

Playwright Workshop

Modernes E2E-Testing leicht gemacht



Daniel Sogl - linktr.ee/daniel_sogl

Über den Trainer

Daniel Sogl

Software Architect & Conference Speaker bei
Thinktecture AG

- 🎯 Spezialist für Angular, Ionic und AI-Coding
- 🎤 Tech Speaker & Workshop Trainer
- 🔎 Experte für Test Automation (Playwright, Cypress)
- 📚 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

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

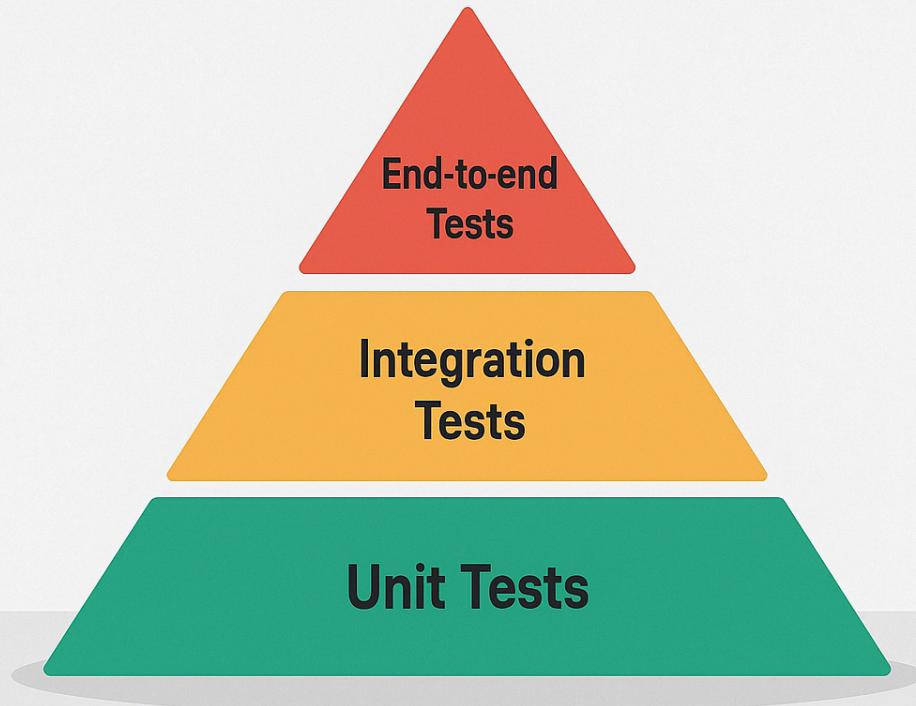
Nach Ebene

- **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

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
- **Schwierige Diagnose:** Fehlerursachen können schwer zu identifizieren sein

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

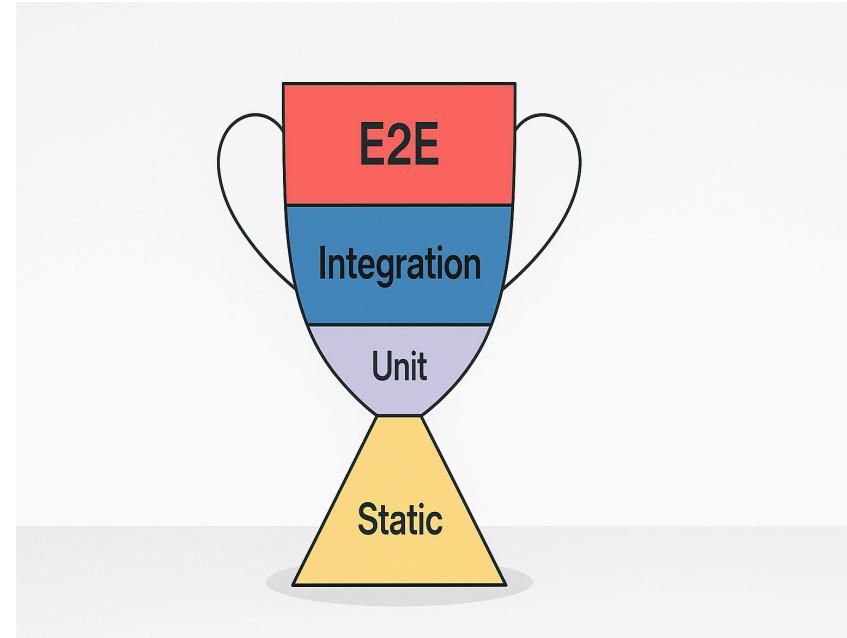
*Die Pyramide ist ein **Modell**, keine starre Regel. Der Kontext (Architektur, Team, Risiken) ist entscheidend.*

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

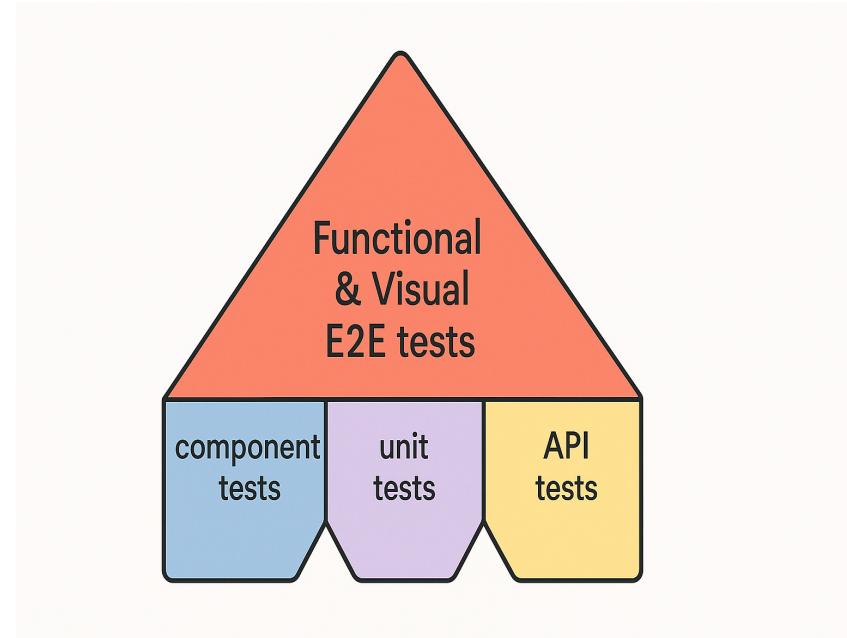


Testing Crab – Struktur & Merkmale

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 Grundlagen

Tag 1: Einstieg in Playwright und E2E-Testing



Daniel Sogl - linktr.ee/daniel_sogl

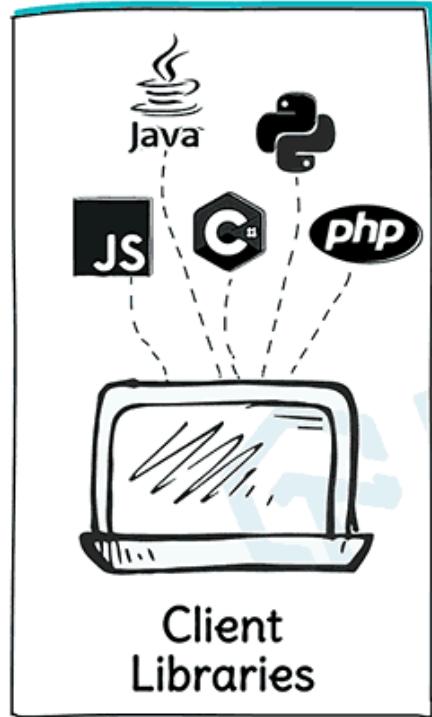
Was ist Playwright? – Überblick & Geschichte

- **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

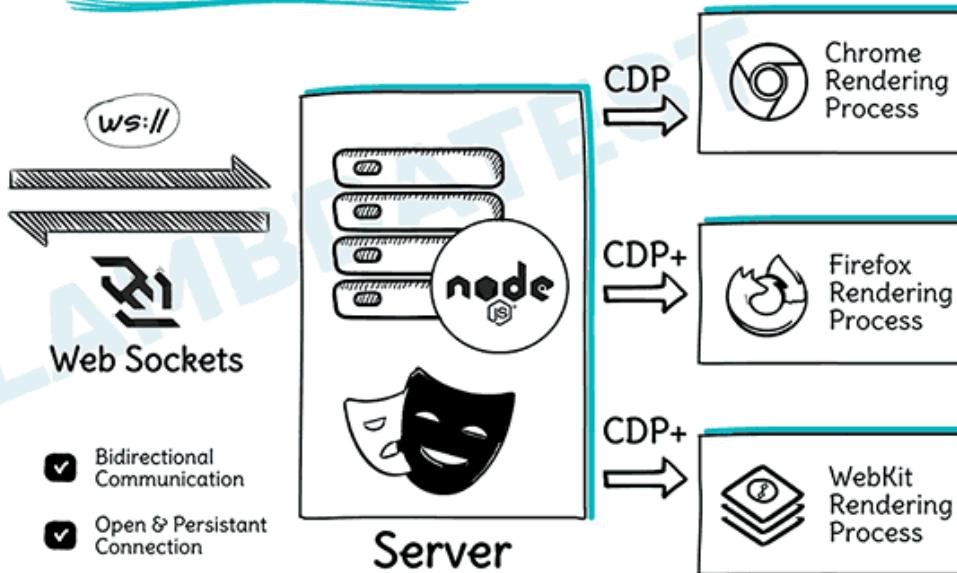
Warum Playwright?

- **Multi-Browser-Support** aus einer Codebasis
- **Auto-Wait:** Intelligente Synchronisation
- **Network Interception:** API-Mocking
- **Mobile Emulation:** Responsive Testing
- **Isolation:** Zuverlässige Tests durch Browser-Contexts
- **Moderne Architekturen:** Unterstützt SPAs, Shadow-DOM, iframes
- **Traces:** Umfangreiche Debug-Möglichkeiten
- **Geschwindigkeit & Zuverlässigkeit**

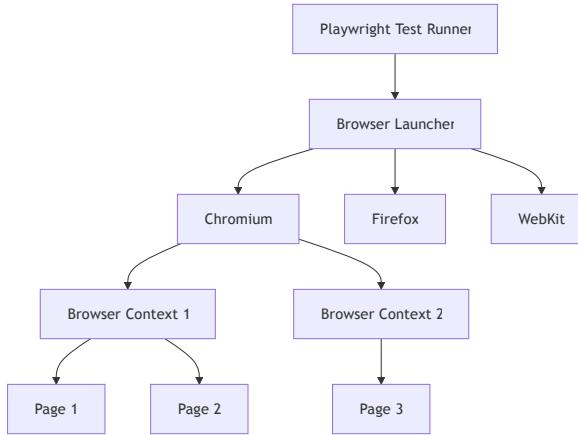
Playwright-Architektur im Überblick



PLAYWRIGHT ARCHITECTURE



Playwright Test-Architektur



- **Browser:** Die Browserinstanz (Chrome, Firefox, Safari)
- **Browser Context:** Isolierte Umgebung (wie Inkognito-Fenster)
- **Page:** Einzelne Browserseite innerhalb eines Contexts

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 (v14 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

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 0 – Projekt-Setup

Ziel: Playwright-Projekt für die Feed App einrichten & prüfen.

Aufgaben:

1. Playwright-Projekt im `e2e/` Ordner initialisieren (`npx playwright test --init`).
2. Abhängigkeiten installieren & App starten.
3. Einen ersten Beispieltest schreiben (Startseite öffnen, Titel prüfen).
4. Projektstruktur (`playwright.config.ts` , `e2e/`) ansehen.

Zeit: 15 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
```

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,
20      reuseExistingServer: !process.env.CI,
21    }
22  }
```

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 });
```

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 });
```

Beispiele für Annotierungen

```
1 import { test, expect } from '@playwright/test';
2
3 test('test', async ({ page }) => {
4     await test.step('Log in', async () => {
5         // ...
6     });
7
8     await test.step('Outer step', async () => {
9         // ...
10        await test.step('Inner step', async () => {
11            // ...
12        });
13    });
14});
```

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 Benutzermenü 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(/TANIA/);
```

Test-Hooks und Fixtures

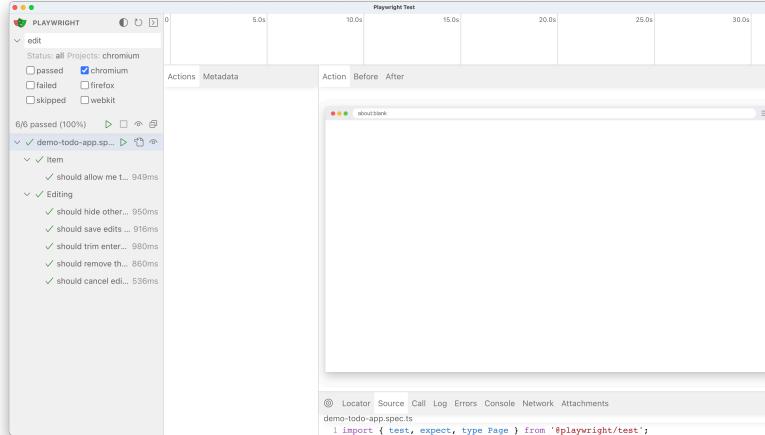
```
1 test.describe("Produktseiten-Tests", () => {
2     // Vor allen Tests in dieser Gruppe ausführen
3     test.beforeAll(async ({ browser }) => {
4         // Gemeinsames Setup, z.B. Testdaten erstellen
5         const context = await browser.newContext();
6         const page = await context.newPage();
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 });
```

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
```



Testausführung – Erweiterte Optionen

```
1 # Bestimmte Testdatei ausführen
2 npx playwright test e2e/login.spec.ts
3
4 # Tests mit bestimmtem Namen ausführen
5 npx playwright test -g "Login Funktionalität"
6
7 # Tests im Watch-Modus ausführen (bei Änderungen neu starten)
8 npx playwright test --watch
9
10 # Headed-Modus (sichtbare Browser)
11 npx playwright test --headed
12
13 # Debugging mit Inspector
14 npx playwright test --debug
15
16 # Parallelle Ausführung konfigurieren
17 npx playwright test --workers=4
```

Headless vs. Headed Testing

Headless Testing (Standard)

- Schnellere Ausführung
- Ressourcenschonender
- Ideal für CI/CD
- Keine UI-Interaktion nötig
- Geeignet für Regressionstests

Headed Testing (Sichtbar)

- Bessere Visualisierung
- Einfacheres Debugging
- Langsamer als Headless
- Mehr Ressourcenverbrauch
- Geeignet für Entwicklung und Fehlersuche

```
1 # Headed-Modus aktivieren
2 npx playwright test --headed
3
4 # In der Konfiguration (playwright.config.ts)
5 use: [
6   headless: false // Global oder pro Projekt
7 }
```

Selektoren

Selektoren in Playwright – Textselektoren

```
1 // Text-Selektor - exakter Text
2 page.locator("text=Anmelden").click();
3
4 // Text-Selektor - teilweiser Text
5 page.locator("text=Anm").click();
6
7 // Text-Selektor - mit Regex
8 page.locator("text=/Anmeld(en|ung)/").click();
9
10 // Text innerhalb eines Elements
11 page.locator('button:has-text("Anmelden")').click();
```

Moderne Selektoren-API

```
1 // Moderne API (empfohlen)
2 await page.getText("Anmelden").click();
3 await page.getText(/Anmeld(en|ung)/).click();
4
5 // Kombiniert mit anderen Eigenschaften
6 await page.getText("Anmelden", { exact: true }).click();
7 await page.getText("Anmelden").first().click();
8
9 // Weitere nützliche Selektoren
10 await page.getLabel("Benutzername").fill("user123");
11 await page.getPlaceholder("Passwort eingeben").fill("pass123");
12 await page.getAltText("Firmenlogo").isVisible();
13 await page.getTitle("Hilfe").hover();
```

Selektoren-Strategien und Best Practices

Selektoren: Empfohlene Reihenfolge

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

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

2. Test-spezifische Attribute:

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

Selektoren: Zu vermeiden (wenn möglich)

- CSS-Selektoren, die auf Implementierungsdetails basieren (Styling-Klassen, komplexe Strukturen)
 - Beispiel: `.button-styles.primary > span`
- XPath-Selektoren (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 Selektoren zu finden! Playwright priorisiert automatisch benutzerorientierte Selektoren.

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 Selektoren:** 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-Selektoren.

Empfehlung

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

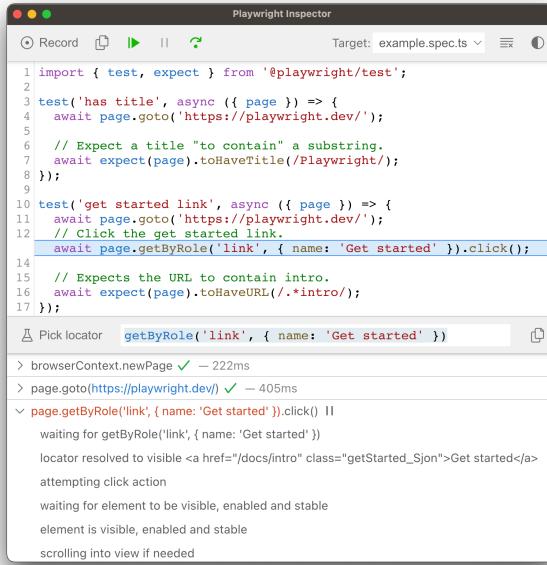
Selektoren in Playwright – CSS & XPath

```
1 // CSS-Selektoren
2 page.locator("#login-button").click();
3 page.locator(".nav-item").first().click();
4 page.locator('form > input[type="password"]').fill("12345");
5
6 // XPath-Selektoren (weniger empfohlen)
7 page.locator('xpath=//button[@type="submit"]').click();
8 page.locator('//div[@class="user-info"]').hover();
```

Fortgeschrittene CSS-Selektoren

```
1 // Mehrere Bedingungen kombinieren
2 page.locator('button.primary[type="submit"]').click();
3
4 // Eltern-Kind-Beziehungen
5 page.locator(".form-group > input").fill("Text");
6
7 // Geschwister-Elemente
8 page.locator("label + input").fill("Text");
9
10 // Nth-Element
11 page.locator("li:nth-child(3)").click();
12
13 // Attribut enthält
14 page.locator('[class*="button"]').click();
15
16 // Mehrere Elemente mit Filterung
17 const buttons = page.locator("button");
18 await buttons.filter({ hasText: "Speichern" }).click();
```

Selektoren finden mit Playwright Inspector



- **Codegen:** Automatische Generierung von Selektoren
- **Inspector:** Interaktives Auswählen von Elementen
- **Picker:** Elemente auf der Seite auswählen

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`.
- **Vorteile:**
 - Spart Zeit bei der Erstellung von Tests.
 - Generiert sofort ausführbaren Code.
 - Hilft bei der Auswahl stabiler Selektoren.
- **Demo:**
 - Öffnen Sie ein Terminal und führen Sie `npx playwright codegen` aus.
 - Interagieren Sie mit Ihrer Anwendung, um automatisch Testcode zu generieren.

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 pageByText("Doppelklick").dblclick();
15
16 // Rechtsklick
17 await pageByText("Kontextmenü").click({ button: "right" });
18
19 // Mit Modifiern klicken
20 await page.getRole("link").click({ modifiers: ["Shift"] });
```

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();
```

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

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');
```

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 1 – Erster Test

Ziel: Ersten Playwright-Test für die Feed App schreiben.

Aufgaben:

1. Test erstellen (`e2e/navigation.spec.ts`).
2. Startseite öffnen, auf "View Public News" klicken.
3. Prüfen, ob zur `/news/public` Seite navigiert wurde (URL, Überschrift).
4. Test ausführen (`npx playwright test`).

Zeit: 20 Minuten

Nacbhesprechung

Übung 2 – Filtertest im News Feed

Ziel: Filter- und Suchfunktionen im News Feed testen.

Aufgaben:

1. Test erstellen (z.B. `e2e/news-feed.spec.ts`).
2. Zur `/news/public` Seite navigieren.
3. Suchfunktion testen (Begriff eingeben, Ergebnisse prüfen, leeren).
4. Kategorie-Filter testen (Kategorie wählen, Ergebnisse prüfen, zurücksetzen).
5. Trace aktivieren & im Report analysieren (`npx playwright show-report`).

Zeit: 25 Minuten

Nacbhesprechung

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 Selektoren 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 Selektoren
- **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, 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
- **Selektoren** 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    // Selektoren testen
16    const buttonExists = await page
17        .getByRole("button", { name: "Submit" })
18        .isVisible();
19    console.log("Button sichtbar:", buttonExists);
20});
```

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

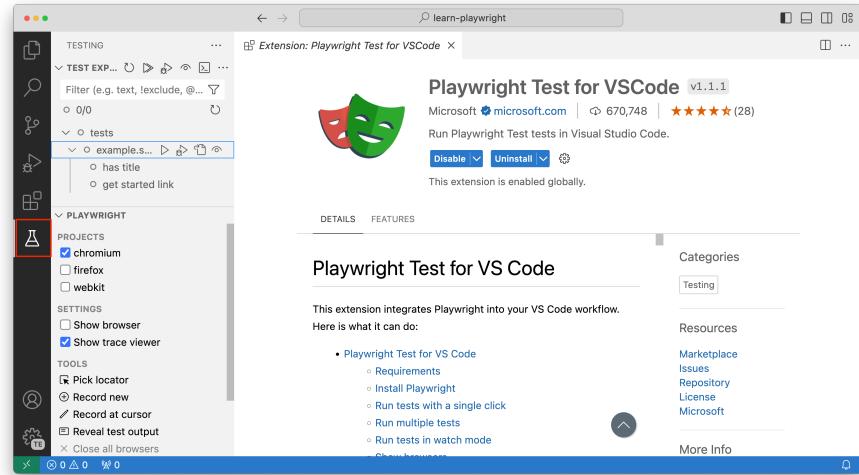
- Screenshots (vor/nach jeder Aktion)
- DOM-Snapshots
- Netzwerkanfragen
- Konsolenausgaben
- Testschritte mit Zeitstempeln
- Browserkontext-Informationen

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 }
```

Playwright in der IDE: Visual Studio Code

- Offizielle Extension: **Playwright Test for VS Code**
- Test-Explorer: Tests ausführen, debuggen und Ergebnisse direkt in der IDE sehen
- Breakpoints setzen und interaktiv debuggen
- Trace-Viewer und Screenshots direkt öffnen
- Selektoren testen und inspizieren
- Installation: Im Extensions Marketplace nach "Playwright Test" suchen



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

Authentifizierung & Teststrukturen

Tag 2: Auth-Handling, Fixtures & Page Objects



Daniel Sogl - linktr.ee/daniel_sogl

Wiederholung: Tag 1 Konzepte

- **Grundlagen von Playwright:**
 - Installation und Konfiguration
 - Selektoren und Aktionen
 - Assertions und Auto-Wait
- **Wichtige Erkenntnisse:**
 - Selektoren sollten robust und semantisch sein
 - Auto-Wait reduziert flaky Tests
 - Teststruktur beeinflusst Wartbarkeit
- **Offene Fragen aus Tag 1?**

Tagesagenda im Detail

- **Vormittag:**
 - Accessibility Testing
 - Authentifizierung (Strategien, Storage State)
 - Session-Handling
- **Nachmittag:**
 - Fixtures (Eingebaute, Eigene)
 - Page Object Model (Einführung, Architektur, Best Practices)
 - Praktische Übungen (Login, Fixtures, POM Refactoring)
- **Offene Fragen aus Tag 1?**

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

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

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 });
```

Übung 2A – Accessibility-Check mit Playwright

Ziel: Automatisierten Accessibility-Check für eine Seite schreiben.

Aufgaben:

1. Installiere `@axe-core/playwright` als Dev-Dependency.
2. Schreibe einen Playwright-Test, der eine Seite aufruft und mit `AxeBuilder.analyze()` prüft, ob keine Accessibility-Violations gefunden werden.
3. Führe den Test aus und analysiere ggf. gefundene Probleme.

Zeit: 20 Minuten

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

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": "eyJhbGciOiJIUzI1NiT0PDECT6TKeyVicj0=
```

Browser-Kontext und Isolation

```
1 // Mehrere isolierte Kontexte
2 const browser = await chromium.launch();
3
4 // Admin-Kontext
5 const adminContext = await browser.newContext({
6   storageState: 'playwright/.auth/admin.json',
7 });
8 const adminPage = await adminContext.newPage();
9
10 // Benutzer-Kontext
11 const userContext = await browser.newContext({
12   storageState: 'playwright/.auth/user.json',
13 });
14 const userPage = await userContext.newPage();
15
16 // ... Interaktion mit beiden Rollen ...
17
18 await adminContext.close();
19 await userContext.close();
```

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 });


```

Verschiedene Benutzerrollen verwalten

```
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');
20    await userPage.goto('/user-dashboard');
21
```

Codebeispiel Storage State

```
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 })
```

Übung 3 – Login automatisieren mit Storage State

Ziel: Effizienten Login durch Speichern/Wiederverwenden des Auth-Status.

Aufgaben:

1. Login-Setup (`auth.setup.ts`) erstellen, das UI-Login durchführt.
2. Konfiguration (`playwright.config.ts`) anpassen, um Setup-Projekt zu nutzen & `storageState` zu laden.
3. Authentifizierten Test schreiben, der direkt auf geschützte Seite zugreift.

Zeit: 30 Minuten

Übung 3B – API-basierte Authentifizierung

Ziel: Login-Setup durch API-Authentifizierung optimieren.

Aufgaben:

1. Login-Setup (`auth.setup.ts`) anpassen, um Login über API-Request (`request.post`) durchzuführen.
2. CSRF-Token und Credentials (aus `.env.test`) verwenden.
3. `storageState` speichern.
4. Tests schreiben, die den API-authentifizierten Zustand nutzen (UI & API).

Zeit: 30 Minuten

Fixtures – Einführung & Nutzen

- **Fixtures:** Wiederverwendbare Testumgebungen
- **Vorteile:**
 - Setup und Teardown Automatisierung
 - Ressourcen-Sharing zwischen Tests
 - Testcode-Vereinfachung
 - Bessere Testorganisation

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

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

Eigene Fixtures erstellen

```
1 // fixtures.ts
2 import { test as base } from "@playwright/test";
3
4 // Eigene Fixtures hinzufügen
5 type MyFixtures = {
6   loggedInPage: Page;
7   testUser: { username: string; email: string };
8 };
9
10 // Test mit eigenen Fixtures erweitern
11 export const test = base.extend<MyFixtures>({
12   // Fixture für eingeloggten User
13   loggedInPage: async ({ page }, use) => {
14     // Setup: Login durchführen
15     await page.goto("/login");
16     await page.fill("#email", "test@example.com");
17     await page.fill("#password", "password");
18     await page.click("#login-button");
19
20     // Fixture bereitstellen
21     await use(page);
22   }
23 });
```

Fixture-Optionen definieren

```
1  // fixtures.ts
2  import { test as base } from "@playwright/test";
3
4  // Fixture-Optionen definieren
5  type MyOptions = {
6    userRole: "admin" | "editor" | "viewer";
7    databaseReset: boolean;
8  };
9
10 // Fixtures mit Optionen
11 type MyFixtures = {
12   authenticatedPage: Page;
13   testData: any;
14 };
15
16 // Test mit Optionen und Fixtures erweitern
17 export const test = base.extend<MyFixtures, MyOptions>({
18   // Option mit Standardwert
19   userRole: ["viewer", { option: true }],
20   databaseReset: [true, { option: true }],
21   ...
22 }
```

Rollenbasierte Authentifizierungs-Fixture

```
1 // fixtures.ts (Fortsetzung)
2 export const test = base.extend<MyFixtures, MyOptions>({
3     // Fixture, die eine Option verwendet
4     authenticatedPage: async ({ page, userRole }, use) => {
5         // Login basierend auf Rolle
6         await page.goto("/login");
7
8         switch (userRole) {
9             case "admin":
10                 await page.fill("#email", "admin@example.com");
11                 await page.fill("#password", "adminpass");
12                 break;
13             case "editor":
14                 await page.fill("#email", "editor@example.com");
15                 await page.fill("#password", "editorpass");
16                 break;
17             default:
18                 await page.fill("#email", "viewer@example.com");
19                 await page.fill("#password", "viewerpass");
20         }
21     }
22 }
```

Testdaten-Fixture mit Optionen

```
1 // fixtures.ts (Fortsetzung)
2 export const test = base.extend<MyFixtures, MyOptions>({
3     // Fixture, die eine Option verwendet
4     testData: async ({ databaseReset }, use) => {
5         // Testdaten vorbereiten
6         let data = {};
7
8         if (databaseReset) {
9             // Datenbank zurücksetzen und Testdaten laden
10            data = await setupFreshTestData();
11        } else {
12            // Existierende Daten verwenden
13            data = await getExistingTestData();
14        }
15
16        await use(data);
17    },
18});
```

Fixture-Abhängigkeiten und Komposition

```
1 // fixtures.ts
2 import { test as base } from "@playwright/test";
3 import { LoginPage } from "../pages/LoginPage";
4 import { DashboardPage } from "../pages/DashboardPage";
5
6 // Fixtures definieren
7 type MyFixtures = {
8   loginPage: LoginPage;
9   dashboardPage: DashboardPage;
10  authenticatedUser: { id: string; name: string; role: string };
11  adminUser: { id: string; name: string; role: string };
12};
13
14 export const test = base.extend<MyFixtures>({
15   // Page-Object-Fixtures
16   loginPage: async ({ page }, use) => {
17     await use(new LoginPage(page));
18   },
19
20   dashboardPage: async ({ page }, use) => {
21     await use(new DashboardPage(page));
22   }
23});
```

Beispiel: Fixture für Login-Setup

```
1 // e2e/example.spec.ts
2 import { test, expect } from "./fixtures";
3
4 test("Dashboard zeigt Benutzerinfo an", async ({ loggedInPage, testUser }) => {
5   await loggedInPage.goto("/dashboard");
6
7   const usernameElement = loggedInPage.getByTestId("username");
8   await expect(usernameElement).toBeVisible();
9   await expect(usernameElement).toHaveText(testUser.username);
10 });
11
12 test("Profil kann bearbeitet werden", async ({ loggedInPage }) => {
13   await loggedInPage.goto("/profile");
14   // Test mit bereits angemeldetem Benutzer
15 });
```

Übung 3C – Eigene Fixtures erstellen

Ziel: Wiederkehrende Setup-Schritte mit eigenen Fixtures kapseln.

Aufgaben:

1. Fixture-Datei (`feed-fixtures.ts`) erstellen und `base.extend` nutzen.
2. Fixtures definieren (z.B. `publicFeeds` für vorbereitete Public News Seite, `feedApi` für authentifizierten API-Client).
3. Tests schreiben, die diese Fixtures verwenden.

Zeit: 40 Minuten

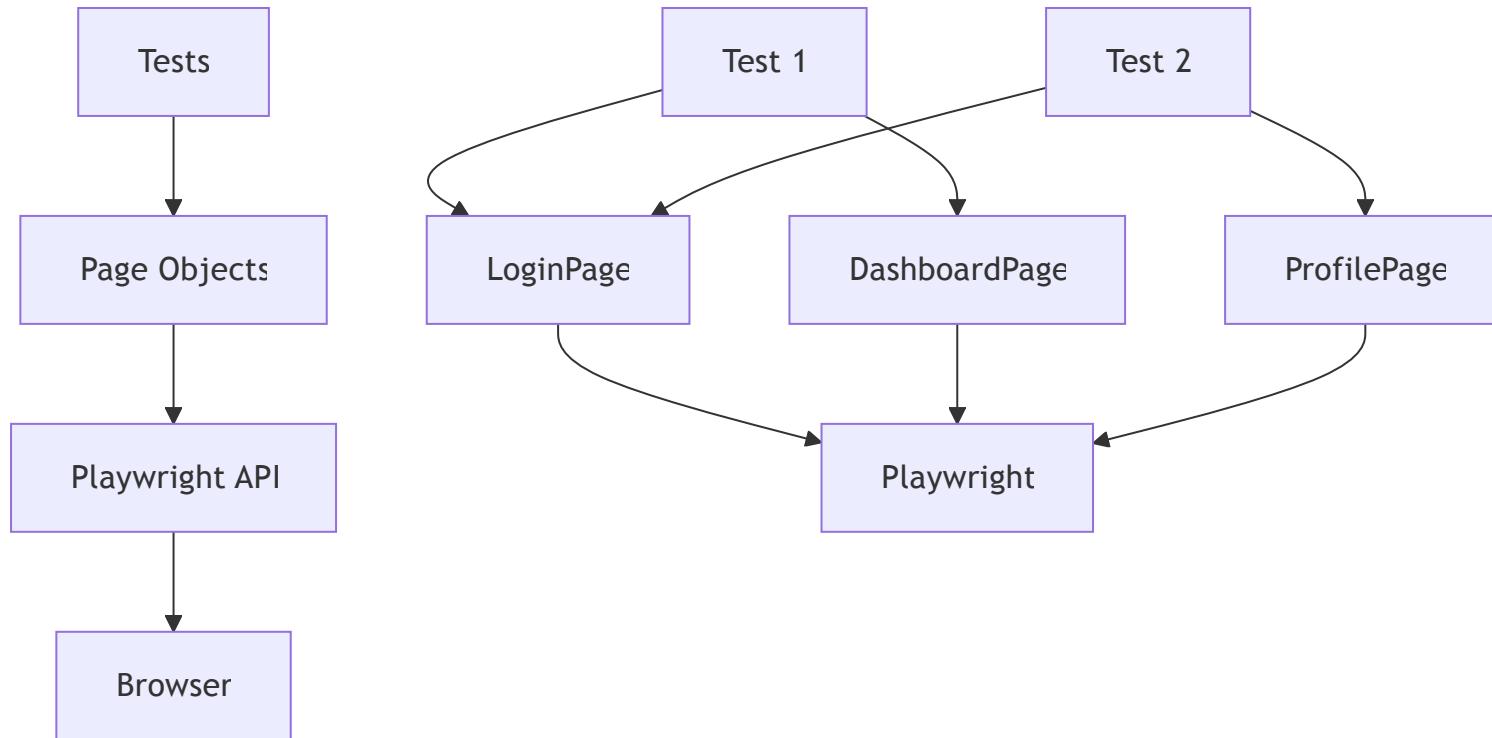
Page Object Model – Einführung

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

Live-Demo: Basic POM

Praktische Demonstration der Konzepte

Page Object Model – Architektur



- **Tests:** Beschreiben Geschäftslogik und Erwartungen

Nachbesprechung: POM Intro

Diskussion der Übungsergebnisse

Alternativen zum Page Object Model

```
1 ## Alternativen zum Page Object Model
2
3 #### Component Object Model
4
5 - Fokus auf UI-Komponenten statt Seiten
6 - Besser für SPAs und komplexe UIs
7 - Wiederverwendbare Komponenten
8 - Beispiel: Header, Sidebar, Modal
9
10 #### App Actions Pattern
11
12 - Fokus auf Benutzeraktionen
13 - Abstrahiert UI-Details
14 - API-ähnliche Schnittstelle
15 - Beispiel: `app.login()`, `app.createTodo()`
```

```
1 // Component Object Beispiel
2 class TodoItem {
3   constructor(private root: Locator) {}
4
5   async toggle() {
6     await this.root.getByRole("checkbox").click();
```

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) {
20    await this.emailInput.fill(email);
21    await this.passwordInput.fill(password);
```

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 4 – Refactoring mit Page Objects

Ziel: Bestehende Tests (Übung 2) mit Page Object Model refaktorieren.

Aufgaben:

1. `PublicNewsPage` Klasse erstellen (`e2e/pages/PublicNewsPage.ts`).
2. Selektoren und Interaktionsmethoden (Suche, Filter) kapseln.
3. Hilfsmethoden für Datenextraktion (Anzahl, Titel, Kategorie) hinzufügen.
4. Tests aus Übung 2 refaktorieren, um `PublicNewsPage` zu verwenden.

Zeit: 30 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, iframe, CI/CD

Fortgeschrittene Themen & Testing-Strategien

Tag 3: Advanced POM, API Mocking, Datenmanagement, Mobile, iframe & CI/CD



Wiederholung: Tag 2 Konzepte

- **Debugging:** Inspector, Trace Viewer, `page.pause()`
- **Authentifizierung:** UI vs. API, Storage State, Global Setup
- **Fixtures:** Eingebaute (`page`, `context`), Eigene (`base.extend`), Scopes
- **Page Object Model:** Grundlagen, Architektur, Best Practices
- **Offene Fragen aus Tag 2?**

Tagesagenda im Detail

Vormittag

- Fortgeschrittene POM Patterns (Komponenten, Fluent Interface)
- API Mocking (`page.route`)
- Testing-Strategien (Datenmanagement, Reset)
- Praktische Übungen (Advanced POM, API Mocking)

Nachmittag

- Mobile Testing (Emulation, Responsive, Touch)
- iframe Testing (`frameLocator`)
- Kurzer Ausblick: CI/CD
- Praktische Übungen (Mobile, iframe)
- Zusammenfassung & Abschluss

Fortgeschrittene Page Object Patterns

```
1 // Basis Page Object
2 abstract class BasePage {
3     constructor(protected page: Page) {}
4
5     // Gemeinsame Methoden
6     async waitForPageLoad() {
7         await this.page.waitForLoadState("networkidle");
8     }
9
10    // Navigation mit Typensicherheit
11    async navigateTo<T extends BasePage>(
12        PageClass: new (page: Page) => T,
13        url: string,
14    ): Promise<T> {
15        await this.page.goto(url);
16        return new PageClass(this.page);
17    }
18 }
19
20 // Konkrete Implementierung
21 class LoginPage extends BasePage {
```

Erweitertes Page Object Beispiel

```
1 // pages/TodoPage.ts
2 import { Page, Locator } from "@playwright/test";
3
4 export class TodoPage {
5     // Selektoren als private Properties
6     private readonly newTodoInput: Locator;
7     private readonly todoList: Locator;
8     private readonly todoItems: Locator;
9     private readonly activeFilterButton: Locator;
10    private readonly completedFilterButton: Locator;
11    private readonly allFilterButton: Locator;
12
13    constructor(private readonly page: Page) {
14        this.newTodoInput = page.getByPlaceholder("What needs to be done?");
15        this.todoList = page.locator(".todo-list");
16        this.todoItems = this.todoList.locator("li");
17        this.activeFilterButton = page.getRole("link", { name: "Active" });
18        this.completedFilterButton = page.getRole("link", { name: "Completed" });
19        this.allFilterButton = page.getRole("link", { name: "All" });
20    }
21}
```

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     private readonly deleteButton: Locator;
8
9     constructor(private readonly root: Locator) {
10         this.checkbox = root.getByRole("checkbox");
11         this.label = root.locator("label");
12         this.deleteButton = root.locator(".destroy");
13     }
14
15     async toggle() {
16         await this.checkbox.click();
17     }
18
19     async delete() {
20         // Hover zum Anzeigen des Lösch-Buttons
21         await this.root.hover()
```

Fluent Interface Pattern

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

Übung 4B – Erweiterte Page Objects (Komponenten & Fluent Interface)

Ziel: POM verbessern durch Komponenten-Objekte & Fluent Interface.

Aufgaben:

1. NewsItemComponent erstellen, das ein einzelnes News-Item kapselt.
2. PublicNewsPage anpassen, um NewsItemComponent zu verwenden.
3. Fluent Interface (return this) in PublicNewsPage implementieren.
4. Tests refaktorieren, um Komponenten und Fluent Interface zu nutzen.

Zeit: 30 Minuten

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 5 – 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

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);
15  }
```

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     // Teardown: Artikel löschen
```

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 7 – Visuelle Regression testen

Ziel: Visuelle Regressionstests für eine Seite oder Komponente einführen.

Aufgaben:

1. Schreibe einen Test mit `toHaveScreenshot()` für eine Seite oder ein UI-Element. 2. Führe den Test aus und prüfe, ob ein Snapshot erstellt wird. 3. Simuliere eine visuelle Änderung und führe den Test erneut aus. 4. Aktualisiere den Snapshot mit `--update-snapshots` und prüfe das Ergebnis.

Zeit: 25 Minuten

Spezialathemen

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
20  const page = await context.newPage();
21  await page.goto("https://example.com")
```

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
20    // Hamburger-Menü testen
21    await page.getByRole("button", { name: "Menü" }).click();
```

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(
```

Übung 6A – Responsive Design Testing (Feed App)

Ziel: Umfassende Tests für responsives Design mit Emulation & Visual Testing.

Aufgaben:

1. Responsive Layouts für verschiedene Viewports testen (Navigation, Grid).
2. Touch-Interaktionen (`tap` , Swipe/Pull-to-Refresh) simulieren.
3. Visuelle Tests mit `toHaveScreenshot` für Seiten und Komponenten durchführen.

Zeit: 45 Minuten

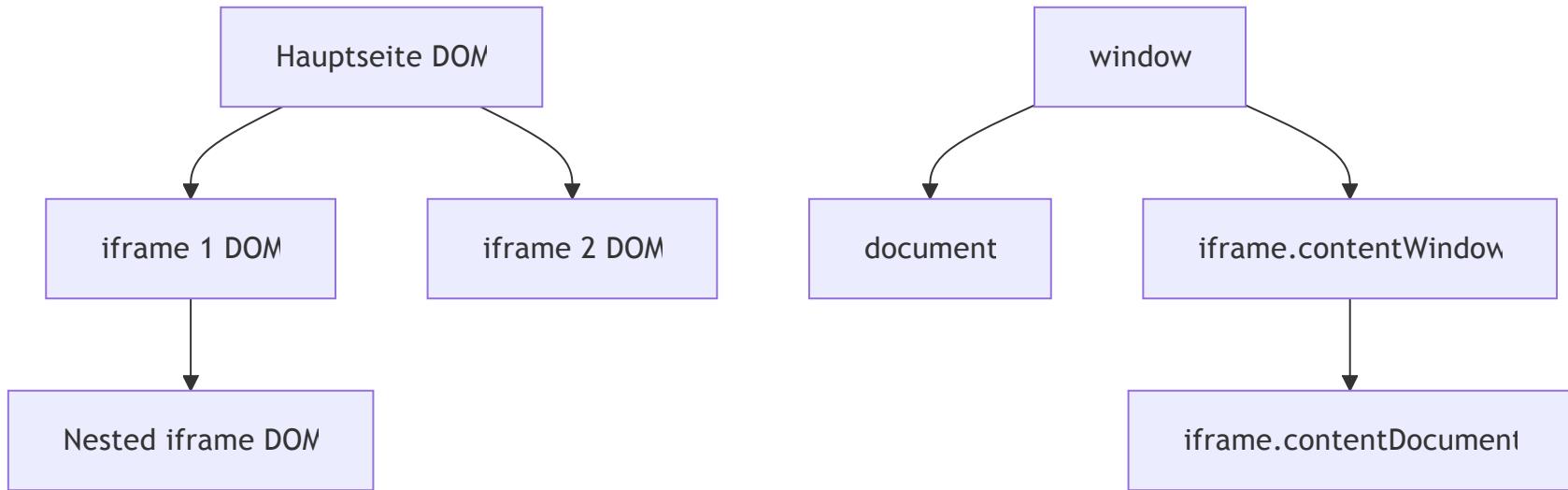
Nachbesprechung: Mobile & Responsive

Diskussion der Übungsergebnisse

iframe Testing – Herausforderungen

- Isolierter DOM-Baum für jedes iframe
- Zugriff auf **iframe-Inhalte** erfordert spezielle Behandlung
- Cross-Origin Beschränkungen können Zugriff erschweren
- Synchronisation zwischen Haupt- und iframe-Kontext
- Shadow DOM versteckt oft iframe-Inhalte

iframe-Architektur verstehen



- **Separate Browsing Contexts:** Jedes iframe hat eigenen DOM-Baum
- **Isolierte JavaScript-Umgebungen:** Variablen und Funktionen sind getrennt
- **Separate Event Loops:** Events werden unabhängig verarbeitet
- **Eigene History:** Navigation im iframe beeinflusst nicht die Hauptseite

Cross-Origin iframes - Grundlagen

Same-Origin Policy

- Gleiche Protokoll + Domain + Port
- Voller Zugriff auf iframe-Inhalte
- JavaScript-Interaktion möglich
- DOM-Manipulation erlaubt

Cross-Origin Restrictions

- Unterschiedliche Herkunft
- Eingeschränkter Zugriff
- Keine DOM-Manipulation
- Kommunikation nur über postMessage
- Cookies und LocalStorage isoliert

Cross-Origin iframes - Beispiel

```
1 // Cross-Origin iframe testen
2 test("Cross-Origin iframe", async ({ page }) => {
3     await page.goto("https://example.com/page-with-iframe");
4
5     // Auf iframe zugreifen
6     const frameLocator = page.frameLocator(
7         'iframe[src^="https://different-origin.com"]',
8     );
9
10    // Mit iframe-Inhalten interagieren
11    await frameLocator.getByRole("button", { name: "Accept" }).click();
12
13    // Kommunikation zwischen Frames prüfen
14    await page.evaluate(() => {
15        // postMessage an iframe senden
16        document
17            .querySelector("iframe")
18            ?.contentWindow?.postMessage(
19                { type: "TEST_MESSAGE" },
20                "https://different-origin.com",
21            );
22    });
23})
```

frameLocator – Einführung & Beispiel

```
1 test("iframe Interaktion", async ({ page }) => {
2   await page.goto("/page-with-iframe");
3
4   // Zugriff auf iframe mit frameLocator
5   const frameLocator = page.frameLocator("#my-iframe");
6
7   // Interaktion mit Elementen im iframe
8   await frameLocator.getByRole("button", { name: "Submit" }).click();
9
10  // Erwartungen an iframe-Inhalte
11  await expect(
12    frameLocator.getText("Form submitted successfully"),
13  ).toBeVisible();
14
15  // Kombinierte iframe und Hauptseiten-Interaktion
16  await frameLocator.getByRole("button", { name: "Close" }).click();
17  await expect(page.getText("iframe closed")).toBeVisible();
18});
```

Verschiedene Methoden für iframe-Zugriff

```
1 // Methode 1: frameLocator (empfohlen)
2 const frameLocator = page.frameLocator("#my-iframe");
3 await frameLocator.getByRole("button").click();
4
5 // Methode 2: frame-Objekt
6 const frame = page.frame("iframe-name"); // nach name-Attribut
7 // oder
8 const frame = page.frame({ url: /iframe-url/ }); // nach URL
9 if (frame) {
10   await frame.getByRole("button").click();
11 }
12
13 // Methode 3: Auf alle Frames zugreifen
14 const allFrames = page.frames();
15 const myFrame = allFrames.find((frame) =>
16   frame.url().includes("iframe-content"),
17 );
18 if (myFrame) {
19   await myFrame.getByRole("button").click();
20 }
```

Verschachtelte iframes

```
1 // Zugriff auf verschachtelte iframes
2 test("Verschachtelte iframes", async ({ page }) => {
3   await page.goto("/page-with-nested-iframes");
4
5   // Zugriff auf äußeres iframe
6   const outerFrame = page.frameLocator("#outer-iframe");
7
8   // Zugriff auf inneres iframe innerhalb des äußeren
9   const innerFrame = outerFrame.frameLocator("#inner-iframe");
10
11  // Interaktion mit Elementen im inneren iframe
12  await innerFrame.getByRole("button", { name: "Submit" }).click();
13
14  // Erwartungen an inneres iframe
15  await expect(
16    innerFrame.getText("Form submitted successfully"),
17  ).toBeVisible();
18
19  // Zurück zum äußeren iframe
20  await outerFrame.getByRole("link", { name: "Back" }).click();
21
```

Häufige iframe-Anwendungsfälle

Typische iframe-Szenarien

- Eingebettete Videos (YouTube, Vimeo)
- Zahlungs-Widgets (Stripe, PayPal)
- Chat-Widgets (Intercom, Drift)
- Social Media Embeds (Twitter, Facebook)
- Formulare von Drittanbietern
- Werbeanzeigen
- Karten (Google Maps, OpenStreetMap)

Beispiel: YouTube-Video testen

```
1  test("YouTube-Video", async ({ page }) => {
2    await page.goto("/page-with-youtube");
3
4    // YouTube iframe finden
5    const ytFrame = page.frameLocator('iframe[src*="youtube.co"];
6
7    // Play-Button klicken
8    await ytFrame.locator(".ytp-large-play-button").click();
9
10   // Prüfen, ob Video abspielt
11   await ytFrame
12     .locator('.ytp-play-button[data-title-no-tooltip="Pause"]')
```

Übung 6 – Mobile und iframe Testing

Ziel: Mobile Emulation & iframe-Interaktion lernen.

Aufgaben:

1. **Mobile Emulation:**

- Mobile Geräte in `playwright.config.ts` definieren.
- Responsiven Navbar-Test schreiben (Desktop vs. Mobile).

2. **iframe Testing:**

- Test-HTML mit iframe erstellen.
- Mit `page.frameLocator()` auf iframe zugreifen und interagieren.

Zeit: 30 Minuten

Nachbesprechung: iframe Basics

Diskussion der Übungsergebnisse

Übung 6B – Komplexe iframe-Szenarien (Generische Beispiele)

Ziel: Verschachtelte & Cross-Origin iframes meistern.

Aufgaben:

1. Verschachtelte iframes testen: Mit `frameLocator()` durch Ebenen navigieren.
2. Cross-Origin iframes testen: Kommunikation mit `postMessage` implementieren und prüfen.
3. (Optional) Drittanbieter-Widget in iframe testen.

Zeit: 35 Minuten

Nachbesprechung: Advanced iframe

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-Selektor
13  await page.locator("my-custom-element >>> button.shadow-button").click();
14
```

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

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 Browsers
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:
20            name: playwright-report
21            path: playwright-report/
```

Best Practices & Zusammenfassung

- **Robuste Selektoren:** `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 & iframe:** 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: Selektoren, 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

- **Selektoren 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 Selektoren:** 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

Kontakt & Support

Kontaktmöglichkeiten

- E-Mail: daniel.sogl@thinktecture.com
- Twitter: [@sogldaniel](https://twitter.com/sogldaniel)
- LinkedIn: linkedin.com/in/sogldaniel
- GitHub: github.com/danielsogl

Weitere Informationen

- Website: danielsogl.de
- Blog: Artikel zu Playwright & Testing
- Workshops: Weitere Trainingsangebote
- Consulting: Individuelle Beratung

Vielen Dank!

Fragen? Kommentare? Anregungen?

