

Automatyzacja testów funkcjonalnych aplikacji internetowych z użyciem **Playwright i TypeScript**

Mateusz Jabłoński



Kim jestem?

- ◆ programista od 2011 roku
- ◆ głównie: Javascript / Typescript / Java / dawniej: PHP
- ◆ szkoleniowiec / trener / mentor od 2016 roku
- ◆ prywatnie tata i mąż
- ◆ ostatnio również aktor w lokalnym teatrze i gracz papierowych RPG

Zaufaj naszemu doświadczeniu

18 lat

na rynku usług IT

29 560+

przeszkolonych osób

500+

klientów biznesowych

4 877+

zorganizowanych szkoleń i warsztatów

GWARANCJA JAKOŚCI USŁUG



98%

zadowolonych klientów*

* Średnia z ankiet poszkoleniowych przeprowadzanych wśród uczestników naszych szkoleń.

Edukacja na najwyższym poziomie

sages

Wiedza specjalistyczna dla branży IT

Oferujemy szeroki katalog szkoleń z technologii mainstreamowych i specjalistycznych, wschodzących i legacy. Zajęcia prowadzimy w trybie warsztatowym, a programy są oparte o praktyczne know-how. Specjalizujemy się w prowadzeniu dedykowanych szkoleń technologicznych, których agendę dostosowujemy do potrzeb naszych klientów i oczekiwania uczestników.

Wybitni eksperci

Od początku naszego istnienia przeszkołiliśmy dziesiątki tysięcy osób, co pomogło absolwentom podnieść konkurencyjność na rynku pracy i jakość projektów realizowanych na co dzień. Nasze szkolenia prowadzą najlepsi trenerzy, a nasze produkty są oparte na najnowocześniejszej technologii. Ich niezawodność i dopasowanie do potrzeb klientów są możliwe dzięki zespołowi składającemu się z wybitnych ekspertów i ekspertek, którzy/e są na pierwszej linii teorii i praktyki tworzenia i wdrażania innowacji technologicznych.

NASZE POZOSTAŁE MARKI EDUKACYJNE

STACJA.IT

kodo/amacz
by sages

Najlepsze standardy usług edukacyjnych

Stosujemy Standard Usługi Szkoleniowej Polskiej Izby Firm Szkoleniowych, a nasze usługi realizowane są na najwyższym poziomie, o czym świadczą stale powracający klienci oraz wdrożony certyfikat ISO 9001. Metodologia prowadzonych przez nas zajęć oparta jest na współczesnych narzędziach i dostosowana do potrzeb i oczekiwania klientów.

Ponadto jesteśmy firmą wpisaną do rejestru instytucji szkoleniowych w Wojewódzkim Urzędzie Pracy w Warszawie pod nr 2.14/00133/2019.

Zaufali nam najlepsi

Wśród naszych klientów są takie firmy jak Alior Bank, OLX Group, Bank Zachodni WBK, Orange Polska, Lufthansa i wiele innych.

STUDIA PODYPLOMOWE

Wspieramy organizację zaawansowanych kierunków studiów podyplomowych. Realizujemy zajęcia na kierunkach: Data Science, Big Data i Wizualna analityka danych, AI & Data Driven Business oraz User Experience Design – projektowanie doświadczeń cyfrowych.



Instytut Informatyki
Wydział Elektroniki i Technik Informacyjnych
Politechniki Warszawskiej



AKADEMIA
LEONA KOŽMIŃSKIEGO

Uwaga

Wszelkie materiały (treści tekstowe, wideo, ilustracje, zdjęcia itp.) wchodzące w skład szkoleń, kursów i webinarów organizowanych przez Sages są objęte prawem autorskim i podlegają ochronie na mocy Ustawy o prawie autorskim i prawach pokrewnych z dnia 4 lutego 1994 r. (tekst ujednolicony: Dz.U. 2006 nr 90 poz. 631). Kopiowanie, przetwarzanie, rozpowszechnianie tych materiałów w całości lub w części jest zabronione.



Ustalenia

- ▶ Cel i agenda
- ▶ Wzajemne oczekiwania
- ▶ Pytania i dyskusje
- ▶ Elastyczność
- ▶ Otwartość i uczciwość

Agenda, czyli co nas czeka?

1. Przygotowanie i konfiguracja projektu
2. TypeScript - przypomnienie
3. Wprowadzenie do Playwright
4. Playwright – zaawansowane funkcjonalności
5. Narzędzia i dobre praktyki
6. Podsumowanie

github.com/matwjablonski/playwright-ts-0126

Demo aplikacji



20 minut

zadanie nr 1

Zapoznanie z projektem

1. sklonuj projekt github.com/mateuszjablonski/playwright-ts-0126
2. zainstaluj lokalnie wszystkie zależności
3. uruchom aplikację (`npm run dev`) i zaloguj się hasłem `admin123`
4. przetestuj ręcznie wszystkie funkcjonalności aplikacji:
 - ▶ dodaj nowe zadanie z różnymi priorytetami i datami
 - ▶ użyj filtrów (status, priorytet) i wyszukiwania
 - ▶ edytuj istniejące zadanie
 - ▶ przetestuj operacje masowe (zaznacz wszystkie, usuń ukończone)
 - ▶ sprawdź nawigację między stronami (Lista zadań ➔ O aplikacji)
5. przeanalizuj strukturę kodu w `src/` - znajdź główne komponenty i serwisy
6. sprawdź dane testowe w `public/data.json`

Cel: Poznanie architektury aplikacji przed rozpoczęciem pracy z TS i Playwright

mateuszjablonski.com × Sages



10

Po co testować kod?

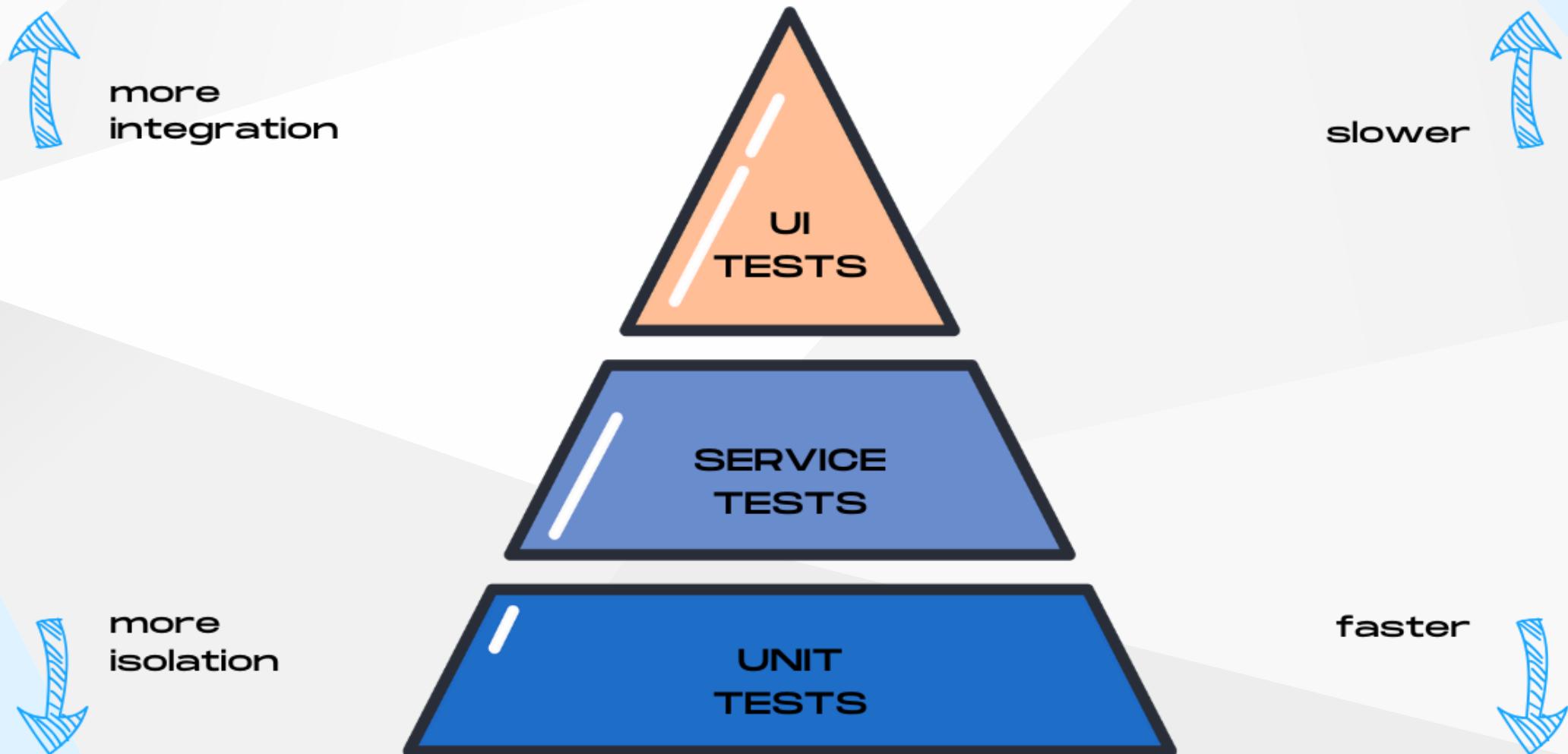
- ◀ szybsze wykrywanie błędów
- ◀ umożliwia zwinne programowanie (refaktoryzacja)
- ◀ testy to żywa i aktualna dokumentacja
- ◀ zapobiegają regresji
- ◀ kod możliwy do przetestowania najczęściej jest wielokrotnego użytku
- ◀ możemy ocenić wydajność poszczególnych części aplikacji
- ◀ możemy sprawdzić pokrycie kodu
- ◀ testy zmniejszają ukrytą złożoność kodu
- ◀ oszczędność czasu i pieniędzy

Rodzaje testów

- ◀ analiza statyczna
- ◀ testy jednostkowe
- ◀ testy integracyjne
- ◀ testy serwisów
- ◀ testy interfejsów
- ◀ e2e – testy funkcjonalne

Klasyczna piramida testów

- ◆ została opracowana w 2009 roku przez Mike'a Cohna
- ◆ model ten przedstawia hierarchię testów automatycznych w oprogramowaniu, zalecając większą liczbę szybkich i tanich testów jednostkowych u podstawy piramidy oraz mniejszą liczbę wolniejszych i droższych testów integracyjnych i interfejsu użytkownika na wyższych poziomach
- ◆ w 2012 Martin Fowler opublikował artykuł [Test Pyramid](#), w którym omówił i spopularyzował tę koncepcję wśród szerszej społeczności
- ◆ w 2018 roku Ham Vocke opublikował na blogu Fowlera artykuł [The Practical Test Pyramid](#), w którym przedstawił praktyczne zastosowanie tej koncepcji w rzeczywistych projektach programistycznych



Diamantowa strategia

- ◀ w aplikacjach frontendowych coraz częściej spotyka się strukturę diamentu, w której największy nacisk kładzie się na testy integracyjne
- ◀ w React (i podobnych bibliotekach) komponenty są małymi jednostkami logiki + UI - testowanie ich w izolacji często nie daje dużej wartości, bo są sztucznie oderwane od kontekstu
- ◀ unit testy sprawdzają się raczej w logice (np. helpery, walidatory), ale stanowią mniejszą część kodu niż w backendzie
- ◀ testy lepiej odzwierciedlają sposób działania aplikacji z perspektywy użytkownika
- ◀ dobrze pasuje do podejścia **testuj zachowanie, nie implementację**

Testy jednostkowe

- ▶ technika testowania oprogramowania, która polega na sprawdzaniu poszczególnych, najmniejszych jednostek kodu źródłowego, takich jak funkcje, metody czy klasy
- ▶ celem testów jednostkowych jest zapewnienie, że każda z tych jednostek działa zgodnie z oczekiwaniami
- ▶ są często wykorzystywane w ramach Continuous Integration/Continuous Deployment

Test jednostkowy to kod wykonujący inny kod w kontrolowanych warunkach w ramach jednego procesu w pamięci, w celu weryfikacji (bez interwencji programisty), że testowana logika działa w ścisłe określony sposób.

Test doubles, czyli podstawienia

Są to zamienniki prawdziwych obiektów używane podczas testów, których celem jest kontrola zachowania testowanego kodu i odizolowanie go od zależności zewnętrznych, np. baz danych, API, systemów plików.

Wyróżniamy kilka rodzajów: dummy, fake, stub, spy i mock.



Dummy – statysta z tytułu sceny

- ▶ jest przekazywany, ale nigdy nie jest używany
- ▶ służy zazwyczaj jedynie do wypełnienia listy parametrów, np. kiedy metoda wymaga jakiegoś argumentu, ale test tego argumentu nie potrzebuje

Przykładowy kod:

```
const dummyUser = null;  
service.doSomething(dummyUser);
```



Fake – improwizujący aktor w wersji beta

- ma działającą implementację, ale zazwyczaj wykorzystuje upraszczenia, które nie nadają się do produkcji (np. baza danych w pamięci)

Przykładowy kod:

```
class InMemoryUserRepository {  
    private users = [];  
    add(user) { this.users.push(user); }  
    findById(id) { return this.users.find(u => u.id === id); }  
}
```



Stub – aktor jednego zdania

- ▶ zwraca z góry zaprogramowane odpowiedzi na zapytania w czasie testu
- ▶ zwykle ignoruje wszystko co nie zostało zaprogramowane

Przykładowy kod:

```
const stubService = {  
  getUser: jest.fn().mockReturnValue({ id: 1, name: 'Anna' })  
};
```



Spy – krytyk teatralny z notatnikiem

- ▶ rejestruje informacje o tym, jak został użyty
- ▶ przykładem może być serwis email, który zlicza liczbę wysłanych wiadomości

Przykładowy kod:

```
const sendMail = jest.fn();
sendWelcomeEmail(user, sendMail);

expect(sendMail).toHaveBeenCalledTimes(1);
```



Mock – reżyser z gotowym scenariuszem

- ▶ zaprogramowany z oczekiwaniemi, które określają specyfikację interakcji
- ▶ może zgłosić wyjątek przy nieoczekiwanyem wywołaniu
- ▶ jest weryfikowany po zakończeniu testu, by sprawdzić, czy otrzymał wszystkie oczekiwane wywołania

Przykładowy kod:

```
const mock = sinon.mock(service);
mock.expects("sendEmail").once().withArgs(userEmail);

service.sendEmail(userEmail);

mock.verify();
```



Podstawowe biblioteki w JS

- ▶ Jest
- ▶ Jasmine
- ▶ Mocha
- ▶ Ava
- ▶ Vitest



Jest

- ▶ najpopularniejsza biblioteka do testowania jednostkowego w ekosystemie JS
- ▶ łatwa konfiguracja
- ▶ popularny test runner
- ▶ wbudowane mocki i asercje
- ▶ świetna integracja z React / NodeJS

Przykładowy kod:

```
test('Dodawanie dwóch liczb', () => {
  function add(a, b) {
    return a + b;
  }
  expect(add(2, 3)).toBe(5);
});
```

Jasmine

- ▶ dobry wybór dla Angular
- ▶ starsza biblioteka (wydana w 2010 roku), ale bardzo popularna w projektach Angularowych
- ▶ prosty interfejs
- ▶ nie wymaga konfiguracji
- ▶ test runner

Przykładowy kod:

```
describe('Funkcja add', () => {
  it('powinna poprawnie dodawać liczby', () => {
    function add(a, b) {
      return a + b;
    }
    expect(add(2, 3)).toBe(5);
  });
});
```

Vitest

- ▶ nowoczesna biblioteka testowa, będąca szybszą alternatywą dla Jest
- ▶ oparta na Vite
- ▶ kompatybilność z Jest (można używać expect, describe, it/test)
- ▶ obsługa TypeScript i ESM bez dodatkowej konfiguracji

Przykładowy kod:

```
import { test, expect } from 'vitest';

test('Dodawanie dwóch liczb', () => {
  function add(a, b) {
    return a + b;
  }
  expect(add(2, 3)).toBe(5);
});
```

Mocha + Chai

- ◆ lekka i elastyczna biblioteka do testowania
- ◆ możliwość używania różnych bibliotek do asercji (np. Chai)
- ◆ elastyczna konfiguracja, dobry wybór dla backendu

Przykładowy kod:

```
const { expect } = require('chai');

describe('Funkcja add', () => {
  it('powinna poprawnie dodawać liczby', () => {
    function add(a, b) {
      return a + b;
    }
    expect(add(2, 3)).to.equal(5);
  });
});
```

Ava

- ◆ minimalistyczne i szybkie testy
- ◆ równoległe wykonywanie testów

Przykładowy kod:

```
import test from 'ava';

test('Dodawanie dwóch liczb', t => {
  function add(a, b) {
    return a + b;
  }
  t.is(add(2, 3), 5);
});
```

Testowanie snapshotowe

- ▶ testy snapshotów są przydatnym narzędziem, gdy chcemy być pewni że nasz UI nie zmienił się w sposób nieoczekiwany
- ▶ typowy test snapshotowy renderuje komponent UI i porównuje go z kodem snapshot'u (artefakt), który najczęściej jest przechowywane razem z testami
- ▶ artefakt snapshot'u powinien być commitowany razem ze zmianami w kodzie i przechodzić review tak samo jak pozostałe części naszego kodu
- ▶ aby zaktualizować snapshot uruchom: `jest -u` lub `jest --updateSnapshot`

Dbaj o swoje snapshoty!

Traktuj snapshot tak jak kod

Commituj snapshot'y i wykonuj na nich proces code review

Testy snapshotowe powinny być deterministyczne

- ◆ Uruchamianie tych samych testów wiele razy na komponentie, który nie zmienił się, powinno zawsze zwrócić ten sam wynik
- ◆ Jesteśmy odpowiedzialni za upewnienie się, że wygenerowany snapshot nie zawiera informacji specyficznych dla platformy czy innych przypadkowych danych

Używaj opisowych nazw dla snapshotów

Testy integracyjne

- ▶ testy integracyjne sprawdzają, czy różne moduły, komponenty lub systemy współpracują ze sobą zgodnie z oczekiwaniami
- ▶ testują komunikację między komponentami (np. w React/Vue)
- ▶ testują interakcję między modułami w aplikacji (np. Redux + komponenty)
- ▶ weryfikują przepływ danych przez poszczególne warstwy aplikacji
- ▶ występują po testach jednostkowych, ale przed testami walidacyjnymi

Testy E2E

- ▶ zwane także testami funkcjonalnymi
- ▶ testują wszystkie funkcjonalności naszej aplikacji z perspektywy użytkownika
- ▶ testy skupiają się użytkowniku naszej aplikacji
- ▶ najpopularniejsze narzędzia do pisania testów end-to-end w JS to Cypress, Playwright, Puppeteer czy Nightwatch

Testy statyczne

Jednym z najczęstszych źródeł błędów są literówki i nieprawidłowe typy danych.

Przekazanie ciągu znaków do funkcji oczekującej liczby lub literówka w wyrażeniu logicznym to **głupie** błędy, których nigdy nie mieliśmy popełnić, ale zdarzają się cały czas.

- ◆ Struktura kodu – Linter
- ◆ Poprawność kodu – Kompilator

Wzorzec Arrange / Act / Assert

Konwencja zwana **given / when / then** albo **arrange / act / assert**. Według niej testy dzielimy na 3 części:

	Odpowiednik	Opis
arrange	given	opis przykładowej sytuacji i warunków początkowych
act	when	akcja wykonywana na testowanym obiekcie klasy (jedno wywołanie)
assert	then	weryfikacja rezultatów

Reguła F.I.R.S.T.

zasada	Opis
Fast	Szybkie: testy (lub ich podzbiór) powinny uruchamiać się błyskawicznie (bo uruchamiane są często)
Independent	Niezależne: testy nie zależą od siebie, więc można je uruchamiać w dowolnej kolejności i kombinacji
Repeatable	Powtarzalne: uruchamiane n razy dają ten sam wynik (pomaga izolować błędy i wspiera automatyzację)
Self-checking	Samosprawdzalne: test automatycznie wykrywa, czy przeszedł, bez konieczności sprawdzania przez człowieka
Timely	Na czas: pisane w tym samym czasie co kod (w TDD – pisane jako pierwsze!)

Co to jest reguła Z.O.M.B.I.E.S.?

To akronim stosowany przy testowaniu kontrolerów (np. w testach API, REST, MVC) - pochodzi od zasad pisania dobrych testów jednostkowych kontrolerów, gdzie testujesz tylko to, co należy.

Litera	Znaczenie	Co robi?
Z	Zero	Upewnij się, że test nie zostawia po sobie stanu
O	One	Jeden test = jeden przypadek
M	Mock	Zamień zależności (np. serwisy) na mocki
B	Before	Przygotuj dane testowe z wyprzedzeniem
I	Isolate	Testuj tylko logikę kontrolera, nic poza tym
E	Exercise	Faktycznie wywołaj metodę/testuj funkcję
S	State/Side Effect	Sprawdź rezultat/stany końcowe/efekty uboczne

TDD

- ◆ Test Driven Development
- ◆ testy bazują na cyklu: **Red / Green / Refactoring**
 - ◆ **Red** – w tej fazie myślimy o tym co chcemy zaprogramować
 - ◆ **Green** – ta faza pokrywa odpowiedź na pytanie jak zrobić, żeby przeszły testy
 - ◆ **Refactor** – myślimy o tym jak poprawić napisaną w poprzednich fazach implementację
- ◆ w tym podejściu do testowania testy piszemy zanim zostanie napisana rzeczywista implementacja

START



RED PHASE



refaktoryzacja
implementacji
i testowanie

GREEN PHASE



pisanie
minimalnej
implementacji



TDD: Red

- ▶ czerwona faza to zawsze punkt wyjściowy od którego zaczynamy testowanie
- ▶ celem tej fazy jest napisanie testów, które informują nas o tym co powinna robić implementacja funkcjonalności (definiują cel funkcjonalności)
- ▶ na końcu tej fazy wszystkie testy powinny być czerwone (funkcjonalność jeszcze nie istnieje)
- ▶ testy będą przehodzić dopiero kiedy wymagania zostaną pokryte

TDD: Green

- ▶ w zielonej fazie piszemy kod implementacji, tak aby testy napisane w fazie czerwonej przeszły (stały się zielone)
- ▶ celem jest znalezienie rozwiązania, bez głębszego wchodzenia w optymalizację naszej implementacji
- ▶ na końcu tej fazy „jesteśmy na zielono”. Możemy zacząć myśleć o optymalizacjach naszego kodu

TDD: Refaktoryzacja

- ▶ w tej fazie nadal „**jesteśmy na zielono**”
- ▶ istotna faza to moment na zastanowienie się jak napisać nasz kod lepiej lub bardziej wydajnie
- ▶ podczas refaktoryzacji nie chodzi o zmiany wyniku naszej funkcji, a o osiągnięcie tego samego efektu przy bardziej opisowym lub szybszym kodzie

Pytania, które powinniśmy sobie zadać kiedy myślimy o refaktoryzacji:

- ▶ Czy mogę poprawić moje testy, aby były lepiej dopasowane?
- ▶ Czy moje testy zwracają wartościowy feedback na temat funkcjonalności?
- ▶ Czy moje testy są odpowiednio odizolowane od innych testów / modułów?
- ▶ Czy mogę zredukować duplikację kodu w moich testach lub w samej implementacji?
- ▶ Czy moja implementacja może być czytelniejsza?
- ▶ Czy moja implementacja może być bardziej wydajna?

Regresja

Regresja to sytuacja, w której coś, co wcześniej działało poprawnie, przestaje działać po wprowadzeniu zmian w kodzie. Najczęściej może wystąpić po: dodaniu nowej funkcji, poprawce błędu, aktualizacji zależności, refaktoryzacji.

Aby wykryć regresję w trakcie rozwoju programu, należy przeprowadzać testy regresji.

Zwykle testy regresji polegają na ponownym uruchomieniu zestawu testów, które wcześniej kończyły się powodzeniem.

TypeScript

Czym jest Typescript?

- ▶ język programowania
- ▶ nadzbiór (superset) dla języka Javascript
- ▶ dodaje statyczne i silne typowanie
- ▶ kompliuje się do Javascriptu
- ▶ można go używać z WebAssembly (AssemblyScript) oraz środowiskach takich jak Deno, ts-node
- ▶ wprowadzony na rynek przez Microsoft w roku 2012

zalety Typescriptu

- ▶ dodatkowa para oczu, która patrzy razem z nami na kod i szuka błędów
- ▶ świetne podpowiadanie składni oraz współpraca narzędzi na znacznie wyższym poziomie niż w czystym JS
- ▶ refaktoryzacja to nie problem (o spójność kodu dba kompilator)
- ▶ pewność zawartości stałych i zmiennych w programie
- ▶ typy w TS to dokumentacja kodu, która zawsze jest aktualna
- ▶ TS to sposób na zapisywanie kontraktów / ustaleń z BE i innymi komponentami w aplikacji

wady Typescriptu

- ◀ sprawdza typy tylko w czasie komplikacji
- ◀ bywa niechlujny (w szczególności w przypadku bibliotek open-source)
- ◀ nie rozwiązuje problemów JSa
- ◀ dodatkowa złożoność dla programistów (krzywa uczenia się) - zmiana nawyków z JSa oraz konieczność przyswojenie nowych koncepcji
- ◀ wymaga doinstalowania kompilatora do środowiska programistycznego

Dynamiczne typowanie

- ▶ nadawanie zmiennym typów w czasie działania programu
- ▶ zmienne nie posiadają typów przypisanych, otrzymują je dopiero w czasie komilacji
- ▶ zmienna w różnych momentach wykonywania może przechowywać wartości różnych typów
- ▶ to wartość niesie typ a nie zmienna
- ▶ języki dynamicznie typowane to: Javascript, Python, Ruby

Statyczne typowanie

- ▶ nadawanie zmiennym typów w czasie komilacji programu, poprzez ich deklarację
- ▶ raz nadany typ nie może zostać zmieniony
- ▶ pomaga uniknąć potencjalnych błędów związanych z typami (proste pomyłki typu zmiana string na number)
- ▶ ograniczenia nadane przez statyczne typowanie utrudniają zrobienie chaosu w kodzie

"Jeśli chodzi jak kaczka i kwacze jak kaczka, to musi być kaczką."

Duck typing

- ◀ zakłada że typ obiekt sprawdzamy na podstawie zawartych w nim pól, a nie deklaracji typu
- ◀ wykorzystywane powszechnie w JS, ale spotykane również w TS
- ◀ misja Apollo13 - duck typing w praktyce

Przykładowy kod:

```
const a: unknown;  
  
if (typeof a === "number") {  
}
```

C# i Java

- ▶ wykorzystują rzeczywisty, nominalny system typów
- ▶ każda wartość lub obiekt ma tylko jeden dokładny typ
- ▶ definicja typu znajduje się w klasie
- ▶ danego typu nie możemy używać poza klasą (chyba, że zachodzi proces dziedziczenia lub implementacji publicznego interfejsu)
- ▶ typy są dostępne również na etapie wykonywania kodu
- ▶ typy są powiązane poprzez deklaracje a nie poprzez wartości

...typy w Typescript?

- ▶ Typescript wykorzystuje typowanie strukturalne
- ▶ lepiej myśleć o typie w TS jako o zestawie wartości, a wartość o danym typie musi po prostu pokryć część struktury
- ▶ jedna wartość nie musi być przypisana do jednego typu, jednocześnie może spełniać wiele typów
- ▶ w TS obiekty nie są jednego typu, **obiekt spełniający dany interfejs może być użyty nawet tam, gdzie nie było deklarowanej relacji między nim a interfejsem, który spełnia**

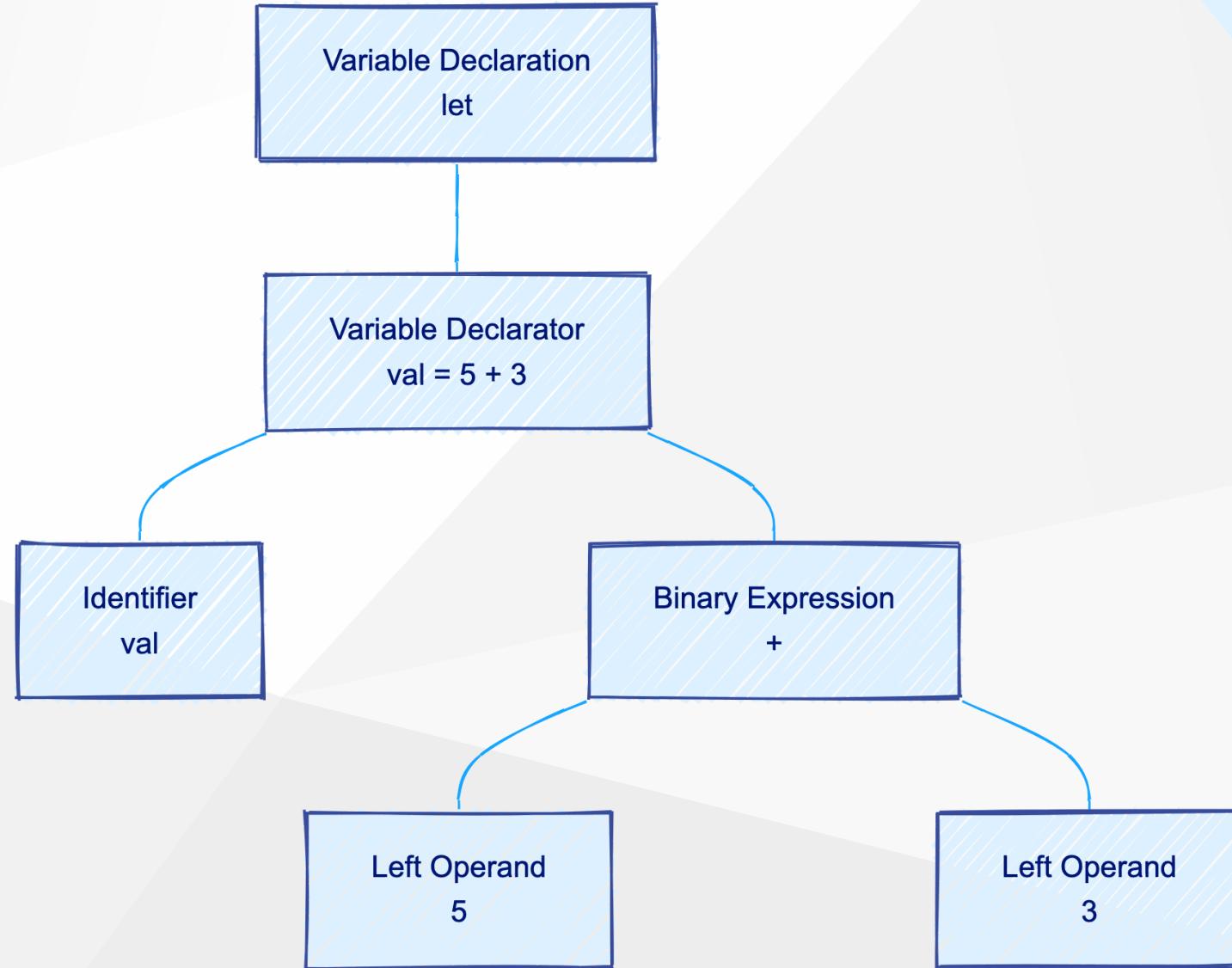
Kompilacja do Javascript

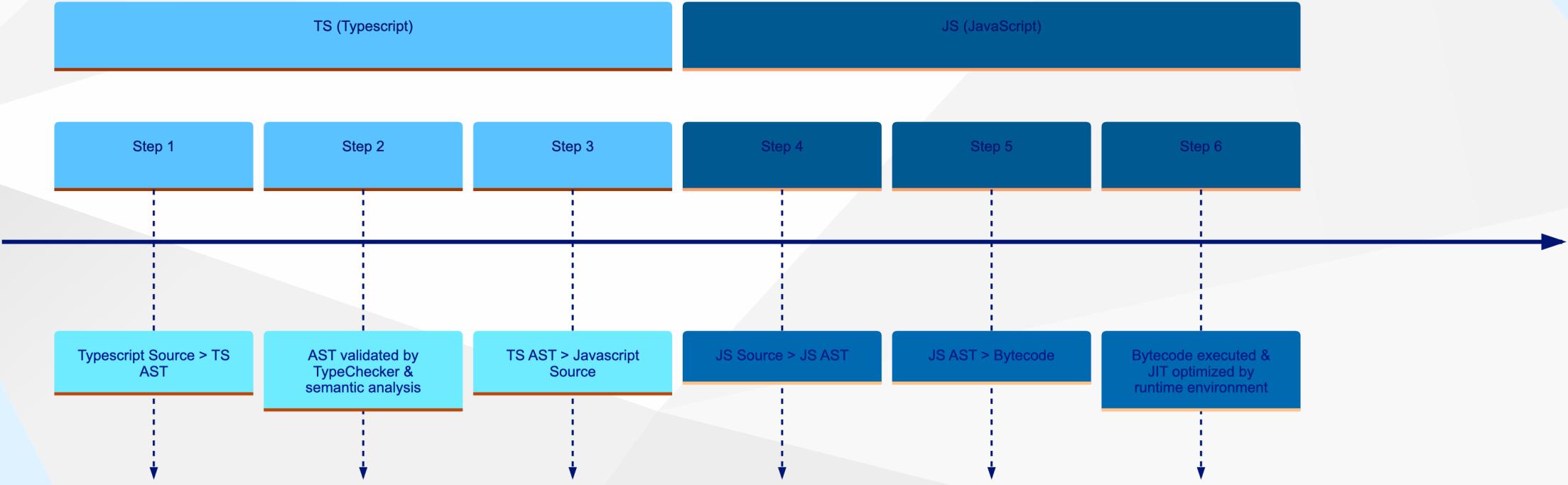
- ◀ sprawdzanie typów odbywa się przed komplikacją kodu do Javascript
- ◀ system typów jest usuwany przekształcania Typescript AST (abstract syntax tree) do kodu Javascript
- ◀ dodane typy nie mają wpływu na wygenerowany kod Javascript
- ◀ ponieważ proces sprawdzania typów odbywa się tak wcześnie i typy nie są brane pod uwagę na kolejnym etapach komplikacji
 - **prawidłowy kod Javascript jest prawidłowym kodem Typescript**

AST

- ◆ AST czyli Abstract Syntax Tree, to abstrakcyjna reprezentacja struktury kodu źródłowego
- ◆ reprezentuje kod w postaci drzewa, gdzie węzły odpowiadają konstrukcjom języka programowania
- ◆ AST składa się z węzłów (nodes) oraz krawędzi
- ◆ **węzły** reprezentują różne elementy kodu (np. funkcje, zmienne, operatory)
- ◆ **krawędzie** łączą węzły w logiczny sposób, odwzorowując strukturę kodu
- ◆ AST jest używane przez kompilatory do generowania kodu wynikowego, bundlery do przekształcenia kodu źródłowego, a także takie narzędzia jak lintery do analizy statyczne kodu

```
let val = 5 + 3
```





Mapowanie kodu źródłowego

- ▶ Typescript umożliwia tworzenie plików typu sourceMap
- ▶ podczas komplikacji do JS powstają wówczas pliki .js.map lub .jsx.map
- ▶ pliki map ułatwiają debuggowanie kodu TS poprzez powiązanie plików .ts z plikami .js

15 minut

zadanie nr 2

Przejście na TS

- zainstaluj wymagane zależności:

Przykładowy kod:

```
npm install --save-dev typescript @types/node
```

- utwórz plik **tsconfig.json** (lub wygeneruj go automatycznie)

Przykładowy kod:

```
npx tsc --init
```

- zmień plik konfiguracyjny Vite na **vite.config.ts**



"...żaden system typów nie może całkowicie wyeliminować występowania bugów, może tylko wykazać ich obecność..."

Podstawowe typy

- ▶ boolean
- ▶ number
- ▶ string
- ▶ symbol
- ▶ bigint
- ▶ null
- ▶ undefined
- ▶ object
- ▶ Array

a ponad to...

- tuple
- enum
- void
- any
- Object
- never
- unknown

boolean

- ▶ przyjmuje wartość logiczną true lub false

number

- ◆ przyjmuje wartość liczbową (liczby zmiennoprzecinkowe)
- ◆ możliwe jest używanie literałów heksadecymalnych, oktalnych i binarnych

Przykładowy kod:

```
const a: number = 123 + 0x0e2f + 0b1100 + 0o710;  
// 123 + 3631 + 12 + 456 = 4222
```

string

- ◆ ciągi znaków podawane w pojedynczym cudzysłowie ('), podwójnym (") oraz template string (`)
- ◆ pojedynczy i podwójny mają to samo zastosowanie
- ◆ template string działa tak samo jak w dokumentacji JS - obsługa wielu linii oraz przekazywanie do nich wywołań funkcji i zmiennych

symbol

- ▶ gwarantuje unikalność
- ▶ zostały wprowadzone do ES2015
- ▶ symbole nie są enumerable

bigint

- ▶ przeznaczony do reprezentowania dowolnie dużych liczb całkowitych
- ▶ mieszanie typów number i bigint w JS kończy się rzuceniem wyjątku, a w TS błędem komilacji

null i undefined

- ◆ typy przydatne do opisywania wartości opcjonalnych albo "nullowalnych"

Array

- ◆ typ opisuje tablice
- ◆ jest typem złożonym - wszystkie typy złożone są porównywane przez referencję a nie wartość co oznacza, że `[] === []` zwróci `false`
- ◆ Array jest typem generycznym `Array<T>`
- ◆ istnieją dwa sposoby zapisu: `Array<T>` oraz `T[]`

tuple (krotka)

- ◆ tupla to skończona lista elementów, w TS jest to tablica o określonej na sztywnie liczbie elementów o określonych typach na konkretnych pozycjach

Przykładowy kod:

```
const interval1: [number, string] = [0, "hour"]; // OK
const interval2: [number, string] = []; // Błąd
```

enum

- enum to zbiór nazwanych wartości
- enumy w TS są domyślnie liczbami rozpoczynającymi się od 0
- możliwe jest tworzenie enumów o wartościach będącymi stringami

Przykładowy kod:

```
enum UserRole {  
    Admin = "admin",  
    User = "user",  
}  
const role: UserRole = UserRole.Admin; // admin
```

void

- ◆ oznacza brak wartości
- ◆ używany do oznaczania funkcji, które nic nie zwracają
- ◆ do zmiennej o typie void można przypisać tylko wartość undefined

Przykładowy kod:

```
function fn1(): void {}  
function fn2(): void { return; }  
function fn3(): void { return undefined; }
```

any

- ◆ domyślny typ
- ◆ oznacza dowolną wartość
- ◆ użycie `any` to zaprzeczenie statycznego i silnego typowania - taki wentyl bezpieczeństwa
- ◆ jeśli użyjemy `any` do opisania obiektu to wszystkie jego pola i metody również będą domyślnie typu `any`
- ◆ `any` oznacza że typ Cię nie obchodzi, a nie że go nie znasz

never

- ▶ typ opisujący wartość, która nigdy nie wystąpi
- ▶ przydatny do funkcji, które nigdy nic nie zwracają - tylko rzucają wyjątek lub które się zawieszają (infinite loop)
- ▶ pozwala też wyłapać sytuacje, które nie powinny się zdarzyć

unknown

- ◀ typ powstał by przestano nadużywać typu any
- ◀ reprezentuje typ, który jest nieznany
- ◀ do unknown mogę przypisać dowolną wartość, ale nie mogą nic z niego odczytać
- ◀ nie możemy przypisać zmiennej o typie unknown do innej zmiennej z określonym jawnie typem

object i Object

- ◆ typ **object** reprezentuje wszystko co nie jest typem prymitywnym (string, number, bigint, symbol, null i undefined)
- ◆ typ **Object** opisuje własności i metody, które są wspólne dla wszystkich obiektów w JS
- ◆ typ **Object** jest używany głównie do dziedziczenia

15 minut

zadanie 2: Typy podstawowe

- ▶ w pliku `types-basic.ts` zaimplementuj funkcje zgodnie z ich sygnaturami i opisami



type

- ▶ w TS możemy definiować własne typy
- ▶ przyjęło się że nazwy typów piszemy wielką literą
- ▶ własne typy są idealne do opisywania kształtu oczekiwanych obiektów
- ▶ pozwalają na tworzenie aliasów typów

Przykładowy kod:

```
type User = {  
    name: string;  
    age: number;  
}  
  
const user: User = { name: "Mateusz", age: 36 };
```

Interface

- ▶ podlega zasadom typowania strukturalnego
- ▶ nazywa i definiuje typ bez implementacji
- ▶ historycznie w TS był to jedyny sposób deklarowania typów
- ▶ interfejsy można rozszerzać (extends) oraz implementować (implements) w klasach
- ▶ interfejsy podlegają mechanizmowi łączenia deklaracji (declaration merging)
- ▶ interfejsy nie pozwalają na tworzenie aliasów (w przeciwieństwie do type)

Przykładowy kod:

```
interface Person extends Human {}  
class PersonClass implements Person {}
```

declaration merging

- ▶ dwa interfejsy o tej samej nazwie zostaną połączone i razem będą tworzyć dany typ
- ▶ mechanizm ten przydaje się, gdy deklaracje pochodzą z różnych źródeł, w szczególności gdy istnieje możliwość, że w naszym kodzie będziemy rozszerzać typ zewnętrznej biblioteki

Przykładowy kod:

```
interface A {  
    a: string;  
}  
  
interface A {  
    b: number;  
}
```

Inferencja

- ▶ inferencja typów to mechanizm wnioskowania typów
- ▶ TS wnioskuje typy na podstawie przypisania wartości do zmiennej, na podstawia ciała funkcji (zwracanych wartości, wykonywanych operacji)
- ▶ należy pamiętać, że inferencja czasami zawodzi i to do nas należy dookreślenie typu
- ▶ z inferencji należy korzystać uważnie, dobrze użyta pozwoli nam uniknąć wpisywania typów wszędzie i jednocześnie zagwarantuje wysoki poziom bezpieczeństwa

Inferencja dla const i let

Przykładowy kod:

```
let b = 123; // number
const a = 123; // '123'
```

Narrowing

- ▶ narrowing typów to mechanizm zawężania typów
- ▶ zawężanie odnosi się do procesu zmniejszania zakresu typu z szerszego typu do bardziej specyficznego typu w określonym bloku kodu lub kontekście
- ▶ jeżeli nie ustawimy odpowiednich zabezpieczeń TS będzie nas ostrzegał, gdy istnieje ryzyko, że typ jest zbyt szeroki
- ▶ do narrowing'u są wykorzystywane type guard'y, np: typeof czy instanceof

Kontrola przepływu

- ◀ analiza przepływu sterowania w TypeScript odnosi się do procesu, w którym kompilator TypeScript analizuje przepływ kodu i wyciąga wnioski na temat typów zmiennych w różnych punktach programu, najczęściej wykorzystując do tego konstrukcje if / else if
- ◀ umożliwia TypeScript'owi zawężenie typów zmiennych w określonych blokach kodu, w oparciu o warunki i instrukcje przepływu sterowania
- ◀ pomaga to uczynić system typów bardziej ekspresyjnym i wychwycić potencjalne błędy typów

Typy generyczne

- ◆ określane też generykami, typami polimorficznymi lub szablonami
- ◆ typ generyczny zakłada, że jakaś część typu może być sparametryzowana
- ◆ zapis `Typ<InnyTyp>` oznacza typ generyczny

Przykładowy kod:

```
const list: Array<string> = ["a", "b", "c"];  
  
type User = { name: string };  
const userList: User[] = [{ name: "Olek" }];  
  
type A<T> = { value: T, name: string }
```

Funkcje generyczne

Przykładowy kod:

```
const id = <T>(x: T): T => x;  
  
const result = id<number>(1)  
  
function value<R>(v: R): R {  
    return v;  
}
```

Generyczne klasy

Przykładowy kod:

```
class Person<T> {  
    equipment: Array<T> = [];  
  
    push(item: T) {  
        equipment.push(item);  
    }  
}
```

Generyczne interfejsy

Przykładowy kod:

```
interface Person<T> {  
    equipment: T[];  
    name: string;  
}
```

20 minut

zadanie 3: Modelowanie danych

- ▶ zamodeluj prosty typ danych Todo
- ▶ uzupełnij kod w pliku `ts-model.ts`, tak aby funkcje i typy spełniały podane wymagania



Unia (union type)

- ▶ unia to taki typ, który zawiera wartości wspólne dla typów składających się na nią
- ▶ unie oznaczamy za pomocą operatora `|`
- ▶ unia z `undefined` oznacza wartość opcjonalną, z `null` nullowalną

Przykładowy kod:

```
type A = { a: string; b: number; }
type B = { b: number; c: string; }
type U = A | B;
declare const union: U;
union.a; // błąd
union.b; // ok
```

Intersection type

- intersection to taki typ, który zawiera wszystkie wartości z typów składających się na niego
- intersection oznaczamy za pomocą operatora `&`
- przydatny do opisywania takich operacji jak `Object.assign` czy `extends`

Przykładowy kod:

```
type A = { a: string; b: number; }
type B = { b: number; c: string; }
type Inter = A & B;
declare const intersection: Inter;
intersection.a; // ok
intersection.b; // ok
```

Zarówno union type, jak i intersection type powinniśmy rozpatrywać jako zbiory typów.

Type guards

- ▶ type guard to blok kodu sprawdzający jaki jest typ przekazanej zmiennej
- ▶ istnieją zdefiniowane guardy, tj: `typeof` , `instanceof` , `in`
- ▶ oprócz powyższych do tworzenia type guardów można użyć porównań lub tworzyć własne funkcje sprawdzające

typeof

- ▶ jest wbudowanym operatorem Javascript
- ▶ nie porównujemy obiektu do instancji danej klasy, ale porównujemy typ (jego nazwę określoną w JS)

Przykładowy kod:

```
function printId(id: number | string) {  
  if (typeof id === "string") {  
    console.log("Twoje ID to litery: " + id.toUpperCase());  
  } else {  
    console.log("Twoje ID to liczba: " + id.toFixed(2));  
  }  
}
```

instanceof

- ▶ wbudowany w Javascript operator
- ▶ sprawdza czy nasza zmienna jest instancją wybranej przez nas klasy
- ▶ metoda ta opiera się na porównaniu łańcuchów prototypów (prototype chain) dla zmiennej oraz klasy

Przykładowy kod:

```
class Dog {  
  bark() {  
    console.log("Woof!");  
  }  
}  
  
class Cat {  
  meow() {  
    console.log("Meow!");  
  }  
}  
  
function makeSound(animal: Dog | Cat) {  
  if (animal instanceof Dog) animal.bark()  
  else animal.meow()  
}
```

- ◆ dostarcza informacji czy obiekt, który sprawdzamy, ma w sobie określone właściwości
- ◆ właściwość znajdująca się w naszym warunku, powinna być unikalna dla jednego konkretnego typu, aby zwiększyć wybór TSA do jednej opcji

Przykładowy kod:

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird) {
  if ("swim" in animal) {
    animal.swim();
  } else {
    animal.fly();
  }
}
```

15 minut

zadanie 3: Typy unii i Type Guards

- wykonaj zadania z pliku `ts-unions.ts`, tak aby spełniały podane wymagania



enum

- ◀ podczas tworzenia enumeracji nie musimy podawać wszystkich nowych wartości liczbowych - TS kolejne elementy zwiększy o jeden
- ◀ enumy są obiektami i tak powinniśmy je traktować
- ◀ koncepcyjnie obiekt, który powstaje po komplikacji zawiera właściwości zarówno jako kolejne liczby oraz przypisane im nazwy (taki podwójny zapis ułatwia konwersji w dwie strony)
- ◀ wartością enuma może być również wynik wykonania wyrażenia, ale wówczas kolejne wartości też muszą być opisane
- ◀ typy enumów są sprawdzane nominalnie, oznacza to, że nie możemy przypisać wartości z jednego enuma do zmiennej typu innego enuma - nawet gdy mają taką samą strukturę
- ◀ enumeracje są kompatybilne z obiektami a zatem tak gdzie są wymagane obiekty możemy używać również enumeracji

const enum

- ◆ zwykły enum jest kompilowany do JS w postaci “małego potworka” obiektu, a miejsca w których był użyty mają odwołanie do niego
- ◆ istnieje możliwość zapisu `const enum`, który nie pozostawia po sobie śladu w JS (pozostają tylko wartości i są opatrzone komentarzem)

Przykładowy kod:

```
const enum Role {  
    A,  
    B,  
    C,  
}  
const allowedRoles = [Role.A, Role.B]  
  
// po komplikacji  
const allowedRoles = [0 /* A */, 1 /* B */]
```

try...catch

Przykładowy kod:

```
try { /* ... */ }
catch (e: unknown) {
  e.message // errors
  if (typeof e === "string") {
    e.toUpperCase() // works, `e` narrowed to string
  } else if (e instanceof Error) {
    e.message // works, `e` narrowed to Error
  }
  // ... handle other error types
}
```

async

Przykładowy kod:

```
async function fetchData<T>(url: string): Promise<T> {
  const response = await fetch(url);

  if (!response.ok) {
    throw new Error(`Failed to fetch data: ${response.statusText}`);
  }

  const data: T = await response.json();
  return data;
}
```

Klasy

Przykładowy kod:

```
class Car {  
    private name: string;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
    public getName() {  
        return this.name;  
    }  
}  
  
const car = new Car('Audi');  
car.name; // Error: Property 'name' is private and only accessible within class 'Car'  
car.getName(); // Audi
```

Modyfikatory dostępu

- ▶ public
- ▶ protected
- ▶ private

Protected

Przykładowy kod:

```
class Car {  
    protected name: string;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
    public getName() {  
        return this.name;  
    }  
}
```

readonly w klasach

Przykładowy kod:

```
class Car {  
    protected name: string;  
    readonly minEcoRate = 0.8;  
  
    constructor(name: string) {  
        this.name = name;  
        this.minEcoRate = 10; // Type '10' is not assignable to type '0.8'  
    }  
    public getName() {  
        return this.name;  
    }  
}
```

implements

Przykładowy kod:

```
interface Thing {  
    setName(): void;  
    getName(): string;  
}  
  
class Box implements Thing {  
    private name: string;  
  
    public setName(): void {}  
  
    public getName(): string { return this.name; }  
}
```

Klasy abstrakcyjne

- ◆ mogą zawierać metody abstrakcyjne, które nie mają implementacji i które muszą być zaimplementowane przez podklasy
- ◆ mogą zawierać zwykłe metody (już w implementacji)
- ◆ metody abstrakcyjne zapewniają, że każda podklasa zapewnia własne specyficzne zachowanie dla metody

Przykładowy kod:

```
abstract class Animal {  
    abstract makeSound(): void; // Abstract method, no implementation  
  
    move(): void { console.log("Moving..."); }  
}
```

Pole prywatne ES

- ▶ w JS pola prywatne opisujemy za pomocą `#`
- ▶ pole prywatne w TS nie jest tym samym czym pole prywatne w JS
- ▶ przed wprowadzeniem `#` pola prywatne oznaczano `_`, ale nie były to pola prywatne znane z innych języków obiektowych

Pola i metody statyczne

- klasy pozwalają na tworzenie pól i metod statycznych, które mogą być wykonywane na klasach a nie na instancjach
- pola i metody statyczne poprzedzamy słowem kluczowym `static`
- deklaracja klasy tworzy dwa typy, np dla klasy `User` :
 - `typeof User` - zawiera wszystkie statyczne metody i pola klasy
 - `User` - oznacza instancję klasy

20 minut

zadanie 3: Klasy TS

- zaimplementuj klasy z pliku `ts-classes.ts`, tak aby spełniały podane wymagania



110

Playwright

111

TypeScript w pracy z Playwright

- ▶ Silne typowanie = mniej błędów w czasie pisania
- ▶ Bezpieczny dostęp do struktur danych (np. `APIResponse.json()` - automatyczne sprawdzanie struktury)
- ▶ Lepsze refaktoryzacje - zmiana nazwy elementu = mniej strachu o złamane testy
- ▶ Własne typy i helpery (np. `type UserRole = 'admin' | 'editor' | 'viewer'`)

WNIOSKI?

TypeScript zwiększa jakość testów i ich długoterminową czytelność – nawet jeśli na początku wymaga więcej pracy.

W połączeniu z Playwright to potężne narzędzie do tworzenia solidnych, skalowalnych testów E2E.

Czym jest Playwright i dlaczego warto go używać?

- ▶ **Playwright** - narzędzie do automatyzacji testów end-to-end stworzone przez Microsoft
- ▶ umożliwia testowanie aplikacji webowych w przeglądarkach Chromium, Firefox i WebKit
- ▶ powstał w 2020 roku jako alternatywa i ewolucja doświadczeń z Puppeteer
- ▶ tworzony przez byłych twórców Puppeteera (Google), którzy przeszli do Microsoftu
- ▶ dostępny jako Open Source na licencji Apache 2.0

Playwright vs Cypress vs Puppeteer

Cecha	Playwright	Cypress	Puppeteer
Twórca	Microsoft	Cypress.io	Google
Multi-browser	✓ Chromium, Firefox, WebKit	⚠ Tylko Chromium/WebKit (WebKit w beta)	✗ Tylko Chromium
Wsparcie TypeScript	✓ Świetne	✓ Dobre	⚠ Ograniczone
Automatyczne oczekiwanie	✓ Tak	✓ Tak	✗ Nie
Mockowanie sieci	✓ Tak	⚠ Tak (z ograniczeniami)	✗ Trudne

Cecha	Playwright	Cypress	Puppeteer
Testy API	✓ Wbudowane	✓ Wbudowane	✗ Trzeba osobno
Debugowanie (trace/video)	✓ Tak	✓ Tak	⚠️ Trzeba skonfigurować
Parallelizacja testów	✓ Tak	✓ (CI Premium)	✗ Brak wbudowanej
Architektura testów	Nowoczesna, modularna	All-in-one (monolityczna)	Niskopoziomowa kontrola

Dowolna przeglądarka | Dowolna platforma | Jedno API

Automatyczne oczekiwanie

Playwright czeka, aż elementy będą gotowe do interakcji przed wykonaniem akcji. Posiada również bogaty zestaw zdarzeń introspekcyjnych. Połączenie tych dwóch funkcji eliminuje potrzebę sztucznych opóźnień – głównej przyczyny niestabilnych testów.

Asersje zorientowane na web

Asersje Playwright zostały stworzone specjalnie dla dynamicznego webu. Sprawdzenia są automatycznie ponawiane, aż do spełnienia wymaganych warunków.

Śledzenie

Skonfiguruj strategię ponawiania testów, rejestruj ślad wykonania, nagrania wideo, zrzuty ekranu, aby eliminować błędy trudne do odtworzenia.

Potężne narzędzia

Codegen

Generuj testy poprzez nagrywanie swoich działań. Zapisuj je w dowolnym języku.

Inspektor Playwright

Inspekcja strony, generowanie selektorów, przechodzenie krok po kroku przez wykonanie testu, podgląd punktów kliknięć, eksploracja logów wykonania.

Trace Viewer

Rejestruj wszystkie informacje potrzebne do zbadania przyczyny niepowodzenia testu. Ślad Playwright zawiera nagranie testu, migawki DOM na żywo, eksplorator akcji, źródło testu i wiele więcej.

zacznijmy od pakietów

Przykładowy kod:

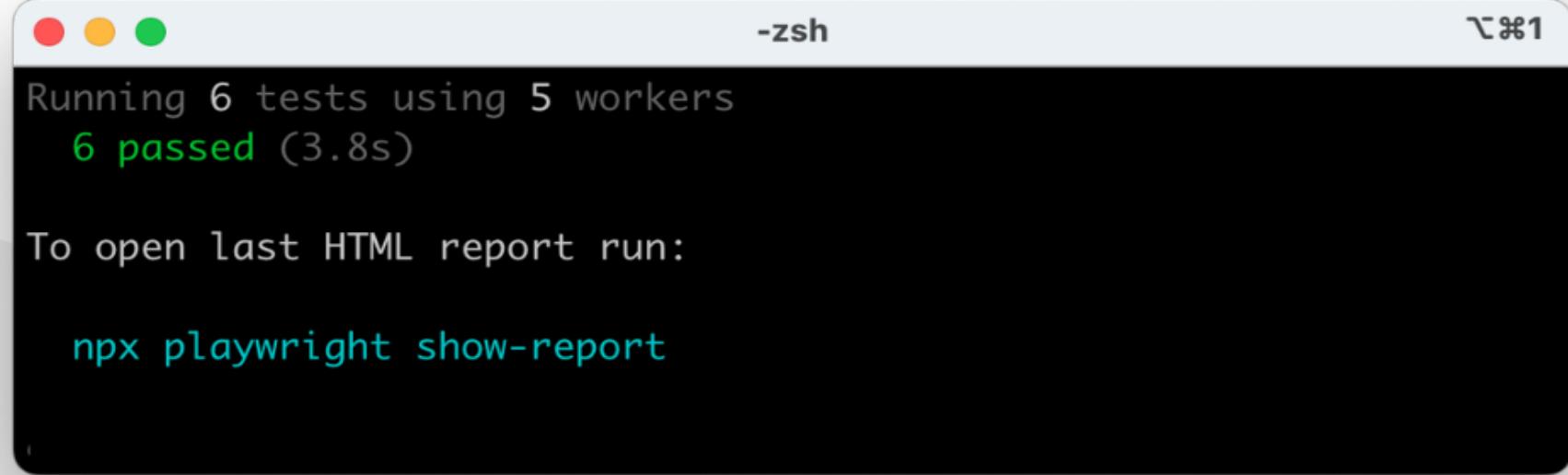
```
# zainstaluj Playwright wraz z binarkami do przeglądarek
npm i -D playwright

# zweryfikuj zainstalowaną wersję
npx playwright --version

# zainstaluj odpowiednie binaria przeglądarki
npx playwright install

# skonfiguruj playwright w projekcie
npm init playwright@latest

# uruchom pierwsze testy
npx playwright test
```



```
Running 6 tests using 5 workers
6 passed (3.8s)

To open last HTML report run:

npx playwright show-report
```

Struktura projektu

Katalog testów

- ↳ domyślnie `/tests` .
- ↳ pliki testowe powinny kończyć się na `.spec.ts` , `.spec.js` , `.test.ts` lub `.test.js`

Przykładowy kod:

```
/tests
└── login.spec.ts
└── homepage.spec.ts
```

playwright.config.ts

- ◆ główny plik konfiguracyjny projektu
- ◆ określa:
 - ◆ domyślną przeglądarkę i urządzenie,
 - ◆ timeouty, retry, lokalizację testów,
 - ◆ reporter (np. HTML, list, CI-friendly)
 - ◆ projekty testowe (np. dla różnych przeglądarek)

Przykładowy kod:

```
export default defineConfig({  
  testDir: './tests',  
  timeout: 30_000,  
  retries: 1,  
  use: {  
    headless: true,  
    baseURL: 'https://example.com',  
  },  
});
```

Projekty

- ◀ każdy projekt to osobne środowisko uruchomienia testów
- ◀ możemy testować w wielu przeglądarkach jednocześnie
- ◀ można dodać także konfiguracje mobilne, np. Pixel 5, iPhone 14, itp.

Przykładowy kod:

```
import { defineConfig, devices } from '@playwright/test';
export default defineConfig({
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },
    {
      name: 'firefox',
      use: { ...devices['Desktop Firefox'] },
    },
  ],
});
```

Uruchomienie testów dla projektu firefox

Przykładowy kod:

```
npx playwright test --project=firefox
```

Konfiguracja w package.json

Przykładowy kod:

```
{  
  "scripts": {  
    "test:dev": "npx playwright test --project=chromium",  
    "test:ui": "npx playwright test --ui",  
    "test": "npx playwright test"  
  }  
}
```

Przykładowy kod:

```
npm run test:dev
```

15 minut

zadanie nr 4

Konfiguracja środowiska testowego

- ▶ zainstaluj Playwright

Przykładowy kod:

```
# zainstaluj Playwright wraz z binarkami do przeglądarek
npm i -D playwright
# zweryfikuj zainstalowaną wersję
npx playwright --version
# zainstaluj odpowiednie binaria przeglądarki
npx playwright install
# skonfiguruj playwright w projekcie
npm init playwright@latest
# uruchom pierwsze testy
npx playwright test
```

- ▶ dodaj `scripts` w `package.json` do uruchamiania testów
- ▶ dodaj folder `tests/`



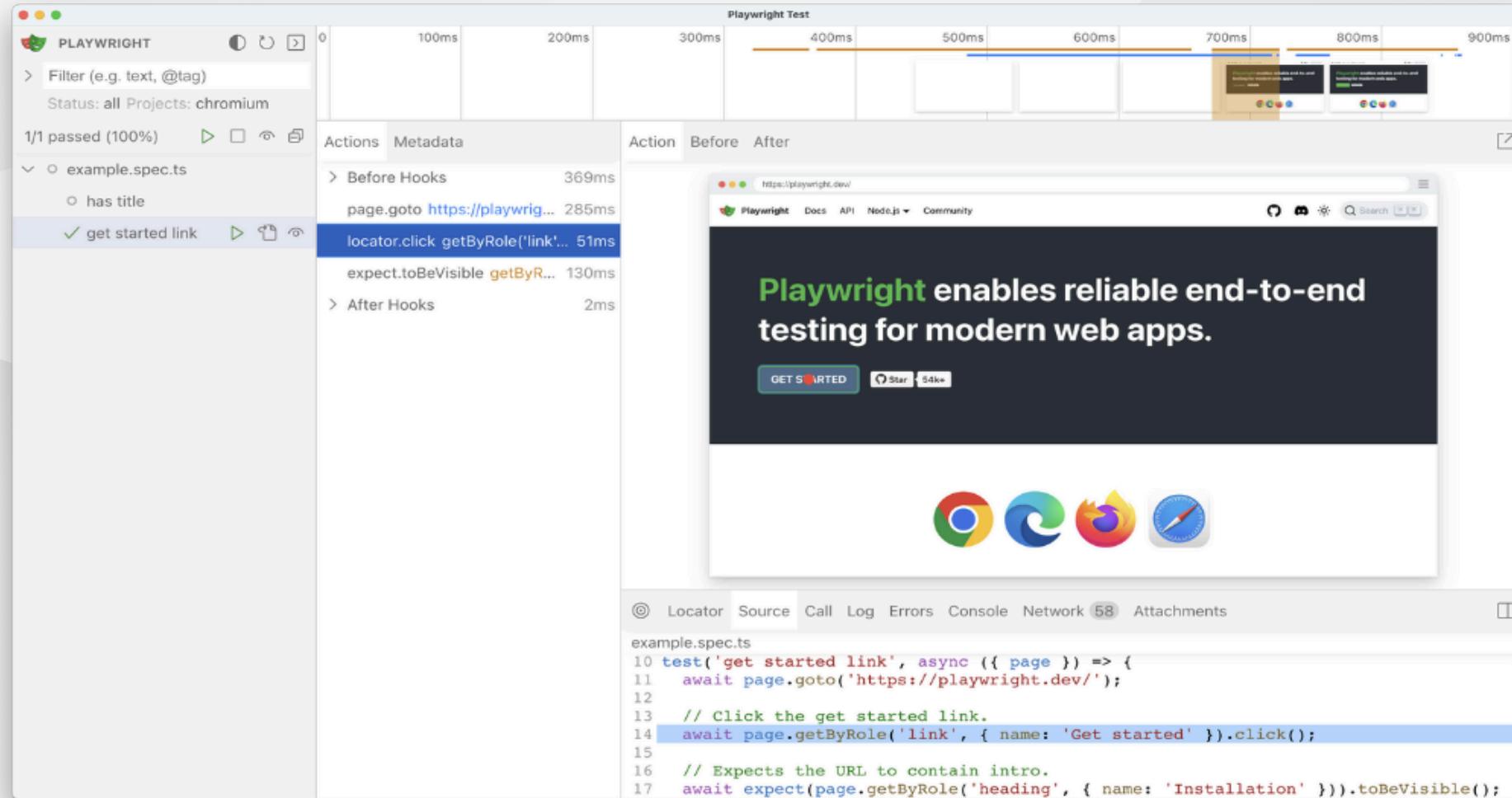
Playwright UI

Co oferuje interfejs graficzny?

- ▶ lista testów z możliwością filtrowania
- ▶ możliwość uruchamiania pojedynczych testów lub całych plików
- ▶ przegląd wyników testów (pass/fail)
- ▶ podgląd zrzutów ekranu, konsoli i trace'ów
- ▶ możliwość debugowania testów krok po kroku
- ▶ tryb interaktywny (z przyciskiem “Run”, “Debug”, itp.)

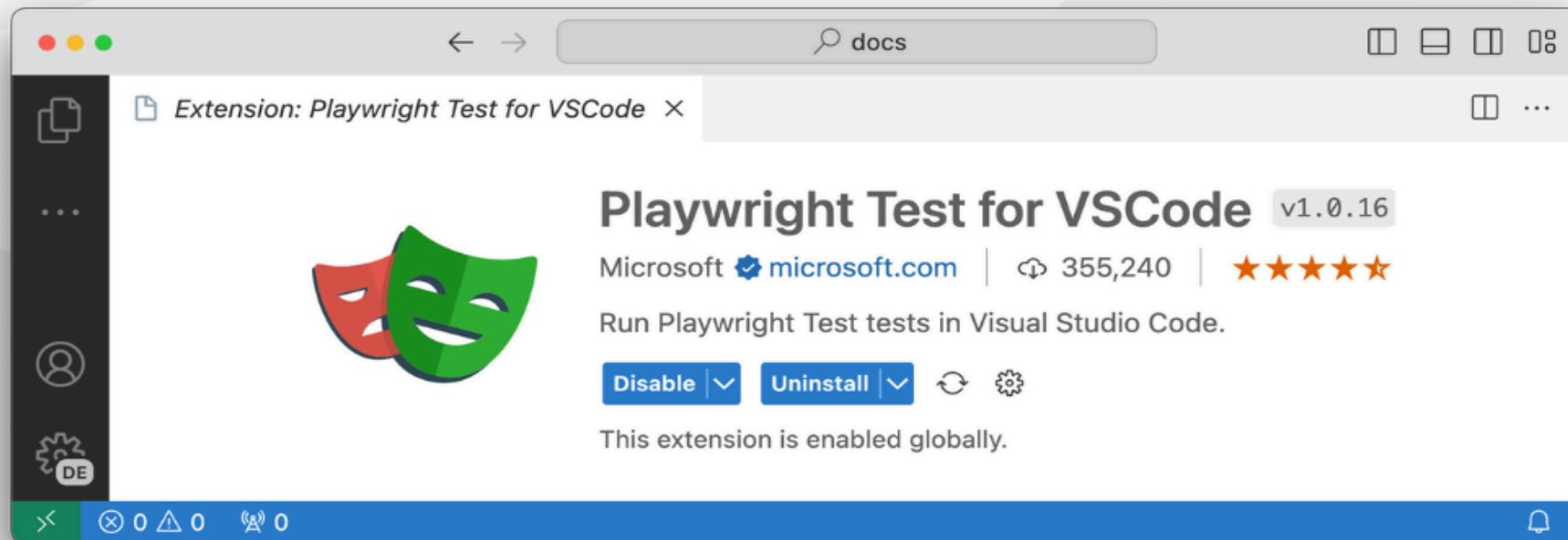
Przykładowy kod:

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



vs Code extension

<https://marketplace.visualstudio.com/items?itemName=ms-playwright.playwright>



129

The screenshot shows a code editor interface with a dark theme. On the left, there's a sidebar with various icons: a clipboard, a magnifying glass, a network connection, a grid, a flask, a person, and a gear. Below these are several buttons: 'Show browser' (which is checked), 'Show trace viewer', 'Pick locator', 'Record new', 'Record at cursor', 'Reveal test output', and 'Close all browsers'. The main area has a title bar with a back arrow, forward arrow, search bar containing 'docs', and window control buttons. The code editor itself has tabs for 'example.spec.ts' and 'docs'. The 'example.spec.ts' tab is active, showing a TypeScript file with two tests. The first test, 'has title', navigates to 'https://playwright.dev/' and checks if the title contains 'Playwright'. The second test, 'get started link', also navigates to the same URL and clicks a link labeled 'Get started', then checks if a heading is visible. The status bar at the bottom shows 'Ln 5, Col 1' and other file metadata.

```
tests > example.spec.ts > test('has title') callback
1 import { test, expect } from '@playwright/test';
2
3 test('has title', async ({ page }) => {
4     await page.goto('https://playwright.dev/'); - 356ms
5
6     // Expect a title "to contain" a substring.
7     await expect(page).toHaveTitle(/Playwright/); - 20ms
8 });
9
10 test('get started link', async ({ page }) => {
11     await page.goto('https://playwright.dev/');
12
13     // Click the get started link.
14     await page.getByRole('link', { name: 'Get started' }).click();
15
16     // Expects page to have a heading with the name of Installation.
17     await expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
18});
```

Debug – co to robi?

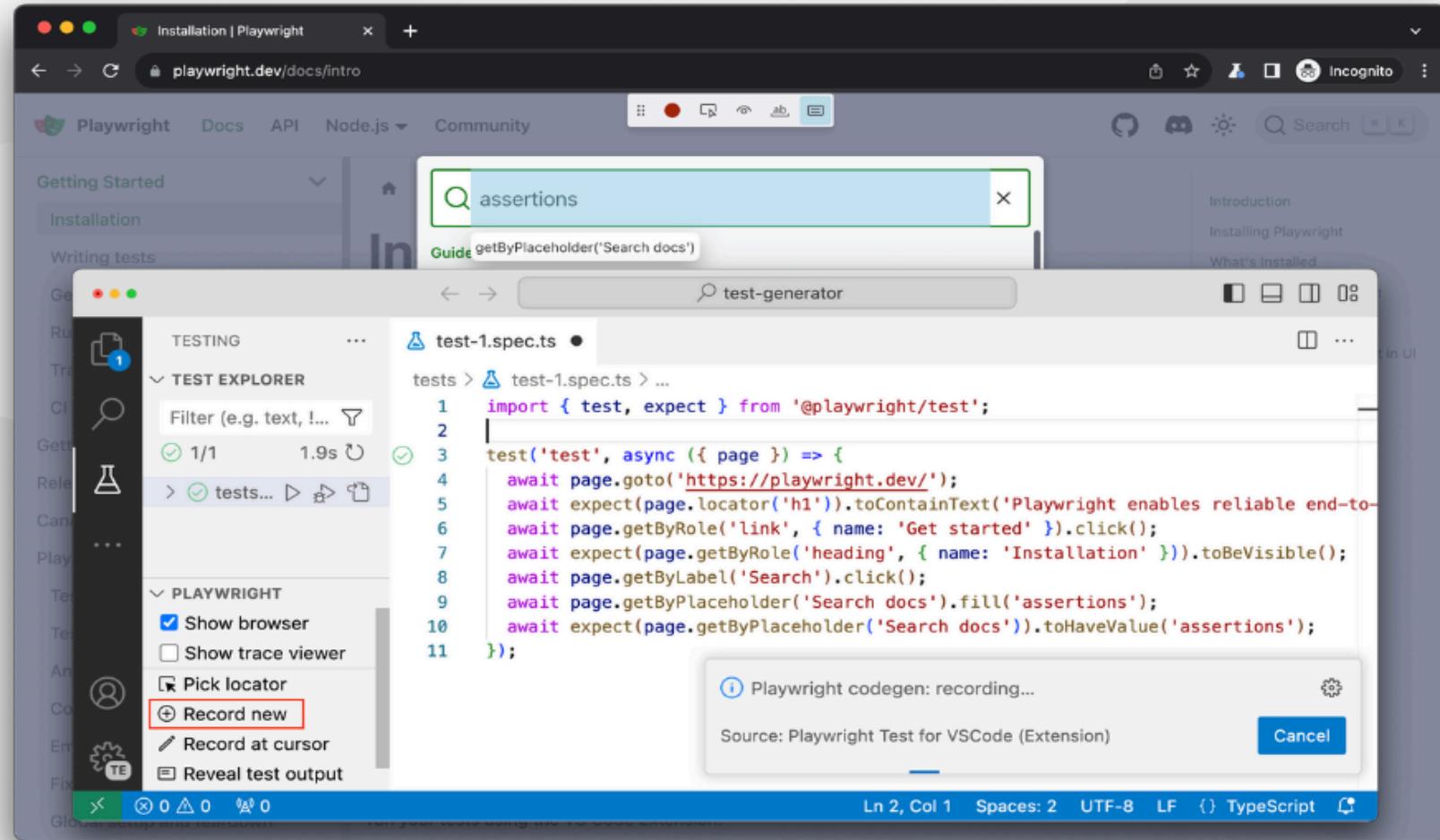
- ◆ przeglądarka uruchamiana jest w trybie headed (widocznym)
- ◆ test zatrzymuje się przed wykonaniem (`await page.pause()`), dzięki czemu można:
 - ◆ ręcznie kliknąć po stronie
 - ◆ analizować elementy
 - ◆ podglądać selektory
 - ◆ krokowo przechodzić przez akcje

Przykładowy kod:

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

The screenshot shows a browser-based IDE interface with the title bar "docs". The left sidebar contains icons for file, search, and navigation, along with sections for "TEST EXPLORER" and "PLAYWRIGHT". The "TEST EXPLORER" section shows 0/1 tests, 2.0s execution time, and a list of tests including "example.spec.ts 1.6s" which has a "has title" status of 858ms. A context menu is open over the line 14 of the "example.spec.ts" file, listing options: Run Test, Debug Test (which is selected), Execute Using Profile..., Peek Error, Reveal in Test Explorer, Add Breakpoint, Add Conditional Breakpoint..., and Add Logpoint... . The code editor shows a failing test for a "get started link" callback.

```
9
10  test('get started link', async ({ page }) => {
11    await page.goto('https://playwright.dev/');
12    const link = page.getByRole('link', { name: 'star' }).click();
13    expect(link).toBeVisible();
14    expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
15  });
16  // The page should have a heading with the name of Installation.
17  // Error: locator.click: Element is not visible.
18}
```



Pick locator

Otwiera przeglądarkę i pozwala kliknąć element, a następnie pokazuje i kopiuje najlepszy selektor (locator) dla tego elementu.

Zastosowanie:

- ↳ ręczne dobieranie najlepszych selektorów
- ↳ omijanie niestabilnych css/xpath
- ↳ debugowanie istniejących selektorów

WAŻNE

Uwaga!!! Pick locator działa tylko w projektach zainicjalizowanych przez Playwright CLI (z plikiem `playwright.config.ts`)

Dlaczego CSS/XPath bywają niestabilne?

Selektory CSS:

- 👉 mogą polegać na klasach, które się zmieniają (`.btn-primary` , `.v-123` , `.ng-star-inserted`)
- 👉 mogą łamać się przy zmianie struktury DOM
- 👉 trudne w utrzymaniu, jeśli zależą od konkretnego układu HTMLa

Przykładowy kod:

```
await page.locator('div > ul > li > a.btn-primary');
```

Ten selektor łamie się, jeśli dodasz span gdziekolwiek w drzewie DOM.

XPath (XML Path Language):

- ◆ bardziej precyzyjne, ale trudniejsze do czytania
- ◆ nieznośne w refaktoryzacji – wymagają utrzymania długich ścieżek
- ◆ często łamią się przy każdej zmianie HTMLa

Przykładowy kod:

```
await page.locator('//div[@class="card"]/following-sibling::div[2]');
```

Dobre praktyki

Testuj zachowania widoczne dla użytkownika

Testy automatyczne powinny weryfikować, że kod aplikacji działa dla użytkowników końcowych, i unikać polegania na szczegółach implementacyjnych, takich jak elementy, z których użytkownicy zazwyczaj nie korzystają.

Izoluj testy tak bardzo, jak to możliwe

Każdy test powinien być całkowicie odizolowany od pozostałych i uruchamiany niezależnie, z własnym local storage, session storage, danymi, ciasteczkami itp.

Unikaj testowania zewnętrznych zależności

Testuj tylko to, nad czym masz kontrolę. Nie testuj linków do zewnętrznych stron ani serwerów stron trzecich.

Testowanie z bazą danych

Jeśli pracujesz z bazą danych, upewnij się, że masz nad nią kontrolę. Testuj na środowisku staging i upewnij się, że dane się nie zmieniają.

Używaj lokalizatorów

Wyszukuj elementy na stronie za pomocą lokalizatorów. Lokalizatory automatycznie czekają i ponawiają próby.

Łańcuchowanie i filtrowanie

Lokalizatory można łańcuchować, aby zwiększyć wyszukiwanie do konkretnej części strony.

Preferuj atrybuty skierowane do użytkownika zamiast XPath lub selektorów CSS

Twoje drzewo DOM może się łatwo zmienić, więc uzależnianie testów od struktury DOM może prowadzić do ich awarii.

Generuj lokalizatory

Generator testów Playwrighta analizuje stronę i znajduje najlepszy lokalizator, dając priorytet roli, tekstowi i test-id.

Używaj asercji web-first

Asercje weryfikują, czy oczekiwany wynik zgadza się z rzeczywistością.

Fixtures

Fixture to przedkonfigurowany zasób, który jest automatycznie dostępny w testach i często przygotowywany oraz sprzątany przed i po każdym teście.

Fixture	Typ	Opis
page	Page	Izolowana strona dla tego testu.
context	BrowserContext	Izolowany kontekst przeglądarki dla tego testu. Fixture <code>page</code> również do niego należy. Dowiedz się, jak go skonfigurować.
browser	Browser	Przeglądarki są współdzielone między testami w celu optymalizacji zasobów.
browserName	string	Nazwa przeglądarki, w której obecnie działa test. <code>chromium</code> , <code>firefox</code> lub <code>webkit</code> .
request	APIRequestContext	Izolowana instancja APIRequestContext dla tego testu.

Dlaczego to ważne?

- ▶ nie musisz ręcznie tworzyć i zamykać zasobów
- ▶ testy są bardziej czyste i powtarzalne
- ▶ możesz definiować własne fixtures (np. zalogowany użytkownik, mock API)

Przykładowy kod:

```
test('dodaje zadanie', async ({ page }) => {
  await page.goto('http://localhost:3000');
  await page.locator('text=Dodaj zadanie').click();
});
```

Browser, Context i Page

Przykładowy kod:

```
// Tworzy nowy kontekst przeglądarki. Nie dzieli ciasteczek/pamięci podręcznej z innymi kontekstami.
const alice = browser.newContext()
const bob = browser.newContext()

// Otwiera nowe zakładki
const alicePage = await alice.newPage();
const bobsPage = await bob.newPage();

// Nawigacja
await bobsPage.goto('https://fakechat.com');
await alicePage.goto('https://fakechat.com');

// Nasłuch otwarcia nowej karty
const messagePopupPage = bobsPage.waitForEvent('popup');

// Wyślij wiadomość od Alice
await alicePage.getByRole('link',{ name:'Message Bob' }).click()

const popup = await messagePopupPage;

// Odbierz wiadomość od Alice
expect(await popup.getText('message from alice')).toBeVisible()
```

Lokalizatory Web First

- ◆ `page.getByRole()` – na podstawie atrybutów dostępności ARIA.
- ◆ `page.getText()` – na podstawie treści tekstowej.
- ◆ `page.getLabel()` – kontrolka formularza na podstawie powiązanej etykiety.
- ◆ `page.getPlaceholder()` – na podstawie tekstu zastępczego (placeholder).
- ◆ `page.getAltText()` – na podstawie tekstu alternatywnego (alt).
- ◆ `page.getTitle()` – element na podstawie atrybutu `title`.
- ◆ `page.getTestId()` – element na podstawie atrybutu `data-testid` (można skonfigurować inne atrybuty)

Generuj – to nic złego

Użyj generatora kodu, aby wygenerować lokalizator, a następnie edytuj go według potrzeb.

Przykładowy kod:

```
await page.getByPlaceholder('User Name')
    .fill('John');

await page.getLabel('Password')
    .fill('secret-password');

await page.getTestId('a-special-button').click();

const buttonLocator = page.getByRole('button', {
    name: 'Sign in'
})

buttonLocator.hover();
buttonLocator.click();

await expect(page.frameLocator('#my-frame')
    .getByText('Welcome, John!'))
    .toBeVisible();
```

20 minut

zadanie nr 5

Test E2E komponentu TodosList

- ▶ uruchom aplikację lokalnie
- ▶ napisz test, który:
 - ▶ otwiera stronę
 - ▶ sprawdza, czy lista zadań jest wyświetlana
 - ▶ sprawdza, ile kart zadań się renderuje



Akcje

Akcja	Opis	Dodatkowe opcje
<code>locator.click()</code>	Kliknięcie w element (domyślnie w jego środek). Obsługuje też automatyczne przewijanie, czekanie na widoczność i gotowość do interakcji. W przypadku pól formularza automatycznie skupi się na nim.	<code>button</code> , <code>clickCount</code> , <code>delay</code> , <code>modifiers</code>
<code>locator dblclick()</code>	Podwójne kliknięcie.	
<code>locator.hover()</code>	Najeżdża myszką na element (np. do wywołania podpowiedzi, rozwinięcia menu itp.).	
<code>locator.fill(value)</code>	Czyści pole i wpisuje podaną wartość. Działa tylko na polach typu <code><input></code> , <code><textarea></code> i <code>[contenteditable]</code> .	
<code>locator.type(value)</code>	Wpisuje znaki jeden po drugim – symuluje realne pisanie (można ustawić opóźnienie).	

Akcja	Opis	Dodatkowe opcje
<code>locator.press('Enter')</code>	Wysyła naciśnięcie klawisza. Obsługuje również modyfikatory (Shift+Tab, Control+A itd.).	
<code>locator.inputValue()</code>	Zwraca bieżącą wartość pola.	
<code>locator.evaluate(fn)</code>	Pozwala wykonać dowolną funkcję na elemencie DOM – np. odczytać atrybuty, styl, stan klasy itp.	
<code>locator.setInputFiles(filePath)</code>	Ustawia pliki w <code><input type="file" /></code> .	
<code>locator.selectOption(value)</code>	Wybiera opcję z <code><select></code> . Można wskazać value, label lub index.	
<code>locator.check() / locator.uncheck()</code>	Zaznacza lub odznacza checkbox (jeśli nie jest już w odpowiednim stanie).	

Akcja	Opis	Dodatkowe opcje
<code>locator.focus()</code> / <code>locator.blur()</code>	Ręczne ustawienie/opuszczenie fokusu – przydatne np. do testowania walidacji onBlur.	
<code>locator.scrollIntoViewIfNeeded()</code>	Wymusza przewinięcie do elementu, jeśli jest poza ekranem.	
<code>locator.selectText()</code>	Zaznacza cały tekst w wybranym polu formularz	

locator.click()

Przykładowy kod:

```
await locator.click({ button: 'right', clickCount: 2, delay: 100, modifiers: ['Control'] });
```

Właściwość	Opcje	Opis
button	left / right / middle	Określa, którego przycisku myszy użyć przy kliknięciu
clickCount		Liczba kliknięć (np. podwójne kliknięcie).
delay		Czas (w milisekundach) między naciśnięciem a puszczeniem przycisku myszy. Symuluje wolniejsze kliknięcie, np. jak człowiek.
modifiers		Lista klawiszy modyfikujących, trzymanych podczas kliknięcia (np. Shift, Control, Alt, Meta).

locator.evaluate(fn)

Funkcja `evaluate()` pozwala wykonać dowolny kod JavaScript wewnątrz przeglądarki, bezpośrednio na wybranym elemencie DOM zlokalizowanym przez `locator`.

To jakby "wejść do przeglądarki" i operować na elemencie dokładnie tak, jak robiłby to użytkownik lub developer w konsoli DevTools.

Przykładowy kod:

```
const bg = await locator.evaluate(el => getComputedStyle(el).backgroundColor);
```

Dobre praktyki

- ◀ nie używaj `page.click()` jeśli możesz użyć `locator.click()` – lokalizatory mają `retry logic` i są bardziej odporne na testy typu flaky
- ◀ każda akcja automatycznie czeka, aż element stanie się widoczny, gotowy do interakcji (np. nie będzie przykryty przez loader)
- ◀ nie musisz dodawać `waitFor` - większość akcji ma `retry` i wewnętrzny `timeout` (domyślnie 30s).

Asercje

- ▶ asercje to sprawdzenia warunków, które muszą być spełnione w trakcie testu
- ▶ jeżeli warunek nie zostanie spełniony, test zakończy się błędem
- ▶ w Playwright są one inteligentne – zamiast natychmiastowej weryfikacji, czekają domyślnie **do 5 sekund** (lub innej wartości timeoutu), próbując ponowić sprawdzenie
- ▶ mechanizm ponawiania pozwala na większą odporność testów na opóźnienia, animacje, renderowanie dynamiczne

Mechanizm retry / ponawiania

Kiedy używasz `expect()`, Playwright nie wykonuje pojedynczego porównania. Zamiast tego:

- ◀ sprawdza dany element lub stronę
- ◀ sprawdza, czy warunek jest spełniony
- ◀ **jeśli nie** - czeka chwilę i ponawia próbę
- ◀ robi to do czasu przekroczenia timeoutu

Dzięki temu możemy pisać testy bez `wait()` i `sleep()`.

Rodzaje asercji

Powiązane z page:

Przykładowy kod:

```
await expect(page).toHaveURL(/.*\/login/);
await expect(page).toHaveTitle(/.*checkout/);
await expect(page).toHaveScreenshot('image.png');
```

Powiązane z lokalizatorami"

Przykładowy kod:

```
await page.locator('#submit-button').click();
await expect(page.locator('.status')).toHaveText('Submitted');
await expect(page.getByText('Hidden text')).toBeAttached();
```

Rodzaje asercji

Związane z widocznością elementów:

Przykładowy kod:

```
// A specific element is visible.  
await expect(page.getText('Welcome')).toBeVisible();  
// At least one item in the list is visible.  
await expect(page.getTestId('todo-item').first()).toBeVisible();
```

Asercje ogólne i negowane:

Przykładowy kod:

```
expect('abc').toEqual(expect.any(String));  
expect([1, 2, 3]).not.toEqual(expect.arrayContaining([1, 4]));  
expect({ foo: 1, bar: 2 }).toEqual(expect.objectContaining({ foo: 1 }));  
  
await expect(page.getText('Welcome')).not.toBeVisible();
```

Nie każda interakcja musi być natychmiast sprawdzana — ale dodanie asercji po kluczowych akcjach (np. kliknięciu przycisku, wysłaniu formularza) sprawia, że testy stają się wiarygodne i łatwe do diagnozy.

20 minut

zadanie nr 6

Test interakcji – dodawanie zadania

- ▶ użyj `page.locator('text=Dodaj zadanie')` do kliknięcia przycisku
- ▶ zweryfikuj czy pojawiło się nowe zadanie
- ▶ przetestuj dynamiczne `data-testid` i `data-done` atrybuty
- ▶ przetestuj walidację pola daty (nie może być w przeszłości)
- ▶ przetestuj walidację priorytetu (musi być wybrany)
- ▶ sprawdź czy błędy walidacji znikają po poprawieniu pól



Rozszerzone selektory CSS

Rozszerzone selektory w Playwright nie są częścią oficjalnej dokumentacji CSS. Służą do wygodniejszego lokalizowania elementów na stronie.

Selektor	Opis
<code>:has()</code>	zagnieżdżone dopasowanie
<code>:has-text()</code>	element zawierający dany tekst
<code>:text()</code>	skrót do <code>:has-text()</code>
<code>:visible</code>	tylko elementy widoczne
<code>:right-of()</code> , <code>:left-of()</code> , <code>:above()</code> , <code>:below()</code> , <code>:near()</code>	selektory przestrzenne
<code>:nth-match()</code>	konkretny dopasowany element z listy

Dlaczego to działa?

Playwright parsuje selektory samodzielnie – dzięki temu może dodawać własne rozszerzenia.

Przykładowy kod:

```
page.locator('button:has-text("Zaloguj")')
```

Przykładowy kod:

```
await page.locator('article:has-text("Playwright")').click();
await page.locator('#nav-bar :text("Home")').click();
await page.locator('button:visible').click();
await page.locator('article:has(div.promo)').textContent();

// Clicks a <button> that has either a "Log in" or "Sign in" text.
await page.locator('button:has-text("Log in"), button:has-text("Sign in")').click();

// Fill an input to the right of "Username".
await page.locator('input:right-of(:text("Username"))').fill('value');

// Click a button near the promo card.
await page.locator('button:near(.promo-card)').click();

// Click the radio input in the list closest to the "Label 3".
await page.locator('[type=radio]:left-of(:text("Label 3"))').first().click();

// Wait until all three buttons are visible
await page.locator(':nth-match(:text("Buy"), 3)').waitFor();

// Click last button
await page.locator('button').locator('nth=-1').click();
```

20 minut

zadanie nr 7

Zaawansowane selektory CSS

- ▶ napisz test, który będzie w menu nawigacyjnym pierwszy element na lewo od **O aplikacji** - czyli "**Strona główna**"
- ▶ sprawdź czy footer zawiera poprawny tekst **/Todo App/**
- ▶ wykorzystaj do tego celu złożone selektory CSS



Lokalizowanie zagnieżdżone i filtrowanie

Używamy chainowania i filtrowania lokalizatorów, aby precyzyjnie odwołać się do konkretnego elementu w złożonej strukturze DOM – np. produktu na liście, który spełnia określone warunki (tekst, zawartość, dostępność).

Zastosowania:

- ▶ interakcja z konkretnym kafelkiem w gridzie (np. kliknij przycisk tylko w produkcie „Product 2”)
- ▶ walidacja listy elementów z oczekiwany stanem
- ▶ budowanie stabilnych testów niezależnych od struktury DOM

Przykładowy kod:

```
// Znajdź przycisk, który musi spełniać dwa warunki:  
// - być przyciskiem z rolą i nazwą  
// - mieć określony test-id  
const addToCartBtn = page.getByRole('button', { name: 'Add to cart' })  
    .and(page.getByTestId('purchase-btn'));  
  
// Zlokalizuj wszystkie elementy listy (np. produkty)  
const products = page.getByRole('listitem');  
  
// Sprawdź, czy w nagłówkach produktów znajdują się 3 konkretne nazwy  
expect(products.locate(page.getByRole('heading')))  
    .toHaveText(['apple', 'banana', 'orange']);  
  
// Wyfiltruj produkt, który zawiera tekst "Product 2"  
const product = products  
    .filter({ hasText: /Product 2/ });  
// Można też filtrować po obecności innego lokalizatora:  
// .filter({ has: page.getByRole('heading', { name: 'Product 2' }) })  
  
// Kliknij przycisk "Add to cart" w kontekście wyfiltrowanego produktu  
product.locate(addToCartBtn).click();  
  
// Sprawdź, czy widoczne jest dokładnie 5 produktów dostępnych w magazynie  
await expect(page.getByRole('listitem'))  
    .filter({ hasNotText: 'Out of stock' })  
    .toHaveLength(5);
```

Page Object Pattern

Page Object Pattern (POP, czasami określany też jako **Page Object Model**) to wzorzec projektowy stosowany w testach automatycznych (szczególnie E2E), który polega na oddzieleniu logiki testowej od szczegółów interfejsu użytkownika.

Celem jest uczynienie testów czytelniejszymi, bardziej odpornymi na zmiany UI i łatwiejszymi w utrzymaniu.

Zamiast pisać w testach bezpośrednio:

Przykładowy kod:

```
await page.locator('input[name="username"]').fill('admin');
await page.locator('input[name="password"]').fill('secret');
await page.locator('button[type="submit"]').click();
```

...tworzymy klasę (tzw. **page object**), która reprezentuje stronę i udostępnia metody takie jak:

Przykładowy kod:

```
await LoginPage.login('admin', 'secret');
```

Przykładowy kod:

```
export class LoginPage {
  constructor(private page: Page) {}

  readonly usernameInput = this.page.locator('input[name="username"]');
  readonly passwordInput = this.page.locator('input[name="password"]');
  readonly submitButton = this.page.locator('button[type="submit"]');

  async goto() {
    await this.page.goto('/login');
  }

  async login(username: string, password: string) {
    await this.usernameInput.fill(username);
    await this.passwordInput.fill(password);
    await this.submitButton.click();
  }
}
```

Przykładowy kod:

```
const LoginPage = new LoginPage(page);
await LoginPage.goto();
await LoginPage.login('admin', 'secret');
```

zalety Page Object Pattern

Zaleta	Opis
Reużywalność	Logika interakcji jest w jednym miejscu, używana przez wiele testów
Czystość kodu testu	Test skupia się na scenariuszu, nie na technicznych szczegółach
Odporność na zmiany UI	Jeśli zmieni się selektor – poprawiasz go tylko raz, w Page Object
Lepsza czytelność	Testy stają się prostsze w interpretacji nawet dla nietechnicznych

Dobre praktyki

- ◆ jeden Page Object to jedna strona lub jeden komponent
- ◆ nazwy metod w Page Object powinny być semantyczne (np. `addToCart()`, `submitForm()`)
- ◆ nie testuj wewnętrz Page Object – testuj tylko z zewnątrz (**testy nie powinny wiedzieć o selektorach**)
- ◆ unikaj **Page Object God Class** – nie ładuj wszystkiego do jednej klasy

20 minut

zadanie nr 8

Page Object Pattern

- stwórz klasę `TodosPage` w `tests/pages/TodosPage.ts`
- umieść metody `addTodo`, `removeTodo`, `getTodoCount`
- przepisz testy na liście `todos` z wykorzystaniem Page Object Pattern



Downloads

Przykładowy kod:

```
// Zaczni j nasłuchiwać pobierania przed kliknięciem
// Uwaga nie używamy AWAIT!
const downloadPromise = page.waitForEvent('download');

// Kliknięcie wyzwalające pobieranie
await page.getText('Download file').click();

// Odbierz obiekt `Download`, gdy się pojawi
const download = await downloadPromise;

// Zapisz plik lokalnie z oryginalną nazwą
await download.saveAs('/path/to/save/at/' + download.suggestedFilename());
```

Uwaga!

- ◆ pobieranie działa tylko dla prawdziwych pobrań plików – nie dla `window.open()` lub przekierowań do PDF
- ◆ folder docelowy musi istnieć – Playwright nie tworzy katalogów automatycznie przy `saveAs`

Testy screenshotowe

Porównują zrzut ekranu aktualnego elementu lub strony z wcześniej zapisanym wzorcem. Jeśli się różnią – test nie przechodzi.

Przykładowy kod:

```
const button = page.getByRole('button', { name: 'Kup teraz' });
await expect(button).toHaveScreenshot('buy-button.png');
```

Przykładowy kod:

```
await expect(page).toHaveScreenshot();
```

Jak działa tolerancja w `toHaveScreenshot()`?

- ▶ domyślnie Playwright toleruje bardzo drobne różnice wynikające np. z antialiasingu, czcionek systemowych itp.
- ▶ jeśli różnica jestauważalna wizualnie – test najczęściej nie przejdzie
- ▶ można ustawić bardziej precyzyjne limity przez opcję `maxDiffPixels` lub `maxDiffPixelRatio`

Przykładowy kod:

```
await expect(page).toHaveScreenshot('homepage.png', {  
  maxDiffPixels: 100,           // maks. 100 różnych pikseli  
  maxDiffPixelRatio: 0.01,      // maks. 1% pikseli może się różnić  
  threshold: 0.2,             // (jeśli robisz screenshot "manualnie", nie `toHaveScreenshot`)  
});
```

Testy snapshotowe (tekstowe / DOMowe)

Porównują tekst lub strukturę DOM (np..textContent, JSON, HTML) z zapisanym wcześniej wzorcem.

Przykładowy kod:

```
expect(await page.textContent('.hero__title'))  
  .toMatchSnapshot('hero-title.txt');
```

Przykładowy kod:

```
expect(apiResponse).toMatchSnapshot('api-response.json');
```

Jak zaktualizować snapshoty?

Przykładowy kod:

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

To sprawi, że:

- ▶ jeśli snapshot nie istnieje - zostanie stworzony
- ▶ jeśli snapshot istnieje, ale się różni - zostanie zaktualizowany

Gdzie są zapisywane snapshoty?

- ▶ domyślnie w katalogu `_snapshots_` obok testu (dla snapshotów tekstowych)
- ▶ screenshoty (np. `image.png`) zapisywane są w `test-results` i jako wzorzec (`*.png`) w katalogu testowym

Uwagi

- ▶ testy screenshotowe mogą być wrażliwe na zmiany środowiska (czcionki systemowe, rozdzielcość, animacje)
- ▶ snapshoty tekstowe mogą się „zacierać” – dobrze je czytelnie nazywać (login-success.txt, cart-total.json, itp.)
- ▶ warto je stosować jako uzupełnienie, nie zamiast testów funkcjonalnych

15 minut

zadanie nr 9

Asercje wizualne

- ◀ przygotuj test wizualny dla całej strony:
`expect(page).toHaveScreenshot()`
- ◀ dodaj test:
 - ◀ przed i po dodaniu zadania
 - ◀ sprawdź różnice w `_screenshots_`



174

API Mocking

Przykładowy kod:

```
test("mocks a fruit and doesn't call api", async ({ page }) => {
  // Przechwytyjemy wszystkie żądania do API owoców
  await page.route('*/**/api/v1/fruits', async route => {
    // Zwracamy własną, mockowaną odpowiedź JSON
    const json = [{ name: 'Strawberry', id: 21 }];
    await route.fulfill({ json });
  });

  // Otwieramy stronę, która normalnie zrobiłaby prawdziwe zapytanie
  await page.goto('https://demo.playwright.dev/api-mocking');

  // Sprawdzamy, że "Strawberry" pojawił się na stronie (czyli mock działa)
  await expect(page.getText('Strawberry')).toBeVisible();
});
```

Kiedy stosować?

- ▶ izolowanie frontendu od backendu
- ▶ testowanie edge case'ów (np. pustych list, błędów serwera)
- ▶ szybsze i bardziej przewidywalne testy
- ▶ brak zależności od środowisk stagingowych

zaawansowane możliwości

Zablokowanie żądania

Przykładowy kod:

```
await page.route('**/ads', route => route.abort());
```

Symulacja błędu serwera

Przykładowy kod:

```
await page.route('**/api/v1/user', route => {
  route.fulfill({
    status: 500,
    body: 'Server error'
  });
});
```

Dobre praktyki

- ◀ mockuj przed `page.goto()` – Playwright musi wiedzieć, że ma podmienić odpowiedź zanim strona zrobi żądanie.
- ◀ trzymaj mocki w osobnych plikach (np. `mocks/fruits.mock.ts`)
- ◀ w CI uruchamiaj testy mockowane, a w stagingu – z prawdziwym API



zapisywanie sesji

- ▶ zalecanym miejscem przechowywania plików sesji (`storageState`) w Playwright jest katalog `playwright/.auth/`
- ▶ Playwright nie ma domyślnie zdefiniowanego miejsca zapisu, dlatego należy samodzielnie o niego zadbać
- ▶ jeśli nie podamy ścieżki podczas zapisu stanu - Playwright zwróci obiekt, ale go nie zapisze nigdzie
- ▶ dobrze jest oddzielić katalog sesji od testów
- ▶ w ramach projektu możemy mieć wiele plików sesji (`admin.json` , `user.json` , `guest.json`)

Jak działa storageState()?

Przykładowy kod:

```
await page.context().storageState({ path: 'user.json' });
```

- ◆ zapisuje wszystkie ciasteczka, `localStorage`, tokeny itp.
- ◆ działa dla całego kontekstu przeglądarki (`browser.newContext()`)

Przykładowy kod:

```
browser.newContext({ storageState: 'user.json' });
```

- ◆ wczytuje storage do nowego kontekstu przeglądarki

UWAGA: storageState nie zapisuje `sessionStorage`

Przykładowy kod:

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

(async ({ page }) => {
  await page.goto('https://github.com/login');
  await page.getLabel('Username or email address').fill('username');
  await page.getLabel('Password').fill('password');
  await page.getRole('button', { name: 'Sign in' }).click();

  // Poczekaj na pełne zalogowanie – czasami proces wymaga wielokrotnego przekierowywania użytkownika
  await page.waitForURL('https://github.com/');
  await expect(page.getRole('button', { name: 'View profile and more' })).toBeVisible();

  // Zapisz stan sesji
  await page.context().storageState({ path: authFile });
})();
```

Przykładowy kod:

```
node authentication.setup.js
```

Używanie sesji w innych testach

Przykładowy kod:

```
test.use({
  storageState: 'playwright/.auth/user.json'
});

test('dashboard is visible', async ({ page }) => {
  await page.goto('https://github.com/');
  await expect(page.getText('Your repositories')).toBeVisible();
});
```

A co gdy muszę użyć sessionStorage?

Przykładowy kod:

```
// Zapisujemy sessionStorage z przeglądarki do pliku jako JSON
const sessionStorage = await page.evaluate(() => JSON.stringify(sessionStorage));
// Zapisujemy ten JSON do pliku na dysku
fs.writeFileSync('playwright/.auth/session.json', sessionStorage, 'utf-8');

// W nowym kontekście dodajemy initScript, który ustawia sessionStorage z powrotem, zanim załaduje się strona
const sessionStorage = JSON.parse(fs.readFileSync('playwright/.auth/session.json', 'utf-8'));
await context.addInitScript(storage => {
  if (window.location.hostname === 'example.com') {
    for (const [key, value] of Object.entries(storage))
      window.sessionStorage.setItem(key, value);
  }
}, sessionStorage);
```

Ograniczenia

- ◀ `storageState()` nie zapisuje `sessionStorage`, tylko `cookies` i `localStorage`
- ◀ `sessionStorage` jest izolowane dla zakładki – nie przetrwa zamknięcia strony
- ◀ aby przywrócić `sessionStorage`, musisz użyć `addInitScript` i uruchomić go przed `goto()`

Zastosowanie

- ◀ gdy aplikacja przechowuje dane tylko w `sessionStorage` (a nie `localStorage` czy `cookies`)
- ◀ gdy chcesz przechować stan logowania bez korzystania z `storageState()`
- ◀ gdy aplikacja odczytuje stan z `sessionStorage` przed jakimkolwiek innym działaniem

Co to jest plik `.setup.ts`?

To dowolny plik TypeScript (lub JavaScript), w którym możesz:

- ▶ ustawić dane w `localStorage` / `cookies`
- ▶ wykonać logowanie i zapisać `storageState`
- ▶ stworzyć pliki testowe
- ▶ dodać dane testowe przez API
- ▶ wyczyścić lub przygotować bazę danych (jeśli masz taką integrację)

Playwright nie traktuje `.setup.js` w żaden szczególny sposób – to tylko konwencja. Możesz go nazwać `auth.prepare.ts` , `session.setup.ts` , `init.env.ts` itd.

Przykładowy kod:

```
node tests/auth.setup.js
```

20 minut

zadanie nr 10

Testowanie autoryzacji / tokenów

- ▶ przygotuj plik setup, który będzie zapisywał dane logowania w pliku powiązanym docelowo z `storageState`
- ▶ uruchom setup (np. poprzez `node tests/auth.setup.js`)
- ▶ wykorzystaj dane do logowania w innym teście



Testowanie API z Playwright

Przykładowy kod:

```
test('should create a bug report', async ({ request }) => {
  const newIssue = await request.post(`repos/${USER}/${REPO}/issues`, {
    data: {
      title: '[Bug] report 1',
      body: 'Bug description',
    }
  });
  expect(newIssue.ok()).toBeTruthy();

  const issues = await request.get(`repos/${USER}/${REPO}/issues`);
  expect(issues.ok()).toBeTruthy();
  expect(await issues.json()).toContainEqual(expect.objectContaining({
    title: '[Bug] report 1',
    body: 'Bug description'
  }));
});
```

Cel testu

- ▶ tworzenie zgłoszenia błędu (issue) przez API
- ▶ sprawdzenie, czy nowe zgłoszenie jest widoczne na liście
- ▶ upewnienie się, że odpowiedzi HTTP są poprawne

Kluczowe techniki testowania

- ◀ `request.post` i `request.get` – testowanie REST API
- ◀ `expect(response.ok()).toBeTruthy()` – kontrola poprawności HTTP
- ◀ `expect.objectContaining(...)` – częściowe dopasowanie JSON-a
- ◀ test end-to-end: tworzenie + odczyt + walidacja

Request context

`request.newContext()` pozwala utworzyć dedykowany kontekst dla żądań HTTP, dzięki czemu nie trzeba powtarzać nagłówków, adresu URL czy tokenów autoryzacyjnych w każdym teście.

Przykładowy kod:

```
test.beforeAll(async ({ playwright }) => {
  apiContext = await playwright.request.newContext({
    // All requests we send go to this API endpoint.
    baseURL: 'https://api.github.com',
    extraHTTPHeaders: {
      // We set this header per GitHub guidelines.
      'Accept': 'application/vnd.github.v3+json',
      // Add authorization token to all requests.
      // Assuming personal access token available in the environment.
      'Authorization': `token ${process.env.API_TOKEN}`,
    },
  });
});
```

Przykładowy kod:

```
test('create issue via API', async () => {
  const response = await apiContext.post('/repos/user/repo/issues', {
    data: {
      title: 'New issue',
      body: 'Bug found here',
    }
  });
  expect(response.ok()).toBeTruthy();
});
```

Co to jest request context?

- ◀ dedykowany „klient API” w Playwright
- ◀ tworzony raz i współdzielony w testach
- ◀ może zawierać:
 - ◀ `baseURL`
 - ◀ nagłówki (`Authorization`, `Accept`, itp.)
 - ◀ `Proxy`, `cookies`, dane sesyjne

Konfiguracja API w playwright.config.ts

Przykładowy kod:

```
import { defineConfig } from '@playwright/test';
export default defineConfig({
  use: {
    // All requests we send go to this API endpoint.
    baseURL: 'https://api.github.com',
    extraHTTPHeaders: {
      // We set this header per GitHub guidelines.
      'Accept': 'application/vnd.github.v3+json',
      // Add authorization token to all requests.
      // Assuming personal access token available in the environment.
      'Authorization': `token ${process.env.API_TOKEN}`,
    },
  },
});
```

Co nam to daje?

Element	Korzyść
baseURL	Brak potrzeby powtarzania pełnych adresów
extraHTTPHeaders	Centralne miejsce na tokeny i nagłówki
process.env.API_TOKEN	Bezpieczne przechowywanie sekretów
defineConfig	Czytelna i skalowalna konfiguracja Playwrighta

Użycie w teście

Przykładowy kod:

```
test('create issue', async ({ request }) => {
  const res = await request.post('/repos/user/repo/issues', {
    data: {
      title: 'New bug',
      body: 'Steps to reproduce...'
    }
  });
  expect(res.ok()).toBeTruthy();
});
```

20 minut

zadanie nr 11

Mockowanie API

- ▶ zamockuj `/data.json` za pomocą `page.route()` i `page.request`
- ▶ przetestuj renderowanie z pustą listą
- ▶ przetestuj renderowanie z błędem HTTP 500



Obsługa iframe

Przykładowy kod:

```
const frame = page.frame({ name: 'my-frame' }); // lub frameLocator  
await frame.click('text=Submit');
```

Można używać **frameLocator('iframe[name="my-frame"]')** dla lepszej stabilności

Test lifecycle hooks

Hook	Kiedy się uruchamia	Przeznaczenie
<code>beforeAll</code>	Raz przed wszystkimi testami w danym describe	Inicjalizacja zasobów wspólnych
<code>afterAll</code>	Raz po wszystkich testach	Zwolnienie zasobów, np. zamknięcie bazy
<code>beforeEach</code>	Przed każdym testem	Przygotowanie danych, stanu
<code>afterEach</code>	Po każdym teście	Czyszczenie danych, reset stanu

Struktura testów w Playwright

Przykładowy kod:

```
tests/  
  └── api/  
    └── create-issue.spec.ts  
  └── ui/  
    └── login.spec.ts  
  └── fixtures/  
    └── auth.ts  
playwright.config.ts  
.env
```

Struktura pliku testowego

Przykładowy kod:

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

test.describe('GitHub issues API', () => {
  test.beforeEach(async ({ request }) => {
    // setup
  });

  test('should create a bug report', async ({ request }) => {
    const res = await request.post('/repos/user/repo/issues', {
      data: { title: 'Bug', body: 'Details' }
    });
    expect(res.ok()).toBeTruthy();
  });

  test.afterEach(async () => {
    // cleanup
  });
});
```

Dobre praktyki

Element	Rekomendacja
Foldery	Podziel testy na api, ui, e2e
<code>describe</code>	Grupuj tematycznie (np. per endpoint, widok)
<code>beforeEach</code> / <code>afterEach</code>	Przygotowanie i sprzątanie
Reużywalność	Wydziel <code>fixtures</code> i <code>utils</code>
Jasne asercje	<code>expect(...).toBeTruthy()</code> / <code>toHaveText()</code> , itd.

Generowanie raportów w Playwright

Domyślny raport

Po uruchomieniu testów Playwright generuje podstawowe raporty tekstowe w konsoli.

HTML Reporter

Przykładowy kod:

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

Interaktywny raport

Przykładowy kod:

```
npx playwright show-report
```

Inne typy reporterów

W `playwright.config.ts` możesz zdefiniować różne typy raportów:

Przykładowy kod:

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

export default defineConfig({
  reporter: [
    ['list'], // konsolowy
    ['html', { outputFolder: 'playwright-report', open: 'never' }],
    ['json', { outputFile: 'report.json' }],
    ['junit', { outputFile: 'report.xml' }],
  ]
});
```

Typowe zastosowania raportów

Typ reportera	Przeznaczenie
html	Analiza lokalna, czytelny UI
json	Integracja z narzędziami CI/CD
junit	Kompatybilność z Jenkins, GitLab CI



Dobre praktyki

- ▶ dodaj reporter do configu — ustandaryzujesz środowisko testowe
- ▶ zautomatyzuj generowanie i publikację raportów w CI/CD
- ▶ dodaj skrypt `posttest` do `package.json`, np.:

Przykładowy kod:

```
"scripts": {  
  "test": "playwright test",  
  "posttest": "playwright show-report"  
}
```

Generowanie zrzutów ekranu i filmów z testów

Robienie zrzutu ekranu ręcznie

Przykładowy kod:

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

Automatyczne zrzuty przy błędach

Przykładowy kod:

```
// playwright.config.ts
use: {
  screenshot: 'only-on-failure', // 'on', 'off', 'only-on-failure'
}
```

Włączenie nagrywania testów

Przykładowy kod:

```
// playwright.config.ts
use: {
  video: 'retain-on-failure', // 'on', 'off', 'retain-on-failure'
}
```

Lokalizacja filmów: test-results//video.webm

Przykład pełnej konfiguracji

Przykładowy kod:

```
export default defineConfig({
  use: {
    screenshot: 'only-on-failure',
    video: 'retain-on-failure',
    trace: 'retain-on-failure', // opcjonalnie: trace z krokami
  }
})
```

15 minut

zadanie nr 12

Generowanie raportów

- ▶ uruchom `npx playwright show-report`
- ▶ dokończ konfigurację generowania raportów, tak aby generował raport do konsoli, w formacie `html` i `json`
- ▶ wygeneruj raporty po testach w różnych formatach



209

Symulacja interakcji na klawiaturze

Playwright pozwala dokładnie odwzorować zachowanie użytkownika, w tym wciskanie klawiszy, kombinacje oraz wpisywanie tekstu.

Przykładowy kod:

```
await page.keyboard.press('Enter');
await page.keyboard.press('ArrowDown');
await page.keyboard.press('Control+A');
```

sharding

To technika dzielenia zestawu testów na mniejsze fragmenty (**shardy**) i uruchamiania ich równolegle - najczęściej na wielu maszynach lub wątkach.

Przykład

Mamy 100 testów e2e. Zamiast uruchamiać je jeden po drugim na jednej maszynie możemy zastosować sharding, czyli podzielić na mniejsze grupy, np. 4 po 25 testów.

- ▶ shard 1 - testy od 1 do 25
- ▶ shard 2 - testy od 26 do 50
- ▶ shard 3 - testy od 51 do 75
- ▶ shard 4 - testy od 76 do 100

Wszystkie shardy mogą być uruchomione jednocześnie w ramach procesu CI / CD. Czas wykonania takich testów może spaść kilkukrotnie.

sharding w Playwright

- ▶ Playwright wspiera sharding domyślnie, bez dodatkowych konfiguracji
- ▶ wystarczy dodać odpowiednią flagę podczas uruchamiania testów

Przykładowy kod:

```
npx playwright test --shard=1/3  
npx playwright test --shard=2/3  
npx playwright test --shard=3/3
```

Powyższy zapis oznacza, że Playwright podzieli wszystkie testy automatycznie na mniej więcej równe grupy (w tym przypadku 3) i uruchomi odpowiedni każdą z nich oddziennie. Zapis `1/3` lub `2/6` to: `<n>/<m>`, gdzie `n` to numer fragmentu, a `m` to ilość ich wszystkich.

Jak Playwright dzieli testy?

- ▶ Playwright automatycznie sortuje i dzieli testy tak, by każdy fragment miał mniej więcej tyle samo pracy
- ▶ Podział jest na poziomie plików testowych (`*.spec.ts`), nie pojedynczych testów (`it(...)`)

25 minut

zadanie nr 13

Testowanie filtrowania i wyszukiwania

Napisz test E2E, który sprawdza filtrowanie zadań po statusie:

- ▶ Kliknij filter "Aktywne" i sprawdź czy wyświetlane są tylko nieukończone zadania
- ▶ Kliknij filter "Ukończone" i sprawdź czy wyświetlane są tylko ukończone zadania
- ▶ Kliknij filter "Przeterminowane" i sprawdź logikę dat

Przetestuj filtrowanie po priorytecie:

- ▶ Użyj selektora dropdown'a priorytetu
- ▶ Sprawdź czy liczba wyświetlanych zadań się zmienia

Przetestuj wyszukiwanie:

- ▶ Wprowadź tekst w pole wyszukiwania
- ▶ Everylin i Rafał z wynikiem **zawiera** wprowadzony tekst w tytule lub opisie

mateusz i alena



214

test.step

- ◆ służy do grupowania i logowania kroków w teście
- ◆ poprawia czytelność raportów i trace
- ◆ lepsza analiza w reportach i trace viewer

Przykładowy kod:

```
await test.step("Log in as user", async () => {
  await page.goto("/login");
  await page.fill("#username", "admin");
  await page.fill("#password", "1234");
  await page.click("button[type=submit]");
});
```

test.describe.parallel

- ▶ umożliwia uruchamianie grupy testów równolegle
- ▶ przyspiesza suite testowy
- ▶ uważaj na dane współdzielone - muszą być izolowane

Przykładowy kod:

```
test.describe.parallel("User roles", () => {
  test("Admin can see dashboard", async ({ page }) => { ... });
  test("User cannot access admin panel", async ({ page }) => { ... });
});
```

test.skip

- ▶ pomija test (np. bug w systemie, niegotową funkcję)
- ▶ raport pokaże test jako pominięty

Przykładowy kod:

```
test.skip("Feature X not ready yet", async ({ page }) => {  
  ...  
});
```

test.only

- ◀ uruchamia tylko wskazany test (ignoruje resztę)
- ◀ nie zapomnij usunąć przed commitem

Przykładowy kod:

```
test.only("Debug this scenario", async ({ page }) => {  
  ...  
});
```

20 minut

zadanie nr 14

Test operacji masowych

- ▶ Przetestuj funkcjonalność "✓ Zaznacz wszystkie":
 - ▶ Sprawdź czy przycisk jest aktywny tylko gdy są aktywne zadania
 - ▶ Kliknij przycisk i zweryfikuj czy wszystkie aktywne zadania zostały oznaczone jako ukończone
- ▶ Przetestuj "✗ Odznacz wszystkie":
 - ▶ Sprawdź czy przycisk jest aktywny tylko gdy są ukończone zadania
 - ▶ Kliknij i sprawdź czy zadania wracają do stanu aktywnego
- ▶ Przetestuj "☒ Usuń ukończone":
 - ▶ Oznacz kilka zadań jako ukończone
 - ▶ Kliknij przycisk usuń ukończone
 - ▶ Potwierdź w dialogu i sprawdź czy zadania zniknęły z listy



25 minut

zadanie nr 15

Testowanie edycji zadań

◆ Napisz test pełnego flow edycji:

- ◆ Znajdź zadanie i kliknij przycisk "✎ Edytuj"
- ◆ Sprawdź czy pojawiła się forma edycji z wypełnionymi polami
- ◆ Zmodyfikuj tytuł, opis, priorytet i datę wykonania
- ◆ Kliknij "💾 Zapisz zmiany" i sprawdź czy zmiany zostały zastosowane

◆ Przetestuj anulowanie edycji:

- ◆ Otwórz edycję zadania
- ◆ Zmodyfikuj dane
- ◆ Kliknij "✖ Anuluj"
- ◆ Sprawdź czy zadanie wróciło do oryginalnego stanu

◆ Przetestuj walidację w edycji:



30 minut

zadanie nr 16

Testowanie routingu i autoryzacji

► Przetestuj system logowania:

- Otwórz aplikację (powinna przekierować na login)
- Wprowadź błędne hasło i sprawdź komunikat błędu
- Wprowadź prawidłowe hasło (**admin123**) i sprawdź przekierowanie

► Przetestuj nawigację:

- Kliknij link "O aplikacji" w menu
- Sprawdź czy URL zmienił się na **#/about**
- Wróć do "Lista zadań" i sprawdź URL **#/todos**

► Przetestuj ochronę tras:

► Wczyść sessionStorage: **await page.evaluate(() =>**
~~mateuszjabłoński@sessionStorage.clear()~~

- Spróbuj wejść bezpośrednio na **#/todos**



221

Dobre praktyki

222

Antywzorce w testach

Obywatel drugiej kategorii

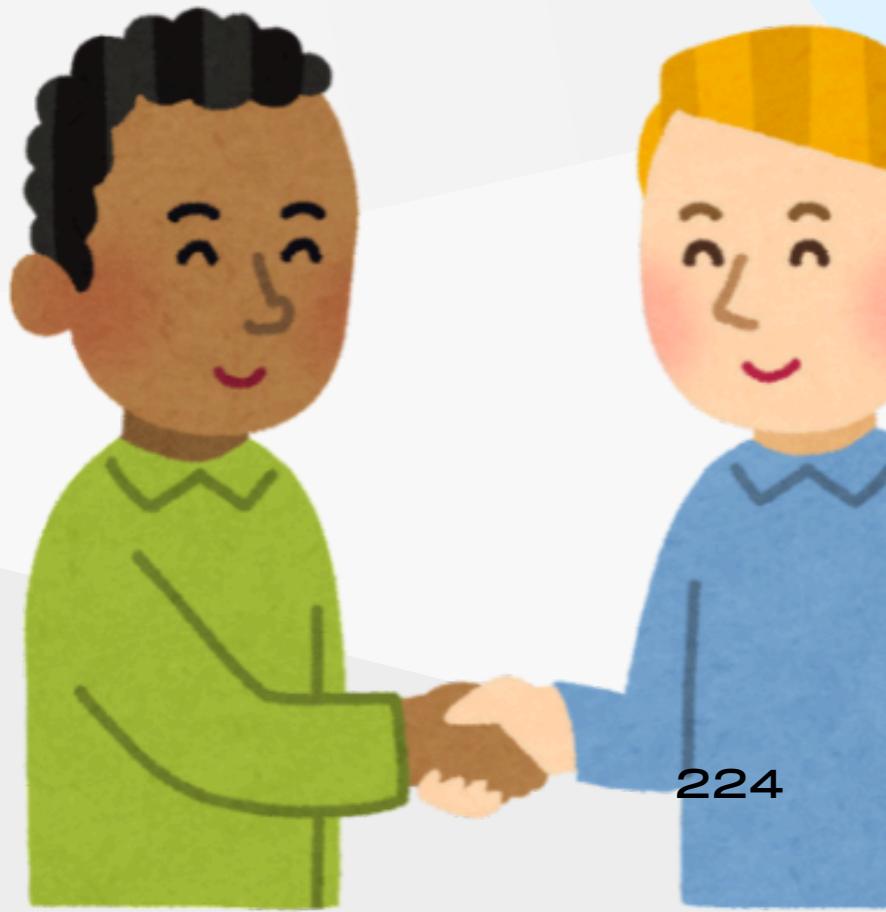
- ◀ kod testów traktowany jako ten gorszy
- ◀ nie jest utrzymywany, wszędzie mamy do czynienia z duplikacją kodu, chaosem i mylącymi nazwami



223

Brak szacunku

- ◀ deweloper robi jakieś zmianę i nie aktualizuje unit testów
- ◀ po wprowadzeniu system continuous integration zwraca błąd spowodowany nieprzechodzącymi testami
- ◀ zamiast poprawić testy usuwa się je, zakomentowuje albo ignoruje



Optymistyczna ścieżka

- ◀ testowanie tylko podstawowego działania funkcji bez zastanowienia się nad możliwymi wyjątkami, warunkami brzegowymi, czy złośliwymi kombinacjami danych wejściowych.

Gigant

- ▶ test zawierający bardzo dużo linii kodu – kilkaset, albo nawet ponad tysiąc
- ▶ test jest tak duży, że nie wiadomo do końca co robi, jego utrzymanie jest bardzo trudne i może on posiadać swoje własne bugi



Pasażer na gapę

- zamiast stworzyć nowy test, dodajemy kolejny cykl Arrange-Act-Assert do istniejącego testu
- w ten sposób istniejące testy się wydłużają, a ich cel się rozmywa
- w końcu taki test przeobraża się w giganta

Przykładowy kod:

```
test("Zbyt długi test – pasażer na gapę", () => {
  const user = createUser("Bob", 25);
  expect(user.name).toBe("Bob");

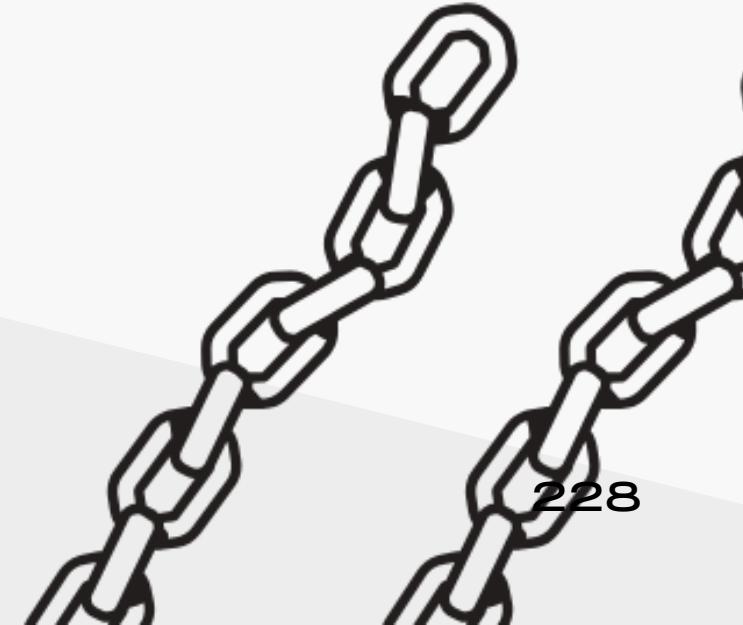
  user.updateAge(26);
  expect(user.age).toBe(26);

  user.deactivate();
  expect(user.isActive).toBe(false);
});
```



chain gang

- ◀ kolejne testy są od siebie zależne i muszą być wykonywane w odpowiedniej kolejności
- ◀ usunięcie lub zmiana jednego testu z łańcucha skutkuje wtedy zepsaniem całej grupy testów



Sobowtór

- ▶ na potrzeby testu mockujemy jakąś zależność, jednak aby wykonać test nie wystarczy nam podstawić zwracaną wartość - zamiast tego w mocku kopujemy zachowanie rzeczywistej funkcji
- ▶ jeżeli będziemy chcieli wprowadzić zmiany, musimy je uwzględnić zarówno w kodzie produkcyjnym, jak i w mocku



Kłamca

- ◀ test jest tak skonstruowany, żeby wchodził w odpowiednie ścieżki w kodzie i nabijał w ten sposób code coverage
- ◀ jednak nie sprawdza on w żaden sposób poprawności wykonywania tych ścieżek



Inspektor

- ◆ aby osiągnąć lepsze pokrycie, test bazuje na konkretnej implementacji i łamie zasady enkapsulacji
- ◆ w ten sposób otrzymujemy skomplikowane testy utrudniające refactor kodu w przyszłości



Flaky test

- ▶ test raz przechodzi, a raz nie – bez zmiany kodu
- ▶ może być zależny od kolejności uruchamiania, środowiska czy nawet czasu
- ▶ powoduje frustrację i zmniejsza zaufanie do testów

Przykładowy kod:

```
test("Flaky test - zależny od czasu", () => {
  const now = new Date().getSeconds();
  expect(now % 2).toBe(0); // Test przejdzie tylko w parzystych sekundach
});
```



Mr. Tester

- ▶ test, który testuje wszystko naraz
- ▶ zamiast jednego konkretnego przypadku sprawdza kilka rzeczy jednocześnie
- ▶ trudny do zrozumienia i poprawienia, bo nie wiadomo, co dokładnie się zepsuło

Przykładowy kod:

```
test("Sprawdza wszystko na raz", () => {
  const user = createUser("Alice", 30);

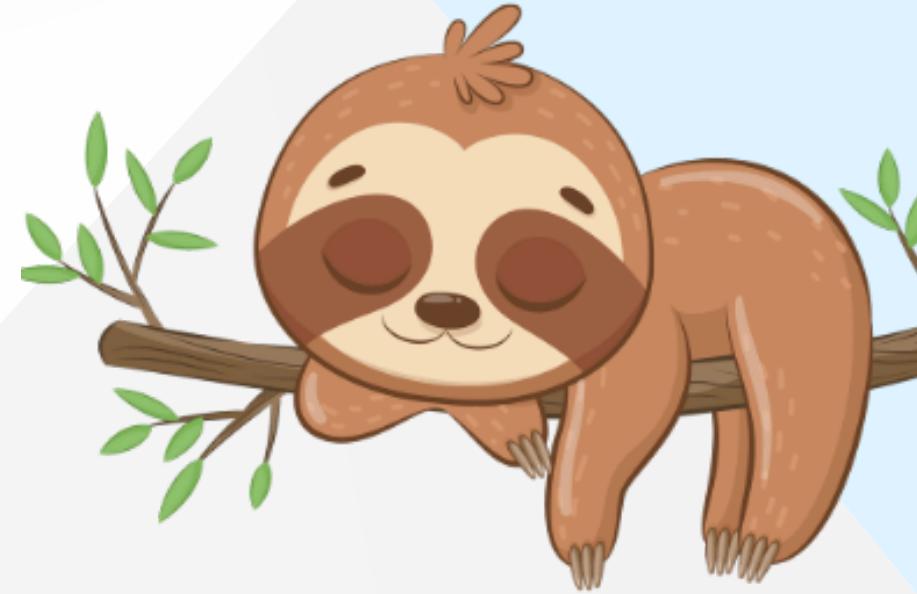
  expect(user.name).toBe("Alice");
  expect(user.age).toBe(30);
  expect(user.isActive).toBe(true);
  expect(user.permissions.length).toBeGreaterThan(0);
});
```

Śpiący test

- ◆ używa `sleep()` lub innych sztucznych opóźnień, zamiast prawidłowego oczekiwania na zakończenie operacji asynchronicznych
- ◆ spowalnia cały proces testowania
- ◆ czasami testy przechodzą, a czasami nie, zależnie od tego, jak wolno działa system

Przykładowy kod:

```
test("Sleepy test", (done) => {
  setTimeout(() => {
    expect(true).toBe(true);
    done();
  }, 5000); // Sztuczne czekanie 5 sekund!
});
```



Martwy kod testowy

- ▶ testy, których nikt nie uruchamia
- ▶ testy, które są zawsze wykomentowane lub ignorowane
- ▶ testy, które nie mogą się nie powieść (np. `assert(true)`)



Statyczna analiza kodu

- ▶ proces analizy kodu źródłowego bez jego uruchamiania
- ▶ wykorzystywana do wykrywania błędów, luk w zabezpieczeniach i problemów ze zgodnością z wytycznymi
- ▶ może być stosowana na etapie wczesnego developmentu
- ▶ wspiera utrzymanie jakości kodu i poprawia jego czytelność

Jak statyczna analiza kodu wspiera wytwarzanie oprogramowania i testowanie?

- ◀ **wczesne wykrywanie błędów** – zapobiega kosztownym poprawkom na późniejszych etapach
- ◀ **zwiększenie bezpieczeństwa** – identyfikacja potencjalnych podatności w kodzie
- ◀ **standaryzacja kodu** – egzekwowanie określonych zasad kodowania
- ◀ **wsparcie dla refaktoryzacji** – ułatwia optymalizację kodu
- ◀ **redukcja dłużu technicznego** – ułatwia utrzymanie kodu w dłuższym czasie

ESLint – narzędzie do analizy kodu Javascript

- ▶ statyczny analizator kodu Javascript / Typescript
- ▶ pomaga w identyfikowaniu błędów i niespójności w kodzie
- ▶ pozwala definiować własne reguły oraz korzystać z predefiniowanych zestawów reguł
- ▶ ma możliwość automatycznego poprawiania niektórych problemów
- ▶ popularne w ekosystemie frontendowym

Przykładowy kod:

```
{  
  "extends": "eslint:recommended",  
  "rules": {  
    "no-console": "warn",  
    "eqeqeq": "error"  
  }  
}
```

Prettier – narzędzie do formatowania kodu

- ▶ automatyczne formatowanie kodu zgodnie ze standardami
- ▶ zapewnia jednolity styl kodu w całym projekcie
- ▶ obsługuje Javascript, Typescript, HTML, CSS i inne języki
- ▶ integruje się z ESLint, aby zapobiegać konfliktom formatowania
- ▶ może działać jako **pre-commit** hook, np. w Git

Przykładowy kod:

```
{  
  "singleQuote": true,  
  "trailingComma": "es5"  
}
```

SonarQube – platforma do analizy jakości kodu

- ▶ skanuje kod w poszukiwaniu błędów, podatności i problemów z jakością
- ▶ obsługuje wiele języków programowania (Java, Javascript, Python itp.)
- ▶ generuje raporty z analizą kodu i wskaźnikami jakości
- ▶ integruje się z CI/CD (np. Jenkins, GitHub Actions)
- ▶ pomaga w monitorowaniu długu technicznego i utrzymaniu wysokiej jakości kodu

Przykładowe metryki SonarQube:

- ▶ pokrycie kodu testami
- ▶ złożoność cyklotomyczna
- ▶ duplikacja kodu
- ▶ wykryte błędy i podatności

Dodatkowe pytania?





KAHoot

Dołącz na kahoot.it
lub przez aplikację [Kahoot!](#)

XXX

Ankieta

czyli jak mi poszło?

<https://sages.link/725808>

xxx



Dziękuję za uwagę

Mateusz Jabłoński

mail@mateuszjablonski.com

mateuszjablonski.com

STACJA.IT

JABŁOŃSKI

sages