infoShare ACADEMY

TESTY FRONTEND

infoShare Academy

infoShareAcademy.com

HELLO

Kamil Richert

Senior Software Engineer w Atlassian

infoShare ACADEMY

Testowanie oprogramowania

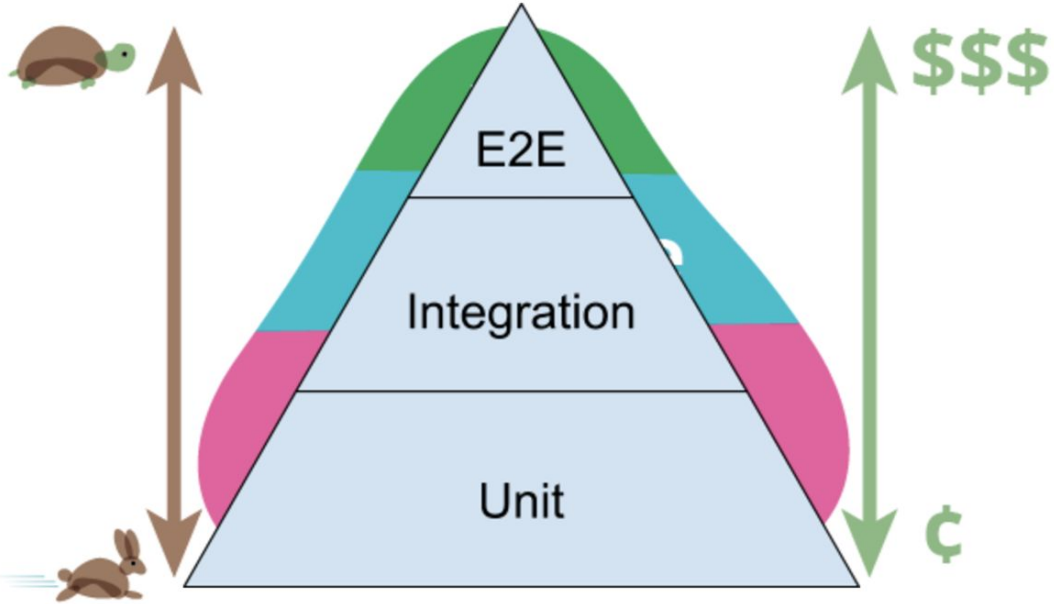
to proces związany z wytwarzaniem oprogramowania. Jest to część procesów zarządzania jakością oprogramowania.

Testowanie ma na celu weryfikację oraz walidację oprogramowania. Weryfikacja oprogramowania pozwala skontrolować, czy wytwarzane oprogramowanie jest zgodne ze specyfikacją.

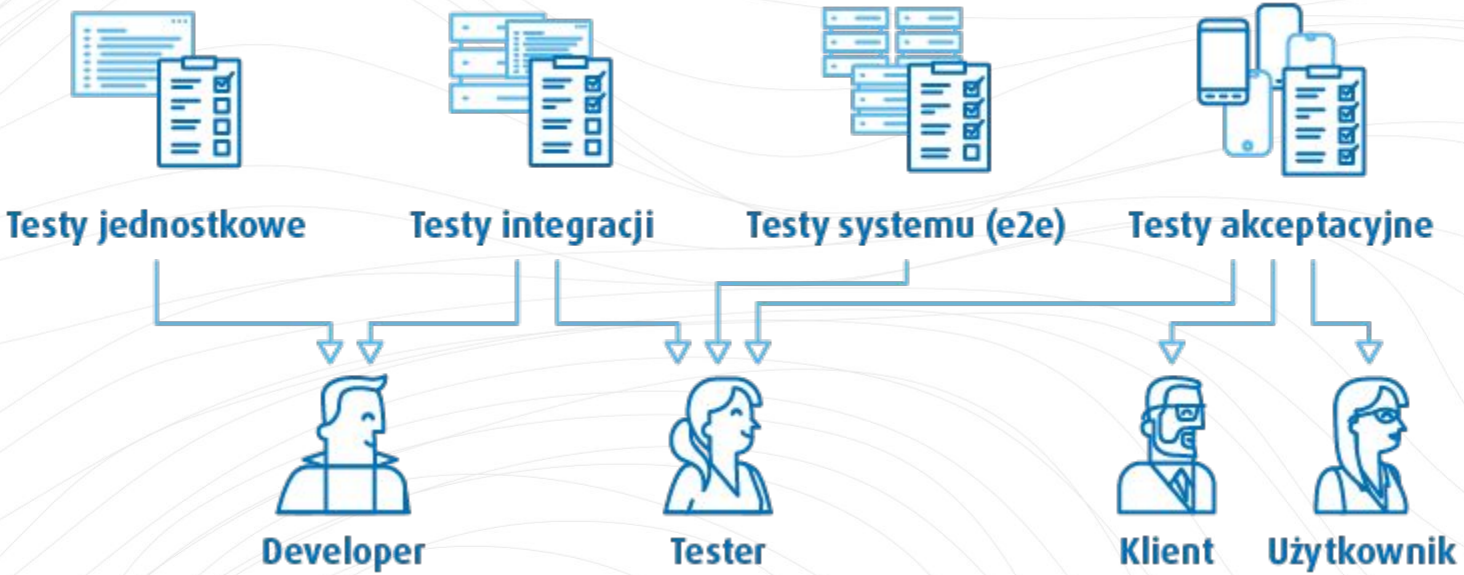
Rodzaje testów

- Testy jednostkowe
- Testy integracyjne
- Testy systemowe (e2e)
- Testy akceptacyjne

Piramida testów



Kto wykonuje testy?



Struktura testu - AAA pattern

arrange

W tej fazie przygotowujemy nasze środowisko testowe, czyli ustawiamy początkowe warunki dla testu. Może to obejmować inicjalizację obiektów, utworzenie potrzebnych zasobów czy wstrzyknięcie zależności. Wszystko to ma na celu zapewnienie stabilnego i spójnego stanu przed wykonaniem testu.

Struktura testu - AAA pattern

act

W tej fazie wykonujemy konkretną czynność lub operację, którą chcemy przetestować. Może to być wywołanie konkretnej metody, symulacja interakcji z interfejsem użytkownika lub jakakolwiek inna akcja, którą chcemy zbadać pod kątem poprawności działania.

Struktura testu - AAA pattern

assert

W tej fazie sprawdzamy, czy otrzymane wyniki działania są zgodne z oczekiwaniami. Porównujemy faktyczne wyniki z oczekiwanymi rezultatami i w przypadku, gdy występuje rozbieżność, zgłaszamy błąd. Może to obejmować porównanie wartości, sprawdzenie stanu obiektów lub weryfikację zachowania.

Zadanie

Zacznijmy od pisania prostych testów w czystym JS (bez żadnego frameworka).

Wykonaj zadania w folderze: *exercises*.

Testy jednostkowe

infoShareAcademy.com

Testy jednostkowe

Test jednostkowy (ang. unit test, test modułowy) **to zazwyczaj domena programistów,** ponieważ to oni zazwyczaj odpowiadają za tworzenie i utrzymanie unit testów. Na tym poziomie **sprawdzają, czy pojedyncza metoda / funkcja robi to, co powinna,** np. czy wywoływana jest odpowiednia akcja lub czy obliczenia są poprawne. Jest to najniższy z poziomów testów, same testy powinny być proste i wykonywać się bardzo szybko. Dlatego właśnie ten poziom testów powinien być podstawą automatycznej weryfikacji poprawnego działania oprogramowania.

Testy jednostkowe

Testowanie jednostkowe pomaga w zapewnieniu jakości kodu i zwiększa pewność, że nasza aplikacja działa poprawnie.

Korzyści testowania jednostkowego:

- Wczesne wykrywanie błędów
- Ułatwia refaktoryzację
- Zwiększa zaufanie do kodu
- Wspiera rozwój zespołowy


JEST


infoShare
ACADEMY

JEST

- Framework do testowania rozwijany przez Facebook'a
- Posiada własny test-runner → program do uruchamiania testów
- Posiada własny zbiór asercji (porównań którymi weryfikujemy wyniki testów)
- Daje nam możliwość mierzenia pokrycia testami
- Wbudowane mocki
- Obsługuje wyjątki (te celowe, które chcemy wytestować)
- Można nim testować projekty frontend'owe (używające np Angular'a, Vue czy React'a), czy projekty oparte o Node.js

JEST

Przydatne linki:

https://jestjs.io/docs/en/getting-started → cała sekcja “Introduction” to podstawy pracy z Jest

https://create-react-app.dev/docs/running-tests/ → testowanie aplikacji korzystających z create-react-app, gdzie Jest jest wykorzystany jako test runner i główną bibliotekę do testów.

JEST – przykładowy test

```
// math.test.js
const { sum, subtract } = require('./math');

test('sum adds two numbers correctly', () => {
  expect(sum(2, 3)).toBe(5);
});

test('subtract subtracts two numbers correctly', () => {
  expect(subtract(5, 2)).toBe(3);
});
```

JEST - matchers

Zamiast polegać na porównywaniu w czystym Javascript, które nie jest takie oczywiste, biblioteka JEST zapewnia nam zbiór matcherów do porównywania wyników naszych testów.

```
test('two plus two is four', () => {
        expect(2 + 2).toBe(4);
});

test('object assignment', () => {
        const data = {one: 1};
        data['two'] = 2;
        expect(data).toEqual({one: 1, two: 2});
});
```

Więcej: https://jestjs.io/docs/using-matchers

JEST - before i after

W plikach testowych Jest umieszcza metody globalne jak *describe* czy *test*. Jednak udostępnia też inne przydatne metody. Najbardziej popularnymi są *beforeEach* i *afterEach*, które wykonują zadaną funkcję przed lub po każdym teście.

```
beforeEach(() => {
        globalDatabase.insert({testData: 'foo'});
});

afterEach(() => {
        cleanUpDatabase(globalDatabase);
});
```

Więcej: https://jestjs.io/docs/api

JEST – testowanie asynchroniczności

Często mamy do czynienia z asynchronicznymi operacjami w aplikacjach React, takimi jak pobieranie danych z API.

```
test('the data is peanut butter', () => {
        return fetchData().then(data => {
                expect(data).toBe('peanut butter');
        });
});

test('the data is peanut butter', () => {
        return expect(fetchData()).resolves.toBe('peanut butter');
});
```

Więcej: https://jestjs.io/docs/asynchronous

Zadanie

Wykonajmy podobne testy do tych co napisaliśmy w czystym JS, ale tym razem z użyciem biblioteki JEST.

Wykonaj zadania w folderze: *jest-exercises*.

JEST - mockowanie funkcji

Czasami w teście potrzebujemy otrzymać konkretną wartość z jednej z zależności by sprawdzić poprawność działania funkcji / komponentu czy sprawdzić czy dana zależność została poprawnie użyta / wywołana. Do tego służy nam mockowanie, które pozwala nam na wymazanie rzeczywistej implementacji funkcji, przechwytywanie wywołań funkcji (i parametrów przekazywanych w tych wywołaniach), i umożliwienie konfiguracji w czasie testu zwracanych wartości.

```
test('post student name has called validate function', () => {
        const validateMock = jest.fn();

        postStudent('Kamil', validateMock);

        expect(validateMock).toHaveBeenCalledWith('Kamil');
});
```

Więcej: https://jestjs.io/docs/mock-functions

JEST - mockowanie implementacji

Mimo to zdarzają się przypadki, kiedy chcemy całkowicie zastąpić implementacje.

```
import { post } from ‘./server/post’;
jest.mock(‘./server/post’, () => {
        post: jest.fn();
});
const postMock = post;


test('should show success if post was successful', async () => {
        postMock.mockReturnValue(Promise.resolve());

        const actual = await addStudent(‘Kamil’);

        expect(actual).toEqual({ status: 200 });
});
```

Więcej: https://jestjs.io/docs/mock-functions#mock-implementations

JEST - snapshot

Testy snapshot są bardzo przydatnym narzędziem zawsze, gdy chcesz mieć pewność, że Twój UI nie zmieni się nieoczekiwanie.

Typowy test renderuje komponent interfejsu użytkownika, wykonuje snapshota, a następnie porównuje ją z zapisaną wartością (inline albo w pliku). Test zakończy się niepowodzeniem, jeśli snapshoty nie są zgodne.

```
import renderer from 'react-test-renderer';
import Link from '../Link';

it('renders correctly', () => {
        const tree = renderer
                .create(<Link page="http://www.facebook.com">Facebook</Link>)
                .toJSON();
        expect(tree).toMatchSnapshot();
});
```

Więcej: https://jestjs.io/docs/snapshot-testing

infoShare ACADEMY

Zadanie

Wykonajmy testy z użyciem mockowania i snapshotów.

Wykonaj zadania w folderze: *jest-mock-exercises*.

React testing library

infoShareAcademy.com

React testing library

- Nie jest samodzielną bibliteką/frameworkiem do testów (jak jest)
- Daje nam możliwość, podczas testów
  - wyrenderowania komponentu React
  - interakcji z komponentem
  - odczytywania wartości jak z drzewa DOM
- Zbudowane w oparciu o DOM Testing Library

React testing library

Przydatne linki:

https://testing-library.com/docs/react-testing-library/intro → cała sekcja React Testing Library opisuje podstawy pracy z biblioteką

https://testing-library.com/docs/dom-testing-library/api-queries → metody budujące wygodne zapytania do DOM

https://testing-library.com/docs/dom-testing-library/api-events → odpalanie eventów

https://create-react-app.dev/docs/running-tests/#react-testing-library → react-testing-library jest domyślnie w create-react-app

React testing library - przykładowy test

```
import { render, screen } from '@testing-library/react';
import MyComponent from './MyComponent';

test('renders component correctly', () => {
  render(<MyComponent />);
  const headingElement = screen.getByRole('heading', { name: /hello/i });
  expect(headingElement).toBeInTheDocument();
});
```

React testing library - podstawowe metody

- **render**: Renderuje komponent React do drzewa DOM.
- **screen.getBy**...: Pobiera element z drzewa DOM na podstawie różnych kryteriów, takich jak etykieta, rolę, tekst, itp. https://testing-library.com/docs/queries/about

React testing library - getBy, queryBy, findBy

Type of Query	0 Matches	1 Match	>1 Matches	Retry (Async/Await)
Single Element				
`getBy...`	Throw error	Return element	Throw error	No
`queryBy...`	Return `null`	Return element	Throw error	No
`findBy...`	Throw error	Return element	Throw error	Yes
Multiple Elements				
`getAllBy...`	Throw error	Return array	Return array	No
`queryAllBy...`	Return `[]`	Return array	Return array	No
`findAllBy...`	Throw error	Return array	Return array	Yes

React testing library - priorytety

Twój test powinien w jak największym stopniu przypominać interakcję użytkowników z Twoim kodem (komponentem, stroną itp.). Twórcy biblioteki zalecają następującą kolejność priorytetów:

1. Queries Accessible to Everyone Queries
    i. getByRole
    ii. getByLabelText
    iii. getByPlaceholderText
    iv. getByText
    v. getByDisplayValue
2. Semantic Queries HTML5 and ARIA compliant selectors.
    i. getByAltText
    ii. getByTitle
3. Test IDs
    i. getByTestId

Więcej: https://testing-library.com/docs/queries/about/#priority

React testing library - testowanie interakcji

React Testing Library oferuje dwie metody do testowania interakcji użytkownika: **fireEvent** i **user event**, co symuluje zdarzenia, takie jak kliknięcie.

fireEvent wywołuje zdarzenia DOM, podczas gdy **user event** symuluje pełne interakcje, które mogą wywołać wiele zdarzeń i po drodze przeprowadzić dodatkowe kontrole.

W związku z powyższym, w większości przypadków powinniśmy korzystać z **user event**.

React testing library - testowanie interakcji

```
import { Counter } from './Counter';
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event'

test('should have increase counter when increase clicked', async () => {
    render(<Counter />);

    await userEvent.click(screen.getByRole('button', { name: 'increase' }))

    expect(screen.getByText('1')).toBeInTheDocument();
});
```

infoShare ACADEMY

React testing library - testowanie interakcji

```
import { render, screen, fireEvent } from '@testing-library/react';
import Counter from './Counter';

test('increments counter on button click', () => {
  render(<Counter />);
  const counterElement = screen.getByTestId('counter');
  const buttonElement = screen.getByRole('button', { name: /increment/i });

  fireEvent.click(buttonElement);
  expect(counterElement.textContent).toBe('1');
});
```

React testing library - snapshots

Snapshoty są przydatnym narzędziem do automatycznego porównywania wyników testów z oczekiwanymi wynikami.
JEST automatycznie generuje snapshoty, które są zapisywane w plikach.

```
import { render } from '@testing-library/react';
import MyComponent from './MyComponent';

test('renders component correctly', () => {
  const { container } = render(<MyComponent />);
  expect(container.firstChild).toMatchSnapshot();
});
```

Zadanie

Wykonajmy testy z użyciem biblioteki react-testing-library.

Wykonaj zadania w folderze: *react-testing-library-exercises*.

Testy integracyjne

infoShareAcademy.com

Testy integracyjne

Testy integracyjne natomiast **mogą być domeną zarówno testerów, jak i developerów.** Podstawowym wyznacznikiem będzie tutaj to, na jakim poziomie sprawdzamy integrację. Jeżeli sprawdzamy **integrację pojedynczych metod i funkcji** (po sprawdzeniu testami jednostkowymi, czy one same działają), to raczej będzie to domena programistów. Jeżeli jednak będziemy testować integrację na wyższym poziomie, np. **integracje między dwoma modułami aplikacji,** to pracę tę może, a często powinien, wykonywać tester. Wynika to z faktu, że programista testujący swój własny kod testuje swoją interpretację wymagań, ale niekoniecznie tę prawidłową. Nie jest to przytyk w kierunku programistów, ale raczej wskazanie, dlaczego nie powinno testować się własnych rozwiązań.

Testy e2e

infoShareAcademy.com

Testy e2e

To jednak testy funkcjonalne na poziomie systemowym stanowią ten domyślny rodzaj testów w oczach większości osób. Testowanie systemowe jest to **sprawdzanie funkcjonalności oprogramowania na zintegrowanym środowisku,** gdzie funkcjonalności możemy testować **od początku do końca (end-to-end).** Jest to też poziom, który jest najbardziej intuicyjny dla początkujących testerów czy osób z testowaniem niezwiązanych. Ale to nie wszystko – skoro jest to testowanie funkcjonalności od początku do końca, to są to również bardzo istotne testy z punktu widzenia biznesu i interesariuszy, bo użytkownicy końcowi będą korzystać z oprogramowania w bardzo podobny sposób. Często dopiero teraz możemy zwrócić uwagę na poprawność interfejsu użytkownika (zarówno zgodność z makietami, jak i prostotę jego obsługi) oraz kwestię poprawności prezentowania danych użytkownikowi. Na tym poziomie możemy w końcu sprawdzić dostępność produktu dla osób z niepełnosprawnościami, co jest szczególnie istotne dla testerów pracujących w instytucjach publicznych.

infoShareAcademy.com

infoShare ACADEMY

Cypress

- Kompleksowe rozwiązanie do testów e2e czy integracyjnych
- Posiada własny test runner
- Niezależne od aplikacji którą testuje (nie jest istotna technologi użyta w testowanej aplikacji)
- Testy, które “klikają” po stronie - opisujemy te zachowania w JavaScript
- Cypress dba o to aby flow naszych testów podążał za tym jak zmieniają się elementy na stronie - nie musimy ręcznie “czekać” na wykonanie operacji
- Własna biblioteka asercji

Cypress

Przydatne linki:

https://docs.cypress.io/guides/getting-started/installing-cypress.html → cała sekcja Getting Started, pokrywa wszystkie podstawy pracy z Cypress

https://www.youtube.com/watch?v=LcGHiFnBh3Y → kompleksowo podstawy Cypress w formie video

Cypress – przykładowy test

```
// przykladowy test
describe('My First Test', () => {
  it.skip('clicking "type" shows the right headings', () => {
    cy.visit('https://example.cypress.io');

    cy.pause();

    cy.contains('type').click();

    // Should be on a new URL which includes '/commands/actions'
    cy.url().should('include', '/commands/actions');

    // Get an input, type into it and verify that the value has been updated
    cy.get('.action-email')
      .type('fake@email.com')
      .should('have.value', 'fake@email.com');
  });
});
```

Cypress - przydatne metody

cy.contains() - pobiera element DOM zawierający tekst. Elementy DOM mogą zawierać więcej niż żądany tekst i nadal być zwracane.
https://docs.cypress.io/api/commands/contains

cy.get() - pobiera jeden lub więcej elementów DOM według selektora lub aliasu.
https://docs.cypress.io/api/commands/get

cy.first() - zwraca pierwszy element z listy elementów DOM.
https://docs.cypress.io/api/commands/first

cy.filter() - zwraca elementy DOM pasujące do określonego selektora.
https://docs.cypress.io/api/commands/filter

cy.not() - odfiltrowuje elementy DOM ze zbioru elementów DOM.
https://docs.cypress.io/api/commands/not

Cypress - asercje

cy.should() - sprawdza czy dany element powinien mieć jakąś wartość / być w określonym stanie
https://docs.cypress.io/api/commands/should

cy.and() - dodatkowy warunek do innych metod
https://docs.cypress.io/api/commands/and

Tworzone asercje są automatycznie ponawiane w ramach poprzedniego polecenia, dopóki się nie spełnią lub upłynie limit czasu.

W tych metodach podajemy string z wartość zgodną dla Chai (biblioteka asercji BDD/TDD dla przeglądarek, którą można doskonale połączyć z dowolnym frameworkiem testowym JavaScript)
https://docs.cypress.io/guides/references/assertions#Chai

Chainers mogą być używanie również w **get** i **contains**.

Testy akceptacyjne

infoShareAcademy.com

Testy akceptacyjne

Testy akceptacyjne to ostatni etap testów, który jest wykonywany zazwyczaj na gotowym produkcie, przez użytkowników końcowych, przez klienta lub z jego udziałem. Testy funkcjonalne przeprowadza się na tym poziomie, aby **ostatecznie zweryfikować i zatwierdzić, że powstały produkt spełnia stawiane mu wymagania.** Jest to także ostatni etap na znalezienie defektów przed przekazaniem produktu użytkownikom końcowym i ostatni dzwonek by sprawdzić zgodność produktu z wymaganiami wynikającymi z legislacji.

Zadanie

Wykonajmy testy z użyciem biblioteki cypress.

Wykonaj zadania w folderze: *cypress-exercises*.

Dzięki!

Znajdziecie mnie:
https://www.linkedin.com/in/kamil-richert/
https://github.com/krichert