@playwright/test';
3
4 // Beispiel: Interaktion zwischen Admin und User in einem Test
5 test('authenticate', async ({ browser }) => {
6     // Admin-Kontext mit gespeicherter Authentifizierung
7     const adminContext = await browser.newContext({
8         storageState: 'playwright/.auth/admin.json',
9     });
10    const adminPage = await adminContext.newPage();
11
12    // User-Kontext mit gespeicherter Authentifizierung
13    const userContext = await browser.newContext({
14        storageState: 'playwright/.auth/user.json',
15    });
16    const userPage = await userContext.newPage();
17
18    // Beispielhafte Interaktion
19    await adminPage.goto('/admin-dashboard');
```

Übung 7 – Authentifizierung optimieren

Ziel: Verschiedene Authentifizierungs-Ansätze: UI-Login und API-basiert.

Teil A: UI-basierte Authentifizierung

1. Login-Setup (`auth.setup.ts`) mit UI-Login erstellen.
2. Auth-Status in `playwright/.auth/user.json` speichern.
3. `playwright.config.ts` : Setup-Projekt mit Dependencies konfigurieren.

Teil B: API-basierte Optimierung

1. Login via API: CSRF-Token holen, Credentials senden.
2. Schnellere Tests durch direkten API-Zugriff.
3. Multi-Role Testing (optional): Admin vs. User.

Zeit: 35 Minuten

Global Setup & Teardown

- Einmalige Vorbereitung für alle Tests
- Datenbank-Seeding, API-Token generieren
- Authentifizierung für alle Test-Sessions
- Test-Accounts erstellen
- Läuft vor allen Tests - nur einmal

Global Setup Implementierung

```
1 // global-setup.ts
2 import { chromium, FullConfig } from "@playwright/test";
3
4 async function globalSetup(config: FullConfig) {
5   const { baseURL, storageState } = config.projects[0].use;
6   const browser = await chromium.launch();
7   const context = await browser.newContext();
8   const page = await context.newPage();
9
10  // 1. API-Login
11  const response = await page.request.post(`${baseURL}/api/auth/login`, {
12    data: {
13      username: process.env.TEST_USERNAME || "testuser",
14      password: process.env.TEST_PASSWORD || "testpass123",
15    },
16  });
17
18  const { token, refreshToken } = await response.json();
19
20  // 2. Cookies setzen
21  await context.addCookies([
22    {
```

Global Teardown - Aufräumen

```
1 // global-teardown.ts
2 import { FullConfig } from "@playwright/test";
3 import fs from "fs/promises";
4
5 async function globalTeardown(config: FullConfig) {
6   const { baseURL } = config.projects[0].use;
7
8   // 1. Testdaten über API löschen
9   try {
10     const storageState = JSON.parse(
11       await fs.readFile("storageState.json", "utf-8")
12     );
13
14     const token = storageState.cookies.find(
15       (c: any) => c.name === "auth_token"
16     )?.value;
17
18     if (token) {
19       await fetch(`#${baseURL}/api/test/cleanup`, {
20         method: "DELETE",
21         headers: { "Authorization": `Bearer ${token}` }
22       });
23     }
24   } catch (err) {
25     console.error("Error during global teardown: ", err);
26   }
27 }
```

Konfiguration mit Dependencies

```
1 // playwright.config.ts
2 export default defineConfig({
3   globalSetup: require.resolve('./global-setup'),
4   globalTeardown: require.resolve('./global-teardown'),
5
6   projects: [
7     // Setup-Projekt muss zuerst laufen
8     {
9       name: "setup",
10      testMatch: /global\.\setup\.\ts/,
11    },
12
13     // Tests mit Dependency auf Setup
14     {
15       name: "chromium",
16       use: [
17         ... devices["Desktop Chrome"],
18         storageState: "storageState.json"
19       },

```

Worker-Scope Fixtures

```
1 // fixtures.ts - Geteilte Ressourcen pro Worker
2 import { test as base } from "@playwright/test";
3
4 export const test = base.extend<{}, { dbConnection: DatabaseConnection }>({
5     // Worker-Fixture (einmal pro Worker-Prozess)
6     dbConnection: [async ({}, use) => {
7         // Setup: Einmal pro Worker
8         const connection = await createDatabaseConnection();
9         await connection.query("BEGIN TRANSACTION");
10
11         // Fixture bereitstellen
12         await use(connection);
13
14         // Teardown: Nach allen Tests im Worker
15         await connection.query("ROLLBACK");
16         await connection.close();
17     }, { scope: "worker" }],
18
19     // Test-scoped Fixture nutzt Worker-Fixture
```

Fixtures – Einführung & Nutzen

- **Fixtures:** Wiederverwendbare Bausteine für Tests
- **Vorteile:**
 - Test-Daten vorbereiten
 - Setup und Cleanup automatisieren
 - Zwischen Tests geteilt werden
 - Weniger Code-Duplikation

```
1 test("Mit eingebauten Fixtures", async ({ page, context, browser }) => {
2   // page, context, browser sind bereits bereitgestellt!
3});
```

Warum Fixtures nutzen?

✗ Ohne Fixtures:

```
1 test('user test 1', async ({ page }) => {
2   const userData = {
3     name: 'Test User',
4     email: 'test@example.com'
5   };
6   await page.goto('/users');
7   await page.getByLabel('Name').fill(userData.name);
8   // ... duplicate setup code
9 });
10
11 test('user test 2', async ({ page }) => {
12   const userData = {
13     name: 'Test User',
14     email: 'test@example.com'
15   };
16   await page.goto('/users');
17   // ... same setup code again
18 });
```

✓ Mit Fixtures:

```
1 test('user test 1', async ({ testUser, userPage }) => {
2   await userPage.addUser(testUser);
3   // Clean, focused test logic
4 });
5
6 test('user test 2', async ({ testUser, userPage }) => {
7   await userPage.editUser(testUser);
8   // No duplicate setup
9 });
```

Eingebaute Playwright-Fixtures

Fixture	Beschreibung	Scope
page	Einzelne Browserseite	Test
context	Browser-Kontext (wie Inkognito)	Test
browser	Browser-Instanz	Worker
browserName	Name des Browsers (chromium, firefox, webkit)	Worker
playwright	Playwright-API-Instanz	Worker
request	API für HTTP-Anfragen	Worker

Einfache Test-Data Fixture

```
1 import { test as base, expect } from '@playwright/test';
2
3 // Definiere eine einfache Fixture für Test-Daten
4 const test = base.extend<{ testUser: { name: string; email: string; role: string } }>({
5   testUser: async ({}, use) => {
6     // Setup: Erstelle eindeutige Test-Daten
7     const userData = {
8       name: `Test User ${Date.now()}`,
9       email: `test-${Date.now()}@example.com`,
10      role: 'user'
11    };
12
13    console.log('✅ Test user data prepared:', userData.name);
14
15    // Fixture bereitstellen
16    await use(userData);
17
18    // Teardown (hier optional)
19    console.log('✗ Test user data cleanup completed');
20  },
21});
22
```

Page Helper Fixture

```
1 // Erweitere das Interface
2 interface FixturesDemo {
3   testUser: { name: string; email: string; role: string };
4   userPage: {
5     addUser: (user: { name: string; email: string; role: string }) => Promise<void>;
6     getUserCount: () => Promise<number>;
7   };
8 }
9
10 const test = base.extend<FixturesDemo>({
11   testUser: async ({}, use) => {
12     const userData = {
13       name: `Test User ${Date.now()}`,
14       email: `test-${Date.now()}@example.com`,
15       role: 'moderator'
16     };
17     await use(userData);
18   },
19
20   userPage: async ({ page }, use) => {
21     await page.goto('/fixtures-demo');
```

Fixture Scopes

Test-scoped (Standard)

-  **Neue Instanz** für jeden Test
-  **Schnell** für einfache Daten
-  **Isolation** zwischen Tests

Worker-scoped

-  **Eine Instanz** pro Worker
-  **Teuer** für Datenbank-Connections
-  **Shared State** zwischen Tests

Worker-scoped Fixture Beispiel

```
1 // Worker-scoped Fixture für teure Ressourcen
2 const test = base.extend<{}, { sharedDB: Database }>({
3   sharedDB: [
4     async ({}, use) => {
5       // Setup: Einmal pro Worker-Prozess
6       console.log('🌐 Creating database connection ...');
7       const db = await createDatabase();
8
9       // Fixture bereitstellen für alle Tests im Worker
10      await use(db);
11
12      // Teardown: Nach allen Tests im Worker
13      await db.close();
14      console.log('🌐 Database connection closed');
15    },
16    { scope: 'worker' } // Worker-scoped!
17  ],
18
19 // Test-scoped Fixture kann Worker-scoped nutzen
20 testData: async ({ sharedDB }, use) => {
21   const data = await sharedDB.query('SELECT * FROM test_data');
22   await use(data);`
```

Best Practices für Fixtures

✓ Do's

- **Eindeutige Namen:** `testUser` statt `data1`
- **Eine Verantwortung:** Eine Fixture, eine Aufgabe
- **TypeScript nutzen:** Bessere Entwicklererfahrung
- **Proper Cleanup:** Teardown für Seiteneffekte
- **Semantic Locators:** `getByLabel()`,
`getByRole()`

✗ Don'ts

- **Keine verschachtelten Dependencies**
- **Kein geteilter Zustand** (außer worker-scoped)
- **Keine Test-Logik in Fixtures**
- **Keine Seiteneffekte ohne Cleanup**
- **Keine Test-IDs:** Use user-facing locators

Live Demo: Fixtures Exercise

⌚ Übung 10: Fixtures

Szenario: User Management Interface testen

1. Test-Data Fixture

- Eindeutige Benutzerdaten pro Test erstellen
- Setup/Teardown-Lifecycle beobachten

2. Page Helper Fixture

- `addUser()` und `getUserCount()` Funktionen entwickeln
- Wiederverwendbare Page-Operationen abstrahieren

3. Tests ausführen

- Code-Duplikation vermeiden
- Fixture-Benefits erleben

Zeit: 20-25 Minuten | 👉 <http://localhost:3000/fixtures-demo>

Wann Fixtures verwenden?

💚 Perfekt für:

- 🔑 Test-Daten vorbereiten
- 📄 Page-Helper Funktionen
- 🔒 Authentifizierung/Setup
- 🌐 API-Client Konfiguration
- 🗄 Datenbank-Connections

◆ Nicht optimal für:

- Einfache Variablen
- Test-spezifische Logik
- Einmalige Operationen
- Komplexe Geschäftslogik

💡 Tipp

Beginne mit einfachen **Test-Data** Fixtures. Erweitere sie schrittweise zu **Page Helpers**, wenn du Wiederholung in deinen Tests siehst!

Test-Strukturierung vor dem POM

Warum strukturierten Testcode schreiben?

- **Probleme unstrukturierter Tests:**
 - Duplizierter Code für wiederkehrende Aktionen und Locators
 - Schwer zu lesen, zu verstehen und zu warten
 - Änderungen an der UI erfordern Anpassungen an vielen Stellen
 - Höhere Fehleranfälligkeit mit zunehmender Testanzahl
- **Evolutionsstufen der Teststrukturierung:**
 1. **Einfache Tests:** Direkter Code ohne besondere Struktur
 2. **Hilfsfunktionen:** Wiederverwendbare Funktionen in der Testdatei
 3. **Fixture-basiert:** Wiederkehrendes Setup in Fixtures auslagern
 4. **Page Object Model:** Vollständige Kapselung von Seiteninteraktionen

Erste Schritte: Hilfsfunktionen

```
1 // Duplizierter, unstrukturierter Code:  
2 await page.goto('https://demo.playwright.dev/todomvc/');  
3 await page.getByPlaceholder("What needs to be done?").fill("Todo 1");  
4 await page.keyboard.press("Enter");  
5  
6 // Besser: Hilfsfunktion zur Wiederverwendung  
7 async function createTodo(page, text) {  
8     await page.getByPlaceholder("What needs to be done?").fill(text);  
9     await page.keyboard.press("Enter");  
10 }  
11  
12 // Verwendet in Tests:  
13 test('Todo anlegen', async ({ page }) => {  
14     await page.goto('https://demo.playwright.dev/todomvc/');  
15     await createTodo(page, "Todo 1");  
16     await expect(page.getText("Todo 1")).toBeVisible();  
17 });  
18  
19 // Gemeinsame Locators für einfacheren Zugriff  
20 function todoSelectors(page) {  
21     return {  
22         newTodoInput: page.getByPlaceholder("What needs to be done?"),
```

Vor- und Nachteile einfacher Hilfsfunktionen

Vorteile

- Einfach zu implementieren
- Reduziert Code-Duplikation
- Bessere Lesbarkeit
- Einfacher Einstieg in Strukturierung
- Tests bleiben übersichtlich

Nachteile

- Funktionen nur innerhalb einer Datei nutzbar
- Keine klare Kapselung von Zuständen
- Bei komplexen UIs schnell unübersichtlich
- Keine Standardisierung über Testdateien hinweg
- Nicht ideal für größere Projekte

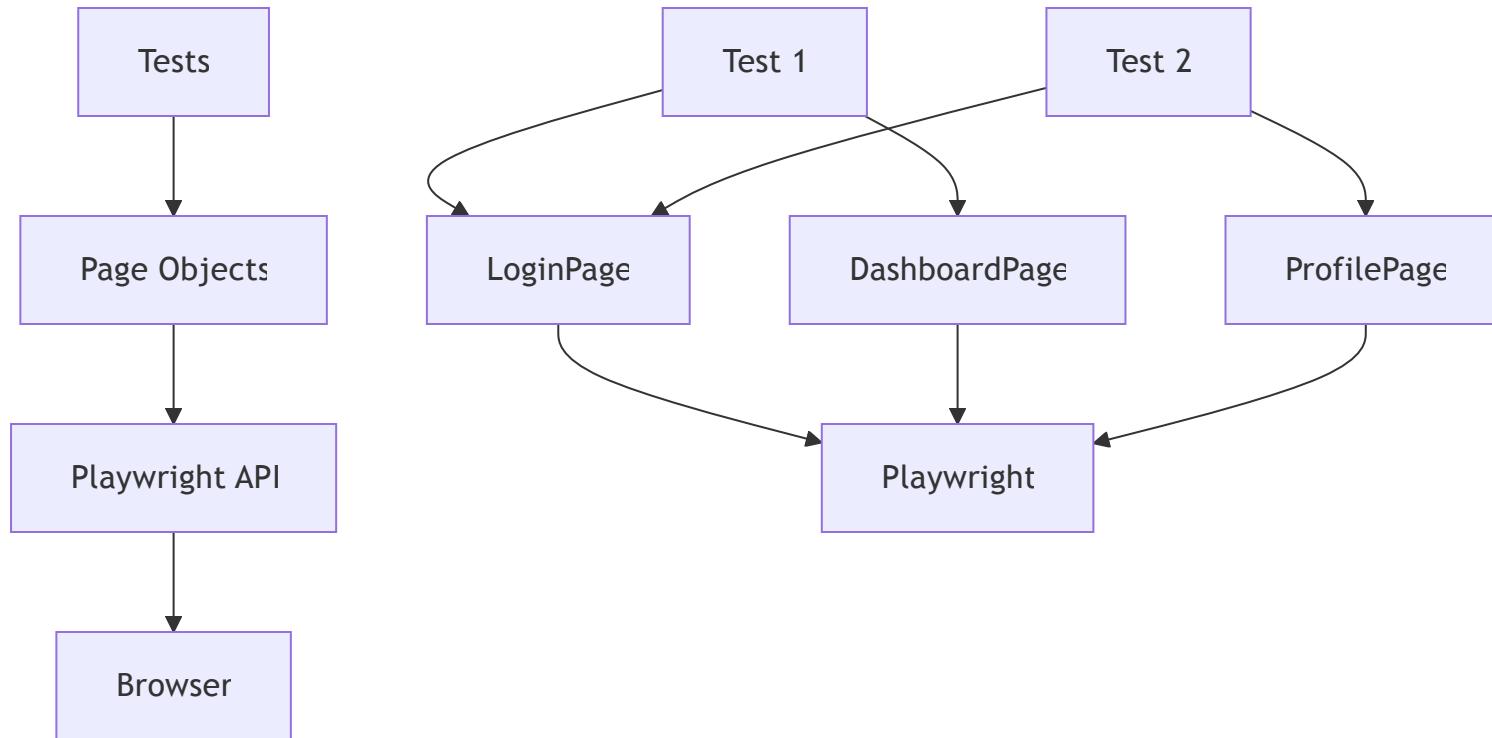
Tipp

Hilfsfunktionen sind ein guter erster Schritt zur besseren Testorganisation. Bei wachsender Komplexität ist jedoch der Übergang zum Page Object Model sinnvoll!

Page Object Model – Einführung

- Design Pattern zur Strukturierung von Tests
- Abstrahiert Seiteninteraktion in Klassen
- Kapselt Seitenelemente und -funktionen
- Vermeidet Code-Duplikation
- Vereinfacht Test-Wartung

Page Object Model – Architektur



- **Tests:** Beschreiben Geschäftslogik und Erwartungen

Nachbesprechung: POM Intro

Diskussion der Übungsergebnisse

Alternativen zum Page Object Model

Component Object Model

- Fokus auf UI-Komponenten statt Seiten
- Besser für SPAs und komplexe UIs
- Wiederverwendbare Komponenten
- Beispiel: Header, Sidebar, Modal

App Actions Pattern

- Fokus auf Benutzeraktionen
- Abstrahiert UI-Details
- API-ähnliche Schnittstelle
- Beispiel: `app.login()`, `app.createTodo()`

```
1  class TodoItem {
2      constructor(private root: Locator) {}
3
4      async toggle() {
5          await this.root.getByRole("checkbox").click();
6      }
7      async delete() {
8          await this.root.getByRole("button").click();
9      }
10     async getText() {
11         return this.root.getByRole("textbox").textContent();
12     }
13 }
```

Vorteile des Page Object Models

- **Wartbarkeit:** Änderungen nur an einer Stelle
- **Lesbarkeit:** Tests beschreiben "Was", nicht "Wie"
- **Wiederverwendbarkeit:** Funktionen in mehreren Tests nutzen
- **Stabilität:** Reduziert Fehleranfälligkeit
- **Organisation:** Strukturierte Testarchitektur

Live-Demo: POM Refactoring

Praktische Demonstration der Konzepte

Best Practices für Page Objects

- **Verantwortlichkeiten trennen:**
 - Page Objects: UI-Interaktionen
 - Tests: Assertions und Geschäftslogik
- **Methoden sinnvoll benennen:**
 - Geschäftsprozesse abbilden & Domänsprache verwenden
 - Beispiel: `loginAsAdmin()` statt `fillEmailAndPassword()`
- **Rückgabewerte nutzen:**
 - Methoden geben neue Page Objects zurück
 - Beispiel: `loginPage.login().shouldShowDashboard()`
- **Assertions vermeiden:**
 - Page Objects sollten keine Assertions enthalten
 - Ausnahme: Hilfsmethoden wie `shouldBeVisible()`

Page Object Model – Beispielcode

```
1 import { Page, Locator } from "@playwright/test";
2 export class LoginPage {
3     readonly page: Page;
4     readonly emailInput: Locator;
5     readonly passwordInput: Locator;
6     readonly loginButton: Locator;
7
8     constructor(page: Page) {
9         this.page = page;
10        this.emailInput = page.getByLabel("Email");
11        this.passwordInput = page.getByLabel("Passwort");
12        this.loginButton = page.getByRole("button", { name: "Anmelden" });
13    }
14
15    async goto() {
16        await this.page.goto("/login");
17    }
18
19    async login(email: string, password: string) {
```

Umsetzung im Projekt

```
1 // e2e/login.spec.ts
2 import { test, expect } from "@playwright/test";
3 import { LoginPage } from "../pages/LoginPage";
4 import { DashboardPage } from "../pages/DashboardPage";
5
6 test("Erfolgreicher Login", async ({ page }) => {
7     const loginPage = new LoginPage(page);
8     const dashboardPage = new DashboardPage(page);
9
10    await loginPage.goto();
11    await loginPage.login("user@example.com", "password123");
12
13    // Prüfen, ob wir zum Dashboard weitergeleitet wurden
14    await expect(page).toHaveURL("/dashboard/");
15    await expect(dashboardPage.welcomeMessage).toBeVisible();
16});
```

Übung 8 – Page Object Model (POM)

Ziel: Tests aus Übung 5 mit Page Object Pattern refaktorieren.

Aufgaben:

1. NewsPage Klasse erstellen (e2e/pages/NewsPage.ts).
2. Locators als Getter definieren (lazy loading).
3. Aktionsmethoden: searchNews() , clearSearch() , goto() .
4. Helper-Methoden: getNewsCount() , getFirstNewsTitle() .
5. Tests refaktorieren: Page Object statt direkte Selektoren.

Zeit: 25 Minuten

Nachbesprechung: POM Best Practices

Diskussion der Übungsergebnisse

Zusammenfassung Tag 2 & Ausblick Tag 3

- Debugging-Techniken
- Authentifizierung & Storage State
- Fixtures für Setup/Teardown
- Page Object Model Grundlagen
- Morgen: Fortgeschrittenes POM, API Mocking, Testing-Strategien, Mobile, CI/CD

Fortgeschrittene Page Object Patterns

```
1 // Basis Page Object
2 abstract class BasePage {
3     constructor(protected page: Page) {}
4
5     async waitForPageLoad() {
6         await this.page.waitForLoadState("networkidle");
7     }
8
9     // Navigation mit Typensicherheit
10    async navigateTo<T extends BasePage>(PageClass: new (page: Page) => T, url: string): Promise<T> {
11        await this.page.goto(url);
12        return new PageClass(this.page);
13    }
14 }
15
16 // Konkrete Implementierung
17 class LoginPage extends BasePage {
18     private emailInput = this.page.getByLabel("Email");
19     private passwordInput = this.page.getByLabel("Passwort");
20
21     async login(email: string, password: string): Promise<DashboardPage> {
22         await this.emailInput.fill(email);
23         await this.passwordInput.fill(password);
24         await this.page.click("button[type='submit']");
25         return new DashboardPage(this.page);
26     }
27 }
```

Erweitertes Page Object Beispiel

```
1 // pages/TodoPage.ts
2 import { Page, Locator } from "@playwright/test";
3
4 export class TodoPage {
5     private readonly newTodoInput: Locator;
6     private readonly todoItems: Locator;
7     private readonly filterButtons: Record<string, Locator>;
8
9     constructor(private readonly page: Page) {
10         this.newTodoInput = page.getByPlaceholder("What needs to be done?");
11         this.todoItems = page.locator(".todo-list li");
12         this.filterButtons = {
13             active: page.getByRole("link", { name: "Active" }),
14             completed: page.getByRole("link", { name: "Completed" }),
15             all: page.getByRole("link", { name: "All" })
16         };
17     }
18
19     // Aktionen
20     async goto() {
21         await this.page.goto("https://todomvc.com/examples/vanilla-es6/");
22     }
}
```

Komponenten-basiertes Page Object Modell

```
1 // components/TodoItem.ts
2 import { Locator } from "@playwright/test";
3
4 export class TodoItem {
5     private readonly checkbox: Locator;
6     private readonly label: Locator;
7
8     constructor(private readonly root: Locator) {
9         this.checkbox = root.getByRole("checkbox");
10        this.label = root.locator("label");
11    }
12
13    async toggle() {
14        await this.checkbox.click();
15    }
16
17    async getText() {
18        return this.label.textContent();
19    }
20}
21
22 // pages/TodoPageWithComponents.ts
```

Fluent Interface Pattern

```
1 import { Page } from "@playwright/test";
2
3 export class LoginPage {
4     constructor(private readonly page: Page) {}
5
6     async goto() {
7         await this.page.goto("/login");
8         return this; // Rückgabe der Instanz für Verkettung
9     }
10
11    async fillEmail(email: string) {
12        await this.page.getByLabel("Email").fill(email);
13        return this; // Rückgabe der Instanz für Verkettung
14    }
15
16    async fillPassword(password: string) {
17        await this.page.getByLabel("Passwort").fill(password);
18        return this; // Rückgabe der Instanz für Verkettung
19    }
20
21    async clickLogin() {
22        await this.page.getByRole("button", { name: "Anmelden" }).click();
```

Übung 9 – Erweiterte Page Objects (BONUS)

Ziel: POM mit Komponenten-Objekten & Fluent Interface erweitern.

Hinweis: Dies ist eine optionale Bonusübung für Fortgeschrittene.

Aufgaben:

1. NewsItemComponent erstellen für einzelne News-Items.
2. NewsPage anpassen für Komponenten-Verwendung.
3. Fluent Interface (return this) implementieren.
4. Tests mit verketteten Aktionen refaktorieren.

Zeit: 30 Minuten (optional)

Nachbesprechung: Advanced POM

Diskussion der Übungsergebnisse

Warum API Mocking?

- Kontrolle über Testumgebung
- Abkopplung von externen Diensten
- Testdaten gezielt manipulieren
- Edge Cases und Fehlerzustände simulieren
- Schnellere Tests ohne Netzwerkverzögerungen
- Stabilere Tests unabhängig von externen Diensten

Arten von Netzwerk-Mocking

Request Mocking

- Anfragen abfangen und modifizieren
- Anfragen blockieren
- Anfragen verzögern
- Anfragen umleiten

Response Mocking

- Antworten simulieren
- Statuscodes ändern
- Antwortdaten manipulieren
- Fehler simulieren

Beispiel: Request & Response Mocking

```
1 // Request Mocking
2 await page.route("**/api/data", (route) => {
3     // Anfrage abfangen und modifizieren
4     const url = route.request().url();
5     const newUrl = url.replace("api/data", "api/mock-data");
6     route.continue({ url: newUrl });
7 });
8
9 // Response Mocking
10 await page.route("**/api/users", (route) => {
11     // Antwort simulieren
12     route.fulfill({
13         status: 200,
14         contentType: "application/json",
15         body: JSON.stringify([{ id: 1, name: "Test User" }]),
16     });
17 });
```

Wann sollte man API Mocking einsetzen?

- Für Tests mit externen Abhängigkeiten:
 - Drittanbieter-APIs
 - Zahlungs-Gateways
 - Authentifizierungsdienste
- Für Tests mit nicht-deterministischen Daten:
 - Zeitabhängige Daten
 - Zufällige Daten
 - Daten von anderen Benutzern

Route-Matching-Strategien: Exakte URL

```
1 // Exakte URL
2 await page.route("https://example.com/api/users", (route) => {
3     // Nur exakte URL wird abgefangen
4});
```

Route-Matching-Strategien: Wildcards

```
1 // URL-Muster mit Wildcards
2 await page.route("**/api/users", (route) => {
3     // Alle URLs, die mit /api/users enden
4});
```

Route-Matching-Strategien: Regex

```
1 // Regex-Muster
2 await page.route(/\/api\/users\/\d+/, (route) => {
3   // URLs wie /api/users/123, /api/users/456, etc.
4 });

});
```

Route-Matching-Strategien: HTTP-Methode

```
1 // Nach HTTP-Methode filtern
2 await page.route("**/api/users", (route) => {
3     if (route.request().method() === "POST") {
4         // Nur POST-Anfragen abfangen
5     } else {
6         // Andere Anfragen normal durchlassen
7         route.continue();
8     }
9});
```

Route-Matching-Strategien: Ressourcentyp

```
1 // Nach Ressourcentyp filtern
2 await page.route("**/*", (route) => {
3     if (route.request().resourceType() === "image") {
4         // Alle Bildanfragen abfangen
5         route.abort();
6     } else {
7         route.continue();
8     }
9});
```

page.route() – Fehlerfall simulieren

```
1  test("Zeigt Fehlermeldung bei API-Fehler", async ({ page }) => {
2      // API-Fehler simulieren
3      await page.route("**/api/data", (route) =>
4          route.fulfill({
5              status: 500,
6              contentType: "application/json",
7              body: JSON.stringify({ error: "Server Error" }),
8          }),
9      );
10
11     await page.goto("/dashboard");
12
13     // Prüfen, ob Fehlermeldung angezeigt wird
14     await expect(page.getByTestId("error-message")).toBeVisible();
15     await expect(page.getByTestId("error-message")).toContainText("Server Error");
16 });

});
```

page.route() – Ladezustand testen

```
1  test("Zeigt Ladezustand während API-Anfrage", async ({ page }) => {
2      // Verzögerte API-Antwort
3      await page.route("**/api/slowdata", async (route) => {
4          await new Promise((r) => setTimeout(r, 1000));
5          await route.fulfill({ status: 200, body: '{"data":"success"}' });
6      });
7
8      await page.goto("/slowpage");
9
10     // Prüfen, ob Ladeindikator angezeigt wird
11     await expect(page.getByTestId("loading")).toBeVisible();
12 });
```

Übung 10 – API Mocking

Ziel: Netzwerkanfragen mocken für unabhängige Tests & Szenarien-Simulation.

Aufgaben:

1. `page.route('**/api/news/public', ...)` verwenden, um API-Antworten zu mocken.
2. Erfolgsfall mit Mock-Daten simulieren und prüfen.
3. Fehlerfall (Status 500) simulieren und Fehler-UI prüfen.
4. Leere Daten (`items: []`) simulieren und UI prüfen.
5. (Optional) Langsame Antwort simulieren und Ladezustand prüfen.

Zeit: 30 Minuten

Nachbesprechung: API Mocking

Diskussion der Übungsergebnisse

Netzwerkanfragen mit waitForResponse abfangen

- **Problem:** Wie können wir auf bestimmte Netzwerkanfragen warten und sie auswerten?
- **Lösung:** `page.waitForResponse()` - Wartet auf eine Netzwerkantwort, die einem Muster entspricht
- **Anwendungsfälle:**
 - API-Antworten validieren
 - Daten aus Antworten extrahieren
 - Asynchrone Operationen synchronisieren

Grundlegende Verwendung von waitForResponse

```
1 // Auf eine bestimmte Anfrage warten
2 test('API-Antwort abfangen', async ({ page }) => {
3     // Navigation und Anfrage gleichzeitig ausführen
4     const responsePromise = page.waitForResponse('**/api/news/**');
5     await page.goto('/news');
6
7     // Auf die Antwort warten
8     const response = await responsePromise;
9
10    // Antwort validieren
11    expect(response.status()).toBe(200);
12    const data = await response.json();
13    expect(data.items.length).toBeGreaterThan(0);
14});
```

Auf Netzwerkanfragen während Interaktionen warten

```
1  test('Auf API-Antwort nach Benutzeraktion warten', async ({ page }) => {
2    await page.goto('/dashboard');
3
4    // Auf Antwort warten, während Button geklickt wird
5    const responsePromise = page.waitForResponse('**/api/refresh-data');
6    await page.getByRole('button', { name: 'Daten aktualisieren' }).click();
7
8    // Antwort abwarten und prüfen
9    const response = await responsePromise;
10   expect(response.ok()).toBeTruthy();
11
12   // UI-Updates prüfen, die von der API-Antwort abhängen
13   const data = await response.json();
14   await expect(page.getText(`Letzte Aktualisierung: ${data.timestamp}`)).toBeVisible();
15});
```

waitForResponse mit Prädikaten für präzise Filterung

```
1  test('Auf spezifische API-Antworten warten', async ({ page }) => {
2    await page.goto('/products');
3
4    // Auf POST-Anfrage mit bestimmtem Body warten
5    const responsePromise = page.waitForResponse(response => {
6      return response.url().includes('/api/filter') &&
7        response.request().method() === 'POST';
8    });
9
10   // Filter anwenden
11   await page.getByLabel('Kategorie').selectOption('elektronik');
12   await page.getRole('button', { name: 'Filtern' }).click();
13
14   // Antwort verarbeiten
15   const response = await responsePromise;
16   const data = await response.json();
17
18   // Daten validieren
19   expect(data.items.every(item => item.category === 'elektronik')).toBeTruthy();
```

waitForResponse kombiniert mit Mocking

```
1  test('API Mocking mit Response-Validierung', async ({ page }) => {
2      // API-Antwort mocken
3      await page.route('**/api/search', route => {
4          route.fulfill({
5              status: 200,
6              contentType: 'application/json',
7              body: JSON.stringify({
8                  results: [
9                      { id: 1, name: 'Gemocktes Produkt' }
10                 ]
11             })
12         });
13     });
14
15     // Auf die gemockte Antwort warten
16     const responsePromise = page.waitForResponse('**/api/search');
17     await page.goto('/search?q=test');
18     const response = await responsePromise;
19 }
```


Playwright Request API für direktes API Testing

- Playwright bietet nicht nur Browser-Automatisierung, sondern auch direkte **API-Test-Funktionalitäten**
- Die `request` Fixture ermöglicht HTTP-Anfragen ohne Browser-Kontext
- Ideal für:
 - Reine API-Tests
 - API-Prüfungen vor/nach UI-Tests
 - Testdaten-Setup via API
 - Backend-Validierung
- Unterstützt alle HTTP-Methoden: GET, POST, PUT, DELETE, etc.

Grundlegende API-Anfragen mit Request

```
1 import { test, expect } from '@playwright/test';
2
3 test('Basic API testing', async ({ request }) => {
4     // GET-Anfrage
5     const getResponse = await request.get('https://jsonplaceholder.typicode.com/posts/1');
6
7     // Validierung
8     expect(getResponse.ok()).toBeTruthy();
9     expect(getResponse.status()).toBe(200);
10
11    const responseBody = await getResponse.json();
12    expect(responseBody.id).toBe(1);
13    expect(responseBody.title).toBeTruthy();
14}):
```

- **request.get()**: Führt eine GET-Anfrage durch
- **response.ok()**: Prüft, ob Status im Bereich 200-299 liegt
- **response.status()**: Gibt den HTTP-Statuscode zurück
- **response.json()**: Parst die Antwort als JSON

Request-Methoden mit JSONPlaceholder

```
1  test('CRUD operations with API', async ({ request }) => {
2    // Create: POST-Anfrage
3    const createResponse = await request.post('https://jsonplaceholder.typicode.com/posts', {
4      data: {
5        title: 'foo',
6        body: 'bar',
7        userId: 1
8      }
9    });
10   expect(createResponse.ok()).toBeTruthy();
11   const newPost = await createResponse.json();
12   expect(newPost.id).toBeTruthy();
13
14   // Read: GET-Anfrage
15   const getResponse = await request.get(`https://jsonplaceholder.typicode.com/posts/${newPost.id}`);
16   expect(getResponse.ok()).toBeTruthy();
17
18   // Update: PUT-Anfrage
19   const updateResponse = await request.put(`https://jsonplaceholder.typicode.com/posts/${newPost.id}`, {
```

Erweiterte Request-Optionen

```
1  test('API request with advanced options', async ({ request }) => {
2      // Anfrage mit verschiedenen Optionen
3      const response = await request.get('https://jsonplaceholder.typicode.com/posts', {
4          // URL-Parameter
5          params: {
6              userId: 1,
7              _limit: 5
8          },
9
10         // Request-Header
11         headers: {
12             'Accept': 'application/json',
13             'Custom-Header': 'test-value'
14         },
15
16         // Timeout setzen (in Millisekunden)
17         timeout: 30000,
18
19         // Redirects nicht folgen
```

Request Fixtures konfigurieren

```
1 // playwright.config.ts
2 import { defineConfig } from '@playwright/test';
3
4 export default defineConfig({
5   use: {
6     // Basis-URL für alle Anfragen
7     baseURL: 'https://jsonplaceholder.typicode.com',
8
9     // Zusätzliche HTTP-Header für alle Anfragen
10    extraHTTPHeaders: {
11      'Accept': 'application/json',
12      'Authorization': 'Bearer test-token',
13    },
14  },
15});
16
17 // Tm Test
```

API-Schema-Validierung mit JSONSchema

```
1 import { test, expect } from '@playwright/test';
2 import Ajv from 'ajv';
3
4 test('Validate API response against JSON Schema', async ({ request }) => {
5   const response = await request.get('https://jsonplaceholder.typicode.com/posts/1');
6   expect(response.ok()).toBeTruthy();
7
8   const responseBody = await response.json();
9
10  // JSON Schema für einen Post
11  const postSchema = {
12    type: 'object',
13    required: ['id', 'title', 'body', 'userId'],
14    properties: {
15      id: { type: 'integer' },
16      title: { type: 'string' },
17      body: { type: 'string' },
18      userId: { type: 'integer' }
19    }
20 }
```

Kombiniertes API & UI Testing

- Besonders effektiver Ansatz: API- und UI-Tests **kombinieren**
- Backend via API testen, Frontend über UI testen
- Schnelleres **Testdaten-Setup** durch API-Aufrufe
- **Vorvalidierung**: API-Status prüfen, bevor UI-Tests starten
- **Vollständige Testabdeckung**: Backend UND Frontend

Beispiel: Kombiniertes API & UI Testing

```
1 import { test, expect } from '@playwright/test';
2
3 test('Kombinierter API und UI-Test', async ({ page, request }) => {
4     // 1. API-Anfrage: Daten abrufen
5     const response = await request.get('https://jsonplaceholder.typicode.com/posts?_limit=5');
6     expect(response.ok()).toBeTruthy();
7     const posts = await response.json();
8
9     // 2. Überprüfen, ob Daten vorhanden sind
10    expect(posts.length).toBe(5);
11    const firstPostTitle = posts[0].title;
12
13    // 3. UI-Test: Seite aufrufen und Daten validieren
14    await page.goto('https://jsonplaceholder.typicode.com/');
15
16    // 4. Mit der API einen neuen Post erstellen
17    const createResponse = await request.post('https://jsonplaceholder.typicode.com/posts', {
18        data: {
19            title: 'Neuer Testbeitrag',
```

Testdaten über API vorbereiten

```
1 import { test, expect } from '@playwright/test';
2
3 // Test Setup mit API-Anfragen
4 test.beforeEach(async ({ request }) => {
5     // Testdaten vorbereiten
6     const createResponse = await request.post('https://jsonplaceholder.typicode.com/posts', {
7         data: {
8             title: 'Testpost für UI-Test',
9             body: 'Dieser Post wird für UI-Tests verwendet',
10            userId: 1
11        }
12    });
13
14    const post = await createResponse.json();
15
16    // ID des erstellten Posts in den Test-Daten speichern
17    test.info().annotations.push({
18        type: 'testPost',
19        description: JSON.stringify(post)
```

API-Fixtures für Tests definieren

```
1 import { test as base, expect } from '@playwright/test';
2
3 // Custom Fixture mit API-Aufrufen
4 type TestFixtures = {
5   testPosts: Array<{ id: number; title: string; }>;
6   testUser: { id: number; name: string; };
7 };
8
9 export const test = base.extend<TestFixtures>({
10   // Fixture: Array von Testposts
11   testPosts: async ({ request }, use) => {
12     const response = await request.get('https://jsonplaceholder.typicode.com/posts?_limit=3');
13     const posts = await response.json();
14     await use(posts);
15   },
16
17   // Fixture: Ein Testbenutzer
18   testUser: async ({ request }, use) => {
19     const response = await request.get('https://jsonplaceholder.typicode.com/users/1');
```

API Response Validierung: Status Code

- **Warum wichtig?** Sicherstellen, dass Ihre Anwendung korrekt auf verschiedene HTTP-Statuscodes reagiert (z.B. 200 OK, 201 Created, 404 Not Found, 500 Server Error).
- **Beim Mocking:** Sie kontrollieren den Status über `route.fulfill({ status: ... })`. Der Test validiert dann das UI-Verhalten basierend auf dem gemockten Status.
- **Bei direkten API-Tests:** Mit Playwright's `request` Fixture können Sie den Status einer echten oder gemockten API-Antwort explizit prüfen.

API Response Validierung: Status Code

```
1 // Beispiel: UI-Test mit gemocktem Fehlerstatus
2 test("UI reagiert auf Serverfehler (500)", async ({ page }) => {
3   await page.route("**/api/data", (route) =>
4     route.fulfill({ status: 500, body: "Internal Server Error" })
5   );
6   await page.goto("/data-display-page"); // Annahme einer Seite, die Daten anzeigt
7   await expect(page.getByText("Ein Fehler ist aufgetreten")).toBeVisible();
8 });
9
10 // Beispiel: Direkter API-Test mit Playwrights request fixture
11 test("API Endpunkt liefert korrekten Statuscode", async ({ request }) => {
12   const response = await request.get("/api/health-check"); // Ein Beispiel-Endpunkt
13
14   // Überprüft, ob der Statuscode im Bereich 200-299 liegt
15   expect(response.ok()).toBeTruthy();
16
17   // Überprüft einen spezifischen Statuscode
18   expect(response.status()).toBe(200);
19 });
```

API Response Validierung: Response Body

- **Warum wichtig?** Überprüfen, ob die API die erwarteten Daten im korrekten Format und Inhalt liefert.
- **Beim Mocking:** Sie definieren den Body mit `route.fulfill({ body: JSON.stringify(...), contentType: "application/json" })`. Der Test validiert, wie die UI diese Daten darstellt.
- **Bei direkten API-Tests:** Die `request` Fixture ermöglicht die detaillierte Überprüfung des Antwort-Bodys (z.B. JSON-Objekte, Textinhalte).

API Response Validierung: Response Body

```
1 // Beispiel: UI-Test mit gemockten JSON-Daten
2 test("UI zeigt gemockte Produktdaten korrekt an", async ({ page }) => {
3   const mockProduct = { id: "prod123", name: "Super Gadget", price: 29.99 };
4   await page.route("**/api/products/prod123", (route) =>
5     route.fulfill({
6       status: 200,
7       contentType: 'application/json',
8       body: JSON.stringify(mockProduct)
9     })
10 );
11 await page.goto("/products/prod123"); // Annahme einer Produkt-Detailseite
12
13 await expect(page.getByRole("heading", { name: "Super Gadget" })).toBeVisible();
14 await expect(page.getText(`Preis: ${mockProduct.price} €`)).toBeVisible();
15 );
16
17 // Beispiel: Direkter API-Test zum Prüfen des Response Body
18 test("API Endpunkt liefert korrekte Benutzerdaten", async ({ request }) => {
19   const response = await request.get("/api/users/current");
```


Zeit-basiertes Testing

Clock API – Zeit unter Kontrolle

- Herausforderung: Tests mit zeitabhängiger Funktionalität sind schwierig
 - Timeouts, Delays, Ablaufdaten
 - Session-Timeouts, Countdown-Timer
 - Zeitstempel, Datumsformatierung
- Problem mit echtem Warten:
 - Tests werden langsam
 - Nicht deterministisch
 - Schwer zu testen (z.B. "in 24 Stunden")
- Lösung: Playwright Clock API
 - Kontrolliere `Date.now()` , `setTimeout()` , `setInterval()`
 - Installiere, pausiere, beschleunige Zeit
 - Deterministisch und schnell

Clock API – Grundkonzept

```
1 import { test, expect } from '@playwright/test';
2
3 test('Zeit-basierte Funktionalität testen', async ({ page }) => {
4     // 1. Clock installieren mit festem Zeitpunkt
5     await page.clock.install({ time: new Date('2024-02-02T08:00:00') });
6
7     // 2. App normal laden (Zeit läuft ab jetzt kontrolliert)
8     await page.goto('http://localhost:3000');
9
10    // 3. Zeit pausieren an bestimmtem Punkt
11    await page.clock.pauseAt(new Date('2024-02-02T10:00:00'));
12
13    // 4. UI-Zustand prüfen
14    await expect(page.getByTestId('current-time')).toHaveText('10:00:00');
15
16    // 5. Zeit vorrspulen
17    await page.clock.fastForward('30:00'); // 30 Minuten
18    await expect(page.getByTestId('current-time')).toHaveText('10:30:00');
19});
```

Clock API – install()

```
1 // Zeit zu Testbeginn festlegen
2 test('App mit spezifischer Startzeit', async ({ page }) => {
3     // Installiere Clock bevor die Seite lädt
4     await page.clock.install({
5         time: new Date('2024-12-24T18:00:00')
6     });
7
8     await page.goto('/');
9
10    // Prüfe, dass die App die richtige Zeit anzeigt
11    await expect(page.getByTestId('date-display'))
12        .toHaveText('24. Dezember 2024, 18:00 Uhr');
13});
```

Wichtig: `install()` muss vor `page.goto()` aufgerufen werden, damit die Zeit von Anfang an kontrolliert wird.

Clock API – pauseAt()

```
1 // Zeit an einem bestimmten Punkt pausieren
2 test('Session-Timeout Warnung', async ({ page }) => {
3     // Starte um 09:00 Uhr
4     await page.clock.install({ time: new Date('2024-02-02T09:00:00') });
5     await page.goto('/app');
6
7     // Benutzer ist aktiv, keine Warnung
8     await expect(page.getByTestId('timeout-warning')).toBeHidden();
9
10    // Pausiere Zeit bei 09:30 (nach 30 Min Inaktivität)
11    await page.clock.pauseAt(new Date('2024-02-02T09:30:00'));
12
13    // Session-Timeout-Warnung sollte erscheinen
14    await expect(page.getByTestId('timeout-warning')).toBeVisible();
15    await expect(page.getByTestId('timeout-warning'))
16        .toHaveText('Ihre Sitzung läuft in 5 Minuten ab');
17});
```

Clock API – fastForward()

```
1 // Zeit schnell vorrspulen
2 test('Countdown Timer', async ({ page }) => {
3   await page.clock.install({ time: new Date('2024-02-02T12:00:00') });
4   await page.goto('/countdown');
5
6   // Starte 10-Minuten-Timer
7   await page.getByRole('button', { name: 'Start Timer' }).click();
8   await expect(page.getByTestId('timer')).toHaveText('10:00');
9
10  // Spule 5 Minuten vor
11  await page.clock.fastForward('05:00');
12  await expect(page.getByTestId('timer')).toHaveText('05:00');
13
14  // Spule weitere 5 Minuten vor (Timer abgelaufen)
15  await page.clock.fastForward('05:00');
16  await expect(page.getByTestId('timer')).toHaveText('00:00');
17  await expect(page.getText('Timer abgelaufen!')).toBeVisible();
18});
```

Clock API – Praktisches Beispiel: Session Management

```
1  test('Auto-Logout nach Inaktivität', async ({ page }) => {
2    // Starte um 14:00 Uhr
3    await page.clock.install({ time: new Date('2024-02-02T14:00:00') });
4    await page.goto('/app');
5
6    // Login
7    await page.getByLabel('Email').fill('user@test.com');
8    await page.getByLabel('Passwort').fill('password');
9    await page.getRole('button', { name: 'Login' }).click();
10   await expect(page.getText('Willkommen zurück')).toBeVisible();
11
12   // Simulierte 15 Minuten Inaktivität (Session-Timeout ist 15 Min)
13   await page.clock.fastForward('15:00');
14
15   // Versuche eine Aktion (sollte zum Login umleiten)
16   await page.getRole('link', { name: 'Profil' }).click();
17   await expect(page).toHaveURL(/.*login/);
18   await expect(page.getText('Ihre Sitzung ist abgelaufen')).toBeVisible();
19});
```

Clock API – Komplexes Szenario: Laptop Sleep/Wake

```
1  test('App Verhalten nach Laptop Sleep', async ({ page }) => {
2    // Starte am Morgen
3    await page.clock.install({ time: new Date('2024-02-02T08:00:00') });
4    await page.goto('/dashboard');
5
6    // Prüfe initiale Zeitanzeige
7    await expect(page.getByTestId('last-sync')).toHaveText('Zuletzt synchronisiert: 08:00 Uhr');
8
9    // Simuliere: Benutzer schließt Laptop-Deckel um 10:00 Uhr
10   await page.clock.pauseAt(new Date('2024-02-02T10:00:00'));
11
12   // Simuliere: Benutzer öffnet Laptop um 14:30 Uhr (4,5 Stunden später)
13   await page.clock.pauseAt(new Date('2024-02-02T14:30:00'));
14
15   // App sollte erkennen, dass Zeit versprungen ist und neu synchronisieren
16   await expect(page.getByTestId('sync-notification')).toBeVisible();
17   await expect(page.getByTestId('sync-notification'))
18     .toHaveText('Daten werden synchronisiert ... ');
19});
```

Clock API – Wann sollte man sie nutzen?

Ideale Anwendungsfälle

- **Session-Timeouts:** Auto-Logout nach Inaktivität
- **Countdown-Timer:** Abgelaufene Timer, Ablaufdaten
- **Scheduled Tasks:** Geplante Aufgaben, Cron-Jobs
- **Time-based UI:** "Vor 5 Minuten", "Gestern", relative Zeitangaben
- **Rate Limiting:** Zeitbasierte Beschränkungen
- **Zeitstempel-Validierung:** Token-Ablauf, Cache-Validierung

Nicht geeignet für

- **Externe Zeitquellen:** Wenn die App Zeit von Server abruft
- **Real-time Kommunikation:** WebSockets mit Server-Zeitstempeln
- **Browser-Performance:** Tatsächliche Rendering-Performance messen

Clock API – Best Practices

1. Installiere Clock vor page.goto()

```
await page.clock.install({ time: ... }); // ZUERST
await page.goto('/'); // DANACH
```

2. Nutze feste Zeitpunkte für Reproduzierbarkeit

```
// ✅ Gut: Deterministisch
await page.clock.install({ time: new Date('2024-01-01T12:00:00') });

// ❌ Schlecht: Abhängig von Testausführungszeit
await page.clock.install({ time: new Date() });
```

3. Verwende sinnvolle Zeit-Sprünge

```
// ✅ Gut: Lesbar
await page.clock.fastForward('02:30:00'); // 2 Std 30 Min

// ❌ Schlechter: Schwer zu lesen
await page.clock.fastForward('9000000'); // Millisekunden
```

Clock API – Debugging-Tipps

```
1  test('Debugging mit Clock API', async ({ page }) => {
2    await page.clock.install({ time: new Date('2024-02-02T10:00:00') });
3    await page.goto('/');
4
5    // Zeit im Test loggen
6    const currentTime = await page.evaluate(() => new Date().toISOString());
7    console.log('Aktuelle Browser-Zeit:', currentTime);
8    // Output: 2024-02-02T10:00:00.000Z
9
10   // Zeit vor spulen
11   await page.clock.fastForward('01:30:00');
12
13   const newTime = await page.evaluate(() => new Date().toISOString());
14   console.log('Nach fastForward:', newTime);
15   // Output: 2024-02-02T11:30:00.000Z
16 });

});
```

Tipp

Bei Debugging kannst du mit `page.evaluate(() => Date.now())` die aktuelle Mock-Zeit auslesen.

Übung 13 – Clock API Testing mit ClockTab.com

Ziel: Die Clock API lernen und an einer echten Website testen.

Website: <http://localhost:3000/clock>

Aufgaben:

1. Clock installieren:

- `page.clock.install()` vor Navigation verwenden
- Spezifische Startzeit setzen
- Zeit auf Website verifizieren

2. Zeit vorspulen:

- `page.clock.fastForward()` nutzen
- Zeitänderungen in der Anzeige prüfen
- Verschiedene Zeitsprünge testen

3. Zeit kontrollieren:

- `page.clock.pauseAt()` zum Pausieren

Nachbesprechung: Clock API

Diskussion der Übungsergebnisse

layout: center

Testing-Strategien: Datenmanagement

Warum ist Testdaten-Management wichtig?

- **Konsistenz:** Jeder Testlauf sollte unter gleichen Bedingungen starten.
- **Isolation:** Tests sollten sich nicht gegenseitig beeinflussen.
- **Wiederholbarkeit:** Tests müssen zuverlässig reproduzierbar sein.
- **Performance:** Setup und Teardown sollten effizient sein.
- **Realismus:** Testdaten sollten realen Szenarien ähneln.

Strategien zur Testdaten-Generierung

1. Statisches Seeding

- Feste Daten in DB laden (vor Testsuite)
- Einfach, aber unflexibel
- Gefahr von Abhängigkeiten

2. Data Factories

- Code zum Erstellen von Daten (z.B. mit Faker.js)
- Flexibel, anpassbar
- Erfordert Wartung

3. API-basiertes Setup

- Daten über App-API erstellen
- Realistisch, testet API mit
- Langsamer als DB-Zugriff

Datenbank-Reset-Strategien

Vor/Nach jedem Test

- `beforeEach / afterEach` : DB leeren/neu
befüllen
- **Vorteil:** Maximale Isolation
- **Nachteil:** Sehr langsam bei vielen Tests

Datenbank-Transaktionen

- `beforeEach` : Transaktion starten
- `afterEach` : Rollback durchführen
- **Vorteil:** Schnell, gute Isolation
- **Nachteil:** Nicht alle DBs/Setups
unterstützen das gut

Vor/Nach der Testsuite

- `beforeAll / afterAll` : Einmaliges Setup/Cleanup
- **Vorteil:** Schnell
- **Nachteil:** Keine Isolation zwischen Tests, Reihenfolge wichtig

Testdaten-Management mit Fixtures

- **Ideal für Setup/Teardown:** Kapselt Logik zur Datenerstellung und -bereinigung.
- **Wiederverwendbar:** Gleiche Daten-Setups für mehrere Tests.
- **Konfigurierbar:** Fixture-Optionen für Variationen (z.B. User-Rollen).
- **Isolation:** Test-Spaced Fixtures sorgen für saubere Daten pro Test.

```
1 // Beispiel: Fixture für Testbenutzer mit API-Setup/Teardown
2 export const test = base.extend<{
3   testUser: { id: string; email: string };
4 }>({
5   testUser: async ({ request }, use) => {
6     // Setup: Benutzer über API erstellen
7     const email = `test-${Date.now()}@example.com`;
8     const response = await request.post("/api/users", {
9       data: { email, password: "password123" },
10      });
11     const user = await response.json();
12
13     // Fixture-Wert bereitstellen
14     await use(user);
```

Beispiel: Testdaten-Fixture

```
1 // fixtures.ts
2 import { test as base } from "@playwright/test";
3 import { createTestArticle, deleteTestArticle } from "../utils/apiHelpers";
4
5 type TestDataFixtures = {
6   testArticle: { id: string; title: string; authorId: string };
7 };
8
9 export const test = base.extend<TestDataFixtures>({
10   testArticle: async ({ request, testUser }, use) => {
11     // Setup: Artikel für den Testbenutzer erstellen
12     const articleData = {
13       title: "Test Article Title",
14       content: "Test content ...",
15       authorId: testUser.id, // Nutzt andere Fixture!
16     };
17     const article = await createTestArticle(request, articleData);
18
19     await use(article);
20   }
21 }
```

Strategie: Tests mit Tags organisieren

- **Warum Tests taggen?**
 - Bessere Organisation von Testsuiten
 - Gezielte Ausführung nach Testtyp oder Wichtigkeit
 - Zeitersparnis durch Filtern in CI/CD-Pipelines
 - Bessere Dokumentation der Testabdeckung
- **Typische Tag-Kategorien:**
 - Nach **Testtyp**: @api , @ui , @visual , @smoke , @regression
 - Nach **Laufzeit**: @slow , @fast
 - Nach **Bereich**: @auth , @payment , @profile
 - Nach **Priorität**: @critical , @p0 , @p1
 - Nach **Datenbedarf**: @needs-db , @needs-api

Tests mit Tags in Playwright implementieren

- Option 1: Tags im Testtitel

```
test('Benutzer kann einloggen @auth @smoke', async ({ page }) => {
  // Test-Implementierung
});
```

- Option 2: Tags über Konfigurationsobjekt

```
test('API-Antwort enthält korrekte Daten', {
  tag: '@api'
}, async ({ request }) => {
  // Test-Implementierung
});
```

- Option 3: Tags für Testgruppen

```
test.describe('Payment Tests', {
  tag: ['@payment', '@critical']
}, () => {
  test('Zahlung erfolgreich verarbeiten', async ({ page }) => {
    // Test-Implementierung
  });
});
```

Tagbasierte Teststrategie: Beispiele nach Typ

```
1 // 1. API-Tests taggen
2 test('API gibt korrekte Benutzerdaten zurück', {
3   tag: '@api'
4 }, async ({ request }) => {
5   const response = await request.get('/api/users/me');
6   expect(response.ok()).toBeTruthy();
7   const data = await response.json();
8   expect(data).toHaveProperty('id');
9 });
10
11 // 2. Visuelle Tests taggen
12 test('Dashboard zeigt korrekte Komponenten an', {
13   tag: ['@visual', '@ui']
14 }, async ({ page }) => {
15   await page.goto('/dashboard');
16   await expect(page).toHaveScreenshot('dashboard.png');
17 });
18
19 // 3. Performancekritische Tests taggen
```

Tagbasierte Teststrategie: Tests nach Tags ausführen

- Nur bestimmte Tags ausführen:

```
# Nur Smoke-Tests ausführen  
npx playwright test --grep @smoke  
  
# Nur API-Tests ausführen  
npx playwright test --grep @api
```

- Bestimmte Tags ausschließen:

```
# Alle Tests außer langsame Tests ausführen  
npx playwright test --grep-invert @slow
```

- Kombinierte Tag-Filterung:

```
# Nur Tests, die sowohl @critical als auch @ui markiert sind  
npx playwright test --grep "(?=.*@critical)(?=.*@ui)"  
  
# Tests, die entweder @smoke oder @fast sind  
npx playwright test --grep "@smoke|@fast"
```

Tagbasierte Teststrategie in CI/CD-Pipelines

```
1  # Beispiel GitHub Actions Workflow
2  name: Playwright Tests
3  on: push
4  jobs:
5    api-tests:
6      name: API Tests
7      runs-on: ubuntu-latest
8      steps:
9        - uses: actions/checkout@v3
10       - uses: actions/setup-node@v3
11       - name: Install dependencies
12         run: npm ci
13       - name: Install Playwright
14         run: npx playwright install --with-deps
15       - name: Run API tests
16         run: npx playwright test --grep @api
```

Tagbasierte Teststrategie in der Konfiguration

```
1 // playwright.config.ts
2 import { defineConfig, devices } from "@playwright/test";
3
4 export default defineConfig({
5   projects: [
6     {
7       name: 'smoke',
8       grep: /@smoke/,
9       use: { ...devices['Desktop Chrome'] },
10    },
11    {
12      name: 'full-regression',
13      grep: /@regression/,
14      use: {
15        ...devices['Desktop Chrome'],
16        ...devices['Desktop Firefox'],
17        ...devices['Desktop Safari'],
18      },
19    },
20  ],
21});
```

Best Practices für Test-Tagging

- **Konsistente Namenskonvention**
 - Eindeutiges Präfix (`@` wird empfohlen)
 - Klare, aussagekräftige Tag-Namen
 - Dokumentierte Bedeutungen im Projekt
- **Organisatorische Struktur**
 - Kombiniere mehrere Kategorien für präzise Filterung
 - Beispiel: `@ui @auth @critical` für kritische UI-Authentifizierungstests
 - Halte die Tag-Anzahl überschaubar (max. 3-5 pro Test)
- **Dynamische Test-Organisation**
 - Smoke-Tests: `@smoke` für schnelle Verifikation nach Deployments
 - Feature-basierte Tags: `@feature-xyz` für Tests einer neuen Funktion
 - Release-Tags: `@v2.0` für Version-spezifische Tests
- **CI/CD-Integration**

Diskussion: Testing-Strategien

Welche Strategien passen zu eurem Projekt?

Visuelle Regression mit Playwright

- **Was ist visuelle Regression?**
 - Automatischer Vergleich von Screenshots, um unbeabsichtigte UI-Änderungen zu erkennen
 - Ergänzt klassische Assertions um visuelle Prüfungen
- **Typische Anwendungsfälle:**
 - UI-Refactorings, Design-Updates, Regressionstests

Visuelle Vergleiche mit Playwright

- **Screenshot-Assertion:**
 - `await expect(page).toHaveScreenshot()`
 - Erstellt beim ersten Lauf einen Referenz-Screenshot ("golden file")
 - Vergleicht bei Folgeläufen gegen die Referenz
- **Vergleich von Text/Binärdaten:**
 - `expect(value).toMatchSnapshot()`
 - Für Text, JSON, API-Responses, etc.

Beispiel: Screenshot- und Snapshot-Test

```
1 import { test, expect } from '@playwright/test';
2
3 test('Landingpage visuell unverändert', async ({ page }) => {
4     await page.goto('https://playwright.dev');
5     await expect(page).toHaveScreenshot();
6 });
7
8 test('Textinhalt bleibt gleich', async ({ page }) => {
9     await page.goto('https://playwright.dev');
10    const title = await page.textContent('.hero__title');
11    expect(title).toMatchSnapshot('hero.txt');
12});
```

Optionen & Best Practices

- **Toleranzen einstellen:**
 - `maxDiffPixels` , `maxDiffPixelRatio` für erlaubte Unterschiede
 - **Beispiel:** `await expect(page).toHaveScreenshot({ maxDiffPixels: 100 })`
- **Dynamische Inhalte ausblenden:**
 - Mit `stylePath` eigene CSS-Regeln beim Screenshot anwenden
 - **Beispiel:** `await expect(page).toHaveScreenshot({ stylePath: './Screenshot.css' })`
- **Globale Defaults:**
 - In `playwright.config.ts` unter `expect.toHaveScreenshot`

Snapshot-Verwaltung & Stabilität

- Snapshots werden im Ordner `<testfile>-snapshots/` gespeichert
- Snapshots sollten versioniert werden (z.B. mit git)
- Aktualisieren bei UI-Änderungen:
 - Mit `npx playwright test --update-snapshots`
- Stabilität:
 - Immer im gleichen OS, Browser, Headless/Headful-Modus testen
 - Fonts, Rendering, Hardware können Unterschiede verursachen

Übung 12 – Visual Regression Testing

Ziel: Visual Regression Tests für die Feed App implementieren.

Aufgaben:

1. Basis-Screenshots erstellen:

- Homepage und News Feed Screenshots
- Full-Page und Component Screenshots
- Animationen deaktivieren für Konsistenz

2. Dark Mode Visual Testing:

- Light/Dark Mode Toggle testen
- Kontrast-Verhältnisse prüfen

3. Responsive Screenshots:

- Desktop, Tablet, Mobile Viewports
- Cross-Browser Consistency
- Dynamische Inhalte maskieren

Nachbesprechung: Visual Testing

Diskussion der Übungsergebnisse

Warum Mobile Testing?

- **Mobiler Traffic** dominiert das Web
- **Responsive Design** ist Standard
- **Unterschiedliche Geräte** zeigen unterschiedliche Probleme
- **Gerätespezifische Features** müssen getestet werden
- **Performance-Unterschiede** zwischen Desktop und Mobil

Mobile-First Testing

- **Mobile-First Entwicklung** erfordert Mobile-First Testing
- **Kritische Nutzerpfade** auf Mobilgeräten priorisieren
- **Eingeschränkte Bildschirmgröße** beeinflusst UI-Design
- **Touch-basierte Interaktionen** statt Maus und Tastatur
- **Netzwerkbedingungen** simulieren (3G, 4G, instabile Verbindungen)
- **Offline-Funktionalität** testen

Mobile vs. Desktop Unterschiede

Mobile Herausforderungen

- Kleinere Bildschirme
- Touch-Interaktionen
- Virtuelle Tastatur
- Gerätesensoren (GPS, Beschleunigung)
- Batterieverbrauch
- Unterschiedliche Browser-Engines
- Langsamere Prozessoren

Desktop Vorteile

- Größere Bildschirme
- Präzise Maussteuerung
- Physische Tastatur
- Konsistenter Performance
- Stabilere Netzwerkverbindung
- Erweiterte Debugging-Tools
- Mehr Rechenleistung

Live-Demo: Mobile Testing

Praktische Demonstration der Konzepte

Geräte-Emulation in Playwright - Konfiguration

```
1 // In playwright.config.ts
2 import { defineConfig, devices } from "@playwright/test";
3
4 export default defineConfig({
5   projects: [
6     {
7       name: "chromium",
8       use: { ...devices["Desktop Chrome"] },
9     },
10    {
11      name: "iPhone 13",
12      use: { ...devices["iPhone 13"] },
13    },
14    {
15      name: "Galaxy S8",
16      use: { ...devices["Galaxy S8"] },
17    },
18  ],
19});
```

Geräte-Emulation im Test

```
1 // Direkt im Test verwenden
2 test("Test auf iPhone", async ({ browser }) => {
3   const context = await browser.newContext({
4     ...devices["iPhone 13"],
5   });
6   const page = await context.newPage();
7   // Test läuft jetzt als wäre er auf einem iPhone
8 });
```

Benutzerdefinierte Geräte-Emulation

```
1 // Benutzerdefiniertes Gerät erstellen
2 const myCustomPhone = {
3   userAgent:
4     "Mozilla/5.0 (Linux; Android 11; Custom Phone) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.110 Mobile Safari/537.36
5   viewport: {
6     width: 412,
7     height: 915,
8   },
9   deviceScaleFactor: 2.625,
10  isMobile: true,
11  hasTouch: true,
12  defaultBrowserType: "chromium",
13};
```

Benutzerdefinierte Geräte verwenden

```
1 // Im Test verwenden
2 test("Test auf benutzerdefiniertem Gerät", async ({ browser }) => {
3   const context = await browser.newContext({
4     ...myCustomPhone,
5     geolocation: { longitude: 48.8584, latitude: 2.2945 },
6     permissions: ["geolocation"],
7     locale: "de-DE",
8     timezoneId: "Europe/Berlin",
9   });
10  const page = await context.newPage();
11  // Test mit benutzerdefinierten Einstellungen
12});
```

Emulierte Funktionen - Teil 1

```
1 // Erweiterte Geräteeigenschaften emulieren
2 test("Test mit emulierten Funktionen", async ({ browser }) => {
3   const context = await browser.newContext({
4     ...devices["iPhone 13"],
5
6     // Geolocation
7     geolocation: { latitude: 52.520008, longitude: 13.404954 },
8     permissions: ["geolocation"],
9
10    // Sprache und Zeitzone
11    locale: "de-DE",
12    timezoneId: "Europe/Berlin",
13  });
14
15  const page = await context.newPage();
16  await page.goto("https://example.com");
17});
```

Emulierte Funktionen - Teil 2

```
1 // Weitere emulierte Funktionen
2 test("Test mit weiteren emulierten Funktionen", async ({ browser }) => {
3   const context = await browser.newContext({
4     ...devices["iPhone 13"],
5     // Farbschema
6     colorScheme: "dark", // oder 'light'
7     // Offline-Modus
8     offline: false,
9
10    // HTTP-Credentials
11    httpCredentials: {
12      username: "user",
13      password: "pass",
14    },
15
16    // Kamera/Mikrofon-Berechtigungen
17    permissions: ["camera", "microphone"],
18  });
19
```

Responsive Tests schreiben - Desktop & Mobile

```
1 // Responsive Tests mit verschiedenen Viewports
2 test.describe("Responsive Tests", () => {
3   test("Navigation auf Desktop", async ({ page }) => {
4     await page.setViewportSize({ width: 1280, height: 800 });
5     await page.goto("/");
6
7     // Desktop-Navigation prüfen
8     await expect(page.getByTestId("desktop-menu")).toBeVisible();
9     await expect(page.getByTestId("mobile-menu")).toBeHidden();
10  });
11
12  test("Navigation auf Mobilgeräten", async ({ page }) => {
13    await page.setViewportSize({ width: 375, height: 667 });
14    await page.goto("/");
15
16    // Mobile Navigation prüfen
17    await expect(page.getByTestId("mobile-menu")).toBeVisible();
18    await expect(page.getByTestId("desktop-menu")).toBeHidden();
19  });
}
```

Breakpoint-basierte Tests - Definition

```
1 // Breakpoints definieren
2 const breakpoints = {
3   xs: { width: 320, height: 568 }, // iPhone SE
4   sm: { width: 375, height: 667 }, // iPhone 8
5   md: { width: 768, height: 1024 }, // iPad
6   lg: { width: 1024, height: 768 }, // iPad Landscape
7   xl: { width: 1280, height: 800 }, // Desktop
8   xxl: { width: 1920, height: 1080 }, // Large Desktop
9 };
```

Breakpoint-basierte Tests - Implementierung

```
1 // Tests für alle Breakpoints
2 for (const [name, viewport] of Object.entries(breakpoints)) {
3   test(`Navigation auf ${name} Breakpoint`, async ({ page }) => {
4     await page.setViewportSize(viewport);
5     await page.goto("/");
6
7     // Responsive Verhalten prüfen
8     if (viewport.width < 768) {
9       // Mobile Verhalten
10      await expect(page.getByTestId("mobile-menu")).toBeVisible();
11      await expect(page.getByTestId("desktop-menu")).toBeHidden();
12    } else {
13      // Desktop Verhalten
14      await expect(page.getByTestId("desktop-menu")).toBeVisible();
15      await expect(page.getByTestId("mobile-menu")).toBeHidden();
16    }
17  });
18}
```

Touch-Events simulieren - Tap & Swipe

```
1 // Touch-Events testen
2 test("Touch-Interaktionen", async ({ page }) => {
3     // iPhone 13 emulieren
4     await page.setViewportSize({ width: 390, height: 844 });
5     await page.goto("/");
6
7     // Tap (einfacher Touch)
8     await page.getByRole("button", { name: "Menü" }).tap();
9
10    // Swipe (Touch mit Bewegung)
11    const carousel = page.locator(".carousel");
12    const boundingBox = await carousel.boundingBox();
13
14    if (boundingBox) {
15        // Swipe von rechts nach links
16        await page.touchscreen.tap(
17            boundingBox.x + boundingBox.width - 10,
18            boundingBox.y + boundingBox.height / 2,
19        );
20        await page.mouse.down();
21        await page.mouse.move(
22            boundingBox.x + 10,
```


Nachbesprechung: Mobile & Responsive

Diskussion der Übungsergebnisse

Video Recording in Playwright

- Automatische Videoaufzeichnung für jeden Test
- Fehleranalyse: Videos nur bei Fehlern behalten
- Performance-Überlegungen: Videos brauchen Speicher und CPU
- CI/CD Integration: Videos als Artifacts speichern

Video Recording Konfiguration

```
1 // playwright.config.ts
2 export default defineConfig({
3   use: {
4     // Video-Optionen
5     video: 'on', // Immer aufzeichnen
6     // Alternativen:
7     // video: 'off' - Keine Videos
8     // video: 'retain-on-failure' - Nur bei Fehlern behalten
9     // video: 'on-first-retry' - Beim ersten Retry aufzeichnen
10
11    // Video-Größe anpassen
12    video: {
13      mode: 'retain-on-failure',
14      size: { width: 1280, height: 720 }
15    }
16  }
17});
18
19 // Im Test auf Video zugreifen
```

HAR Recording - Network Traffic aufzeichnen

- HAR (HTTP Archive): Standardformat für Netzwerk-Logs
- Alle HTTP(S) Requests und Responses aufzeichnen
- Performance-Analyse: Ladezeiten, Größen, Timings
- Debugging: Fehlgeschlagene Requests identifizieren
- Mock-Daten generieren: HAR als Basis für Mocks

HAR Recording einrichten

```
1 // HAR Recording aktivieren
2 test('HAR Recording', async ({ page, context }) => {
3     // Option 1: Über Context (für alle Seiten)
4     await context.routeFromHAR('./hars/api-calls.har', {
5         url: '**/api/**',
6         update: true // HAR aktualisieren
7     });
8
9     // Option 2: Manuell aufzeichnen
10    await page.routeFromHAR('./hars/session.har', {
11        url: '**/*',
12        update: false // Nur abspielen, nicht aufzeichnen
13    });
14
15    await page.goto('/');
16
17    // HAR für Offline-Tests verwenden
18    await context.routeFromHAR('./hars/cached.har', {
19        url: '**/*',
```

Test Reporter in Playwright

- **Built-in Reporter:**
 - `list` - Einfache Liste (Standard)
 - `line` - Einzelige Updates
 - `dot` - Minimalistisch
 - `json` - Maschinenlesbar
 - `junit` - XML für CI/CD
 - `html` - Interaktiver Report
 - `github` - GitHub Actions Annotations
- **Multiple Reporter** gleichzeitig verwenden
- **Custom Reporter** implementieren
- **Third-Party Reporter** integrieren

Reporter Konfiguration

```
1 // playwright.config.ts
2 export default defineConfig({
3     // Einzelner Reporter
4     reporter: 'html',
5
6     // Multiple Reporter
7     reporter: [
8         ['list'], // Konsolen-Output
9         ['json', { outputFile: 'test-results.json' }],
10        ['html', { open: 'never' }], // HTML Report ohne Auto-Open
11        ['junit', { outputFile: 'junit.xml' }]
12    ],
13
14    // Reporter mit Optionen
15    reporter: [
16        ['html', {
17            outputFolder: 'my-report',
18            open: 'always', // 'never', 'on-failure', 'always'
19            host: 'localhost',
```

Custom Reporter implementieren

```
1 // my-reporter.ts
2 import { Reporter, TestCase, TestResult, FullConfig } from '@playwright/test/reporter';
3
4 class MyReporter implements Reporter {
5   onBegin(config: FullConfig, suite: Suite) {
6     console.log(`Starting test run with ${suite.allTests().length} tests`);
7   }
8
9   onTestEnd(test: TestCase, result: TestResult) {
10    console.log(`Test ${test.title}: ${result.status}`);
11    if (result.status === 'failed') {
12      console.log('Error:', result.error);
13    }
14  }
15
16  onEnd(result: FullResult) {
17    console.log(`Test run finished: ${result.status}`);
18  }
19}
```

Network Throttling & Offline Mode

- Langsame Verbindungen simulieren: 3G, 4G, etc.
- Offline-Modus testen
- Latenz und Bandbreite kontrollieren
- Real-World Conditions nachstellen

Network Throttling implementieren

```
1 // Chrome DevTools Protocol für Network Throttling
2 test('Slow 3G Network', async ({ page, browser }) => {
3   const context = await browser.newContext();
4   const client = await context.newCDPSession(await context.newPage());
5
6   // Network Throttling aktivieren
7   await client.send('Network.enable');
8   await client.send('Network.emulateNetworkConditions', {
9     offline: false,
10    downloadThroughput: (1.6 * 1024 * 1024) / 8, // 1.6 Mbps
11    uploadThroughput: (750 * 1024) / 8, // 750 kbps
12    latency: 150 // 150ms Latenz
13  });
14
15  await page.goto('/');
16  // Test mit langsamer Verbindung
17});
18
19 // Offline Mode
```

Network Profiles

```
1 // Vordefinierte Netzwerkprofile
2 const networkProfiles = {
3     'GPRS': {
4         offline: false,
5         downloadThroughput: 50 * 1024 / 8,
6         uploadThroughput: 20 * 1024 / 8,
7         latency: 500,
8     },
9     'Regular 3G': {
10        offline: false,
11        downloadThroughput: 750 * 1024 / 8,
12        uploadThroughput: 250 * 1024 / 8,
13        latency: 100,
14    },
15    'Fast 3G': {
16        offline: false,
17        downloadThroughput: 1.5 * 1024 * 1024 / 8,
18        uploadThroughput: 750 * 1024 / 8,
19        latency: 40,
```

Spezialthemen

Component Testing mit Playwright

- **Was ist Component Testing?**
 - Testen von UI-Komponenten in Isolation
 - Ohne gesamte App zu laden
 - Schneller als E2E-Tests
 - Fokus auf Komponenten-Verhalten
- **Warum Component Testing?**
 - Schnelles Feedback während Entwicklung
 - Präzise Tests für wiederverwendbare Komponenten
 - Komponenten-Bibliotheken testen
 - Ergänzung zu E2E-Tests
- **Playwright unterstützt:**
 - React, Vue, Svelte, Solid
 - Experimental Feature (`@playwright/experimental-ct-*`)

Component Testing – Setup (React Beispiel)

```
1 # Playwright Component Testing für React installieren  
2 npm init playwright@latest -- --ct
```

Oder manuell:

```
1 npm install --save-dev @playwright/experimental-ct-react
```

Konfiguration: `playwright-ct.config.ts`

```
1 import { defineConfig, devices } from '@playwright/experimental-ct-react';  
2  
3 export default defineConfig({  
4   testDir: './src/components',  
5   use: {  
6     trace: 'on-first-retry',  
7   },  
8   projects: [  
9     { name: 'chromium', use: { ...devices['Desktop Chrome'] } },  
10    ],  
11  });
```

Component Testing – Beispiel: React Button

```
1 // src/components/Button.spec.tsx
2 import { test, expect } from '@playwright/experimental-ct-react';
3 import Button from './Button';
4
5 test('Button zeigt Text an', async ({ mount }) => {
6     // Komponente mounten
7     const component = await mount(<Button>Klick mich</Button>);
8
9     // Assertions wie bei normalem Playwright Test
10    await expect(component).toContainText('Klick mich');
11 });
12
13 test('Button Click Event', async ({ mount }) => {
14     let clicked = false;
15
16     const component = await mount(
17         <Button onClick={() => { clicked = true; }}>
18             Klick mich
19         </Button>
```

Component Testing – Props und State

```
1 // Test mit verschiedenen Props
2 test('Button Varianten', async ({ mount }) => {
3   // Primary Button
4   const primaryBtn = await mount(<Button variant="primary">Primary</Button>);
5   await expect(primaryBtn).toHaveClass(/btn-primary/);
6
7   // Disabled Button
8   const disabledBtn = await mount(<Button disabled>Disabled</Button>);
9   await expect(disabledBtn).toBeDisabled();
10 });
11
12 // Test mit State-Änderungen
13 test('Counter Komponente', async ({ mount }) => {
14   const component = await mount(<Counter initialCount={0} />,
15
16   await expect(component.getByTestId('count')).toHaveText('0');
17
18   await component.getByRole('button', { name: '+' }).click();
19   await expect(component.getByTestId('count')).toHaveText('1');
```

Component Testing – Visual Regression

```
1 // Visuelle Tests für Komponenten
2 test('Button Screenshots', async ({ mount }) => {
3   const component = await mount(
4     <div style={{ padding: '20px' }}>
5       <Button variant="primary">Primary</Button>
6       <Button variant="secondary">Secondary</Button>
7       <Button variant="danger">Danger</Button>
8     </div>
9   );
10
11   // Screenshot der Komponente
12   await expect(component).toHaveScreenshot('buttons.png');
13 });
14
15 // Test mit Theme
16 test.use({ colorScheme: 'dark' });
17
18 test('Button im Dark Mode', async ({ mount }) => {
19   const component = await mount(<Button>Dark Mode</Button>);
```

Component Testing – E2E vs. Component Testing

E2E Testing

- Gesamte Anwendung
- Reale User-Flows
- Langsamer (lädt App)
- Integration zwischen Komponenten
- Authentifizierung, Routing
- Kritische User-Journeys

Component Testing

- Isolierte Komponenten
- Komponenten-Verhalten
- Schneller (nur Komponente)
- Einzelne Komponenten
- Props, Events, State
- Wiederverwendbare UI-Elemente

Empfehlung: Kombiniere beide Ansätze für optimale Testabdeckung!

Component Testing – Wann nutzen?

Ideale Anwendungsfälle

- **UI-Bibliotheken:** Design Systems, Komponenten-Bibliotheken
- **Wiederverwendbare Komponenten:** Buttons, Forms, Modals
- **Komplexe UI-Logik:** State-Management in Komponenten
- **Verschiedene Varianten:** Props, Themes, States testen
- **Visual Regression:** Aussehen von Komponenten sicherstellen

Weniger geeignet

- **Integration mehrerer Seiten:** Besser als E2E-Test
- **Backend-Integration:** APIs mocken wird komplex
- **Routing-Tests:** Navigation zwischen Seiten

Component Testing – Ressourcen

Dokumentation:

- playwright.dev/docs/test-components

Unterstützte Frameworks:

- `@playwright/experimental-ct-react` - React
- `@playwright/experimental-ct-vue` - Vue
- `@playwright/experimental-ct-svelte` - Svelte
- `@playwright/experimental-ct-solid` - Solid

Status:

- Derzeit experimental
- Aktiv in Entwicklung
- Produktionsreif für viele Use Cases

Parameterized Testing – Einführung

- Wiederholende Tests mit unterschiedlichen Daten durchführen
- DRY-Prinzip: Don't Repeat Yourself – Testcode reduzieren
- Datengetrieben: Tests mit verschiedenen Eingaben und erwarteten Ergebnissen
- Flexibel: Nutzbar sowohl auf Test- als auch Projektebene
- Leistungsfähig: Kombinierbar mit Fixtures und Hooks

Parameterized Testing – Anwendungsfälle

- **Formularvalidierung** mit verschiedenen Eingabewerten
- **API-Endpunkte** mit unterschiedlichen Parametern
- **Responsive Design** für verschiedene Gerätetypen
- **Mehrsprachige UIs** mit verschiedenen Locales
- **Berechtigungssysteme** mit verschiedenen Benutzerrollen

Grundlegende Test-Parameterisierung

```
1 // Daten-Array definieren und testen
2 [
3   { name: 'Alice', expected: 'Hallo, Alice!' },
4   { name: 'Bob', expected: 'Hallo, Bob!' },
5   { name: 'Charlie', expected: 'Hallo, Charlie!' },
6 ].forEach(({ name, expected }) => {
7   test(`Begrüßung für ${name}`, async ({ page }) => {
8     await page.goto(`https://example.com/greet?name=${name}`);
9     await expect(page.getByRole('heading')).toHaveText(expected);
10  });
11});
```

- **Array von Testfällen definiert**
- **forEach-Schleife** generiert Tests dynamisch
- Einzigartiger **Testname** durch Template-String
- **Gleiche Testlogik** für alle Testfälle

Test-Hooks mit Parameterisierung

```
1 // Hooks außerhalb der forEach-Schleife: werden nur einmal ausgeführt
2 test.beforeEach(async ({ page }) => {
3   await page.goto('https://example.com');
4 );
5
6 // Testfälle
7 [
8   { name: 'Admin', permissions: ['create', 'read', 'update', 'delete'] },
9   { name: 'Editor', permissions: ['create', 'read', 'update'] },
10  { name: 'Viewer', permissions: ['read'] },
11 ].forEach(({ name, permissions }) => {
12   test(`Benutzerrolle ${name} hat korrekte Berechtigungen`, async ({ page }) => {
13     // Login als Benutzer mit dieser Rolle
14     await page.getLabel('Benutzername').fill(name.toLowerCase());
15     await page.getLabel('Passwort').fill('password123');
16     await page.getRole('button', { name: 'Login' }).click();
17
18     // Prüfen, ob UI korrekte Berechtigungen anzeigt
19     for (const permission of permissions) {
```

Test-Hooks pro Testfall

```
1 // Hooks innerhalb eines describe-Blocks: werden für jeden Testfall ausgeführt
2 [
3   { name: 'Admin', role: 'admin', expectedPages: ['Dashboard', 'Benutzer', 'Einstellungen'] },
4   { name: 'Editor', role: 'editor', expectedPages: ['Dashboard', 'Inhalte'] },
5   { name: 'Viewer', role: 'viewer', expectedPages: ['Dashboard'] },
6 ].forEach(({ name, role, expectedPages }) => {
7   test.describe(`Benutzerrolle: ${name}`, () => {
8     test.beforeEach(async ({ page }) => {
9       await page.goto('https://example.com/login');
10      await page.getLabel('Benutzername').fill(role);
11      await page.getLabel('Passwort').fill('password123');
12      await page.getRole('button', { name: 'Login' }).click();
13    });
14
15    test('Kann auf erlaubte Seiten zugreifen', async ({ page }) => {
16      for (const pageName of expectedPages) {
17        await page.getRole('link', { name: pageName }).click();
18        await expect(page.getRole('heading')).toContainText(pageName);
19      }
20    });
21  });
22});
```

Projektebene Parameterisierung – Definition

```
1 // my-test.ts - Benutzerdefinierte Test-Erweiterung
2 import { test as base } from '@playwright/test';
3
4 export type UserRole = {
5     role: string; // Parameter, der auf Projektebene konfiguriert wird
6 };
7
8 export const test = base.extend<UserRole>({
9     // Default-Wert ('user') mit Option-Flag
10    role: ['user', { option: true }],
11});
```

- Typ-Definition für den Parameter (TypeScript)
- **test.extend** erweitert die Test-Fixture
- Default-Wert mit Flag
- Parameter kann in der Konfiguration überschrieben werden

Projektebene Parameterisierung – Verwendung

```
1 // playwright.config.ts
2 import { defineConfig } from '@playwright/test';
3 import type { UserRole } from './my-test';
4
5 export default defineConfig<UserRole>({
6   projects: [
7     {
8       name: 'admin-tests',
9       use: { role: 'admin' },
10    },
11    {
12      name: 'editor-tests',
13      use: { role: 'editor' },
14    },
15    {
16      name: 'viewer-tests',
17      use: { role: 'viewer' },
18    },
19  ]
```

- Mehrere **Projekt-Konfigurationen** definieren
- Jedes Projekt erhält einen anderen **Parameterwert**
- Tests laufen mit **jeder Konfiguration** durch
- Parameterwerte über **use-Objekt** gesetzt

Parameter in Fixtures verwenden

```
1 // my-test.ts
2 import { test as base } from '@playwright/test';
3
4 export type UserRole = {
5   role: string;
6   user: { name: string; email: string; isAdmin: boolean };
7 };
8
9 export const test = base.extend<UserRole>({
10   // Parameter mit Default-Wert
11   role: ['user', { option: true }],
12
13   // Fixture, die den Parameter verwendet
14   user: async ({ role }, use) => {
15     // Benutzerdetails basierend auf der Rolle erstellen
16     const userMap = {
17       admin: { name: 'Admin User', email: 'admin@example.com', isAdmin: true },
18       editor: { name: 'Editor User', email: 'editor@example.com', isAdmin: false },
19       user: { name: 'Regular User', email: 'user@example.com', isAdmin: false },
20     };
21
22     // ... weiterer Code zur Verwendung des Parameters
23   }
24 }
```

Einsatz in Tests

```
1 // test.spec.ts
2 import { test } from './my-test';
3
4 test('Benutzerberechtigungen basierend auf Rolle', async ({ page, role, user }) => {
5   await page.goto('/dashboard');
6
7   // Parameter und abgeleitete Fixture in Tests verwenden
8   console.log(`Teste als ${role} mit Namen ${user.name}`);
9
10  // Administratorbereich nur für Admins sichtbar
11  if (user.isAdmin) {
12    await expect(page.getByRole('link', { name: 'Admin-Panel' })).toBeVisible();
13  } else {
14    await expect(page.getByRole('link', { name: 'Admin-Panel' })).toBeHidden();
15  }
16})
```

- Test nutzt sowohl **direkten Parameter** (role) als auch **abgeleitete Fixture** (user)
- **Bedingte Testlogik** basierend auf Parametern
- **Ein Test** – läuft aber für alle konfigurierten Projekte

Daten aus CSV-Dateien einlesen

```
1 // csv-test.spec.ts
2 import fs from 'fs';
3 import path from 'path';
4 import { test } from '@playwright/test';
5 import { parse } from 'csv-parse/sync';
6
7 // CSV-Datei einlesen und parsen
8 const records = parse(fs.readFileSync(path.join(__dirname, 'testdaten.csv')), {
9   columns: true,
10  skip_empty_lines: true
11 });
12
13 // Tests für jede Zeile der CSV-Datei erstellen
14 for (const record of records) {
15   test(`Formularvalidierung: ${record.testfall}`, async ({ page }) => {
16     await page.goto('/contact');
17
18     // Formular mit Daten aus CSV ausfüllen
19     await page.getByLabel('Name').fill(record.name);
```

Übung 11 – Mobile Testing

Ziel: Mobile Geräte emulieren und responsive Designs testen.

Aufgaben:

1. **Mobile Projekte konfigurieren:**

- Aktiviere Mobile-Projekte in `playwright.config.ts` (Pixel 5, iPhone 13)
- Versteh Device-Emulation mit User-Agent und Touch-Support

2. **Responsive Navigation testen:**

- Desktop vs. Mobile Navigation (Hamburger Menu)
- Touch-Interaktionen (`tap` statt `click`)

3. **Grid Layout auf verschiedenen Viewports:**

- Desktop: 3 Spalten, Tablet: 2 Spalten, Mobile: 1 Spalte
- Verwende `isMobile` Context-Variable für bedingte Tests

Zeit: 25 Minuten

Nachbesprechung: Mobile Testing

Diskussion der Übungsergebnisse

Shadow DOM und Web Components

```
1 // Shadow DOM Testing
2 test("Shadow DOM Elemente", async ({ page }) => {
3   await page.goto("/page-with-web-components");
4
5   // Auf Shadow DOM zugreifen
6   const shadowHost = page.locator("my-custom-element");
7
8   // In Shadow DOM navigieren
9   const shadowButton = shadowHost.locator("pierce/button.shadow-button");
10  await shadowButton.click();
11
12  // Alternativ mit CSS-Locator
13  await page.locator("my-custom-element >>> button.shadow-button").click();
14
```

- **pierce/**: Spezieller Präfix für Shadow DOM-Durchdringung
- **>>>**: Alternative Syntax für Shadow DOM-Locators

Performance Testing mit Playwright

- **Warum Performance wichtig ist:** Benutzererfahrung, Conversion, SEO
- **Was kann gemessen werden:** Ladezeiten, Rendering, Interaktivität
- **Wie Playwright hilft:** Zugriff auf Browser-Performance-APIs, DevTools Protocol
- **Anwendungsfälle:** Regressionstests, Überwachung, Optimierung

Performance-Metriken: Web Vitals

- **Core Web Vitals:** Wichtigste Nutzer-zentrierte Metriken
 - **LCP (Largest Contentful Paint):** Ladeperformance
 - **FID (First Input Delay):** Interaktivität (in Tests: TBT - Total Blocking Time)
 - **CLS (Cumulative Layout Shift):** Visuelle Stabilität
- **Weitere Metriken:**
 - **TTFB (Time to First Byte):** Server-Antwortzeit
 - **FCP (First Contentful Paint):** Wann erster Inhalt erscheint
 - **TTI (Time to Interactive):** Wann die Seite vollständig interaktiv ist

Performance-APIs in Playwright

```
1 import { test } from '@playwright/test';
2
3 test('Navigation Performance messen', async ({ page }) => {
4     await page.goto('https://google.com');
5
6     const navigationTiming = await page.evaluate(() =>
7         performance.getEntriesByType('navigation'),
8     );
9
10    expect(navigationTiming[0].duration).toBeLessThan(1000);
11});
```

```
1 import { test } from '@playwright/test';
2
3 test('LCP und CLS messen', async ({ page }) => {
4     await page.goto('https://google.com');
5
6     const paintMetrics = await page.evaluate(() =>
7         window.performance.getEntriesByType('paint'),
8     );
9
10    const firstPaint = paintMetrics[0];
11    const firstContentfulPaint = paintMetrics[1];
12
13    expect(firstPaint.startTime).toBeLessThan(1000);
14    expect(firstContentfulPaint.startTime).toBeLessThan(600);
15});
```

Chrome DevTools Protocol für Performance

```
1  test('Netzwerkbedingungen simulieren', async ({ page }) => {
2    // CDP-Session erstellen
3    const client = await page.context().newCDPSession(page);
4    await client.send('Network.enable');
5
6    // Langsames 3G emulieren
7    await client.send('Network.emulateNetworkConditions', {
8      offline: false,
9      downloadThroughput: (1.5 * 1024 * 1024) / 8, // 1.5 Mbps
10     uploadThroughput: (750 * 1024) / 8, // 750 Kbps
11     latency: 40 // 40ms
12   });
13
14    // CPU-Drosselung aktivieren
15    await client.send('Emulation.setCPUThrottlingRate', { rate: 4 });
16
17    await page.goto('https://example.com/');
18
19    // Performance-Metriken unter diesen Bedingungen messen ...
```

Performance-Monitoring: Best Practices

Tipps für Tests

- Testen in CI mit konsistenten Umgebungen
- Testen mit unterschiedlichen Netzwerkprofilen
- Performance-Budgets definieren
- Baseline erstellen und Regressionen tracken
- Headless-Modus für konsistente Ergebnisse

Messwerte nutzen

- Trends über Zeit wichtiger als Einzelwerte
- Core Web Vitals priorisieren
- Kontext beachten (Desktop vs. Mobile)
- Visuelle Metriken (LCP, CLS) mit technischen (TTFB, JS-Zeit) kombinieren
- Nutzerbasierte Feldmessungen ergänzen

Integration mit Lighthouse

```
1 import { test } from '@playwright/test';
2 import { playAudit } from 'playwright-lighthouse';
3
4 test('Lighthouse Performance Test', async ({ browserName, playwright, page }) => {
5   if (browserName !== 'chromium') { return; }
6
7   const browser = await playwright.chromium.launch({
8     args: ['--remote-debugging-port=9222'],
9   });
10  await page.goto('https://www.google.com');
11
12  await playAudit({
13    page: page,
14    port: 9222,
15    thresholds: {
16      performance: 90,
17      accessibility: 90,
18      'best-practices': 90,
19      seo: 80,
```

Ausblick: Continuous Integration (CI/CD)

- **Ziel:** Tests automatisch bei Code-Änderungen ausführen.
- **Vorteile:** Frühes Feedback, Qualitätssicherung, weniger manuelle Tests.
- **Tools:** GitHub Actions, GitLab CI, Jenkins, CircleCI etc.
- **Playwright in CI:** Headless-Modus, parallele Ausführung, Reporting.

Beispiel: GitHub Actions Workflow

```
1  name: Playwright Tests
2  on: [push]
3  jobs:
4    test:
5      runs-on: ubuntu-latest
6      steps:
7        - uses: actions/checkout@v3
8        - uses: actions/setup-node@v3
9        with:
10          node-version: 18
11        - name: Install dependencies
12          run: npm ci
13        - name: Install Playwright
14          run: npx playwright install --with-deps
15        - name: Run Playwright tests
16          run: npx playwright test
17        - uses: actions/upload-artifact@v3
18          if: always()
19          with:
```

Best Practices & Zusammenfassung

- **Robuste Locators:** `getByRole`, `getByText`, `getById` bevorzugen.
- **Auto-Wait nutzen:** Playwright wartet automatisch, `waitFor` selten nötig.
- **Struktur:** POM, Fixtures für Wartbarkeit und Wiederverwendbarkeit.
- **Isolation:** Storage State, API Mocking, Testdaten-Management.
- **Debugging:** Inspector, Trace Viewer sind mächtige Werkzeuge.
- **CI/CD:** Tests früh und oft ausführen.
- **Mobile:** Spezifische Techniken anwenden.

Abschluss & Fragen

- Zusammenfassung der drei Tage
- Wichtigste Erkenntnisse
- Ausblick: Weitere Playwright-Features (Visual Testing, API Testing, Component Testing)
- Offene Fragen & Diskussion

Workshop Abschluss

Zusammenfassung und nächste Schritte



Daniel Sogl - linktr.ee/daniel_sogl

Was haben wir gelernt?

Grundlagen & API

- E2E-Testing und Test-Automatisierung
- Playwright API: Locators, Aktionen, Assertions
- Strukturierung: Page Objects, Fixtures
- Wiederverwendbarkeit: Komponenten, Patterns

Fortgeschrittene Features

- Authentifizierung und Session-Handling
- API-Mocking und Netzwerk-Interception
- Visuelle Tests und Screenshots
- Mobile Testing und Geräte-Emulation
- CI/CD-Integration: Tests in der Pipeline

Nächste Schritte

Technische Schritte

- **Testabdeckung erweitern:** Mehr kritische Pfade
- **Refactoring:** Tests verbessern und wartbarer machen
- **CI/CD-Integration:** Tests in den Build-Prozess

Organisatorische Schritte

- **Team-Schulung:** Weitergabe des Wissens
- **Testautomatisierung-Strategie:** Langfristiger Plan
- **Monitoring:** Testergebnisse verfolgen und analysieren

Ressourcen für die Weiterbildung

Offizielle Ressourcen

- Dokumentation: playwright.dev
- GitHub: github.com/microsoft/playwright
- VS Code Extension: Playwright für VS Code
- Workshop-Material: GitHub-Repository

Community & Updates

- Discord: playwright.dev/community/discord
- GitHub Discussions: Fragen und Antworten
- Blog-Artikel: Regelmäßige Updates
- YouTube-Tutorials: Microsoft Developer Channel

Häufige Fehlerquellen

Technische Fehler

- **Locators zu spezifisch:** Anfällig für UI-Änderungen
- **Feste Wartezeiten:** `page.waitForTimeout()` statt Events
- **Fehlende Assertions:** Keine klare Erfolgsvvalidierung

Strukturelle Fehler

- **Übergroße Tests:** Zu viele Schritte in einem Test
- **Unzureichende Isolation:** Tests beeinflussen einander
- **Langsame Tests:** Unnötige Aktionen und Navigationen

Tipps für erfolgreiche Playwright-Tests

Codequalität

- **Kleinhaltung:** Tests auf einen Aspekt fokussieren
- **Wiederverwendbarkeit:** Helper-Funktionen nutzen
- **Gute Locators:** Testdaten-Attribute verwenden
- **Leserliche Tests:** Beschreibende Namen

Testqualität

- **Saubere Umgebung:** Status zwischen Tests zurücksetzen
- **Frühe Fehlererkennung:** Assertions früh platzieren
- **Debugging-Informationen:** Screenshots bei Fehlern
- **Deterministische Tests:** Keine zufälligen Daten

