apobank.de

Nun wollen wir ein echtes Szenario testen

Ziel

Als Ziel haben wir vereinbart das Portal zu testen:

https://portal.banking.apobank.de/create-account/employee

https://portal.banking.apobank.de/create-account/student

Struktur

In diesem Fall empfehle ich die folgende Struktur:

--- apobank.de

------- create-account

---------- create_account_step_1.test.ts

---------- create_account_step_2.test.ts

---------- create_account_step_3.test.ts

Begründung: create-account ist der ganze Flow, und die einzelnen Tests sind die Seiten.

Struktur

Zur weiteren Strukturierung nutzen wir **`describe um TestSuites zu erstellen welche eine logische Gruppierung ermöglichen.`**

Für Vor- und Nachbedingungen (so zum Beispiel auch das Ausfüllen einer vorhergehenden Seite) nutzen wir die hooks:

- beforeEach
- beforeAll

afterEach/afterAll gibt es auch, aber ich sehe keine Anwendung.

Struktur

Weiter verwenden wir Tags, um Gruppen zu erstellen, die uns gewisse Eigenschaften signalisieren. Beispiele dafür sind @smoke oder @notimportant für die Cookie-Tests.

Kommentare zu best practices

`Tests so isoliert wie möglich gestalten`
Jeder Test sollte vollständig von anderen Tests isoliert sein und unabhängig mit eigenem lokalen Speicher, Sitzungsspeicher, Daten, Cookies usw. laufen. Die Testisolierung verbessert die Reproduzierbarkeit, erleichtert die Fehlersuche und verhindert kaskadierende Testfehler.

before/after Hooks sind hier sehr hilfreich.

Kommentare zu best practices

`Locators`:

Um End-to-End-Tests schreiben zu können, müssen wir zunächst Elemente auf der Webseite finden. Dazu können wir die in Playwright integrierten Locators verwenden. Locators verfügen über eine automatische Wartefunktion und Wiederholungsfunktion.

Kommentare zu best practices

Locators können verkettet werden, um die Suche auf einen bestimmten Teil der Seite einzugrenzen.

```
1 const product = page.getByRole('listitem').filter({ hasText: 'Product 2' });
```

```
1 await page
2     .getByRole('listitem')
3     .filter({ hasText: 'Product 2' })
4     .getByRole('button', { name: 'Add to cart' })
5     .click();
```

Kommentare zu best practices

`"user facing" Attribute gegenüber XPath- oder CSS-Selektoren bevorzugen`

Der DOM kann sich leicht ändern, sodass Tests, die von der DOM-Struktur abhängen, fehlschlagen.

Kommentare zu best practices

codegen verwenden:

Um Zeit zu sparen werden wir codegen verwenden, via VSCode, Chrome CRX oder ganz einfach via:

```
npx playwright codegen playwright.dev
```

Kommentare zu best practices

Web-First assertions:

```
// 👍
await expect(page.getByText('welcome')).toBeVisible();

// 👎
expect(await page.getByText('welcome').isVisible()).toBe(true);
```

Kommentare zu best practices

`Soft Tests:`

Wenn derTest fehlschlägt, gibt Playwright eine Fehlermeldung aus, die zeigt, welcher Teil des Tests fehlgeschlagen ist. Wirkönnen dies entweder in VS Code, im Terminal, im HTML-Bericht oder im Trace-Viewer sehen.

Wir können aber auch Soft Assertions verwenden. Diese beenden die Testausführung nicht sofort, sondern kompilieren und zeigen eine Liste der fehlgeschlagenen Assertions an, sobald der Test beendet ist.

Kommentare zu best practices

Soft Tests:

```
1 // Make a few checks that will not stop the test when failed...
2 await expect.soft(page.getByTestId('status')).toHaveText('Success');
3
4 // ... and continue the test to check more things.
5 await page.getByRole('link', { name: 'next page' }).click();
```

Wie geht es weiter

Mit all diesen Tipps sind wir nun bereit die Tests zu starten.

Wir haben die Tools bereit und das Wissen über die Strukturierungsmöglichkeiten von Playwright. Damit können wir beginnen **`hilfreiche`** Tests zu schreiben.

Kommentare zu best practices

Web-First assertions:

```
// 👎
expect(await page.getByText('welcome').isVisible()).toBe(true);
```

Use web first assertions such as `toBeVisible()` instead.

```
// 👍
await expect(page.getByText('welcome')).toBeVisible();
```

Ende

Das war alles für dieses Kapitel