Projektowanie Aplikacji Internetowych
Wykład 8

Testowanie aplikacji frontendowych

Mateusz Pawełkiewicz

1. Wprowadzenie do testowania aplikacji frontendowych

Testowanie jest kluczowym elementem procesu tworzenia oprogramowania, zapewniającym jego wysoką jakość, niezawodność i zgodność z wymaganiami. W kontekście aplikacji frontendowych, testowanie pozwala na wykrycie błędów i problemów związanych z interfejsem użytkownika, logiką biznesową oraz interakcją z backendem.

1.1 Dlaczego testowanie jest ważne

- **Zapewnienie jakości**: Testy pomagają wykryć błędy na wczesnym etapie, co zmniejsza koszty ich naprawy.
- **Niezawodność**: Regularne testowanie gwarantuje, że aplikacja działa poprawnie w różnych scenariuszach.
- **Utrzymanie kodu**: Testy ułatwiają refaktoryzację kodu, zapewniając, że zmiany nie wprowadzają nowych błędów.
- **Dokumentacja**: Testy mogą służyć jako dokumentacja zachowania aplikacji.

1.2 Rodzaje testów w aplikacjach frontendowych

- **Testy jednostkowe (Unit Tests)**: Testują pojedyncze jednostki kodu (np. funkcje, metody) w izolacji.
- **Testy integracyjne (Integration Tests)**: Sprawdzają interakcje między różnymi modułami aplikacji.
- **Testy end-to-end (E2E Tests)**: Symulują rzeczywiste scenariusze użytkownika, testując aplikację jako całość.
- **Testy wizualne**: Porównują wygląd interfejsu użytkownika z oczekiwanym wzorcem.
- **Testy dostępności (Accessibility Tests)**: Sprawdzają, czy aplikacja jest dostępna dla osób z niepełnosprawnościami.

---

2. Testy jednostkowe w JavaScript

Testy jednostkowe skupiają się na weryfikacji poprawności pojedynczych funkcji lub komponentów w izolacji od reszty aplikacji.

2.1 Narzędzia do testowania jednostkowego

2.1.1 Jest

- **Opis**: Jest to framework testowy stworzony przez Facebooka, popularny w społeczności React.
- **Funkcje**:
  - Wbudowany runner testów i asercje.
  - Obsługa mocków i snapshotów.
  - Integracja z Babel i TypeScript.
- **Instalacja**:

```
npm install --save-dev jest
```

2.1.2 Mocha

- **Opis**: Elastyczny framework testowy dla Node.js i przeglądarki.
- **Funkcje**:
  - Obsługa różnych bibliotek asercji (np. Chai).
  - Konfigurowalność i rozszerzalność.
- **Instalacja**:

```
npm install --save-dev mocha
```

2.1.3 Chai

- **Opis**: Biblioteka asercji, która może być używana z Mocha.
- **Tryby asercji**:
  - **Assert**: Tradycyjne asercje.
  - **Expect**: Czytelne asercje w stylu BDD.
  - **Should**: Asercje w stylu łańcuchowym.
- **Instalacja**:

```
npm install --save-dev chai
```

2.2 Pisanie testów jednostkowych

2.2.1 Struktura testów w Jest

- **Bloki `describe`**: Grupują powiązane testy.
- **Bloki `test` lub `it`**: Definiują pojedyncze przypadki testowe.
- **Asercje**: Sprawdzają, czy wynik jest zgodny z oczekiwaniami.

Przykład:

```
// funkcja do testowania
function suma(a, b) {
  return a + b;
}

// test
describe('Funkcja suma', () => {
  test('powinna poprawnie dodawać dwie liczby', () => {
    expect(suma(2, 3)).toBe(5);
  });
});
```

2.2.2 Mockowanie funkcji i modułów

- **Mockowanie**: Tworzenie fikcyjnych implementacji funkcji lub modułów, aby izolować testowany kod.
- **Użycie w Jest**:

```
jest.mock('./modul', () => {
  return {
```

```
    funkcja: jest.fn(() => 'mockowana wartość'),
  };
});
```

2.2.3 Snapshot Testing

- **Opis**: Porównuje aktualny wynik renderowania komponentu z wcześniej zapisanym wzorcem (snapshot).
- **Użycie w React**:

```
import renderer from 'react-test-renderer';

test('komponent renderuje się poprawnie', () => {
  const tree = renderer.create(<Komponent />).toJSON();
  expect(tree).toMatchSnapshot();
});
```

---

3. Testy integracyjne

Testy integracyjne sprawdzają, czy różne moduły aplikacji współpracują ze sobą poprawnie.

3.1 Narzędzia do testów integracyjnych

3.1.1 Enzyme

- **Opis**: Biblioteka stworzona przez Airbnb do testowania komponentów React.
- **Funkcje**:
    - Renderowanie komponentów w izolacji lub z ich dziećmi.
    - Manipulacja i interakcja z drzewem DOM komponentu.
- **Instalacja**:

```
npm install --save-dev enzyme enzyme-adapter-react-16
```

3.1.2 React Testing Library

- **Opis**: Biblioteka skupiająca się na testowaniu zachowania aplikacji z perspektywy użytkownika.
- **Funkcje**:
    - Umożliwia interakcje z komponentami w sposób zbliżony do rzeczywistego użytkowania.
    - Promuje dobre praktyki testowania.
- **Instalacja**:

```
npm install --save-dev @testing-library/react
```

3.2 Pisanie testów integracyjnych z React Testing Library

3.2.1 Renderowanie komponentów

- **Renderowanie komponentu**:

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

test('renderuje komponent', () => {
  render(<Komponent />);
});
```

3.2.2 Wyszukiwanie elementów

- **Metody wyszukiwania**:
  - getByText: Znajduje element na podstawie jego tekstu.
  - getByRole: Znajduje element na podstawie jego roli (np. button, textbox).
  - getByLabelText: Znajduje element powiązany z etykietą.

Przykład:

```
const { getByText } = render(<Komponent />);
const przycisk = getByText('Kliknij mnie');
```

3.2.3 Symulowanie interakcji

- **Symulacja kliknięcia**:

```
import { fireEvent } from '@testing-library/react';

fireEvent.click(przycisk);
```

- **Sprawdzanie efektów interakcji**:

```
expect(getByText('Tekst po kliknięciu')).toBeInTheDocument();
```

---

4. Testy end-to-end (E2E)

Testy E2E symulują rzeczywiste interakcje użytkownika z aplikacją, testując cały system od początku do końca.

4.1 Narzędzia do testów E2E

4.1.1 Cypress

- **Opis**: Nowoczesne narzędzie do testów E2E, które działa w przeglądarce.
- **Funkcje**:
  - Szybkie i niezawodne testy.
  - Interaktywny interfejs do debugowania.
  - Automatyczne oczekiwanie na elementy DOM.
- **Instalacja**:

  ```
  npm install --save-dev cypress
  ```

4.1.2 Selenium WebDriver

- **Opis**: Popularne narzędzie do automatyzacji przeglądarek, obsługuje wiele języków programowania.
- **Funkcje**:
  - Obsługa różnych przeglądarek i platform.
  - Możliwość integracji z różnymi frameworkami testowymi.

4.2 Pisanie testów E2E z Cypress

4.2.1 Struktura testu

- **Plik testowy**: Pliki testowe w Cypress mają rozszerzenie .spec.js.
- **Bloki describe i it**: Podobne do testów jednostkowych.

4.2.2 Przykładowy test

```
describe('Strona główna', () => {
  it('powinna wyświetlać tytuł', () => {
    cy.visit('http://localhost:3000');
    cy.contains('Witaj, świecie!');
  });
});
```

4.2.3 Interakcje i asercje

- **Nawigacja**:

  ```
  cy.visit('http://localhost:3000');
  ```

- **Wyszukiwanie elementów**:

  ```
  cy.get('input[name="email"]');
  ```

- **Interakcje**:

  ```
  cy.get('button').click();
  ```

- **Asercje**:

```
cy.url().should('include', '/dashboard');
cy.get('.error').should('not.exist');
```

---

5. Testy wizualne

Testy wizualne porównują wygląd interfejsu użytkownika z oczekiwanym wzorcem, wykrywając niezamierzone zmiany w stylach lub układzie.

5.1 Narzędzia do testów wizualnych

5.1.1 Storybook

- **Opis**: Narzędzie do budowy i testowania komponentów UI w izolacji.
- **Funkcje**:
  - Tworzenie interaktywnej dokumentacji komponentów.
  - Integracja z narzędziami do testów wizualnych.
- **Instalacja**:

```
npx -p @storybook/cli sb init
```

5.1.2 Chromatic

- **Opis**: Usługa chmurowa do automatycznych testów wizualnych na podstawie Storybooka.
- **Funkcje**:
  - Generowanie screenshotów komponentów.
  - Wykrywanie zmian wizualnych między wersjami.
- **Integracja**:
  - Wymaga konfiguracji konta i połączenia z repozytorium kodu.

5.2 Proces testowania wizualnego

- **Krok 1**: Zdefiniowanie historii (stories) dla komponentów w Storybooku.
- **Krok 2**: Generowanie screenshotów dla każdej historii.
- **Krok 3**: Porównanie aktualnych screenshotów z poprzednimi.
- **Krok 4**: Analiza różnic i decyzja o akceptacji lub odrzuceniu zmian.

---

6. Testy dostępności (Accessibility Tests)

Zapewnienie dostępności aplikacji jest kluczowe dla umożliwienia korzystania z niej osobom z niepełnosprawnościami.

6.1 Narzędzia do testów dostępności

6.1.1 Axe

- **Opis**: Silnik testów dostępności stworzony przez Deque Systems.
- **Funkcje**:
  - Automatyczne wykrywanie problemów z dostępnością.
  - Integracja z narzędziami testowymi i przeglądarkami.
- **Instalacja**:

```
npm install --save-dev axe-core
```

6.1.2 Lighthouse

- **Opis**: Narzędzie od Google do analizy jakości stron internetowych, w tym dostępności.
- **Funkcje**:
  - Generowanie raportów z oceną dostępności.
  - Sugestie dotyczące poprawy.
- **Użycie**:
  - Dostępne jako wtyczka do Chrome lub z linii komend.

6.2 Integracja testów dostępności z testami jednostkowymi

- **Przykład z użyciem React Testing Library i Axe**:

```
import { render } from '@testing-library/react';
import { toHaveNoViolations } from 'jest-axe';
import { axe } from 'jest-axe';

expect.extend(toHaveNoViolations);

test('komponent jest dostępny', async () => {
  const { container } = render(<Komponent />);
  const wyniki = await axe(container);
  expect(wyniki).toHaveNoViolations();
});
```

---

7. Kontrola jakości i integracja ciągła

Automatyzacja procesu testowania i integracja z pipeline CI/CD zapewnia stałe utrzymanie wysokiej jakości kodu.

7.1 Continuous Integration (CI)

- **Opis**: Proces automatycznego budowania i testowania kodu po każdej zmianie w repozytorium.
- **Narzędzia**:

- **Jenkins**: Serwer automatyzacji open-source.
- **GitHub Actions**: Wbudowane w GitHub narzędzie do automatyzacji.
- **Travis CI**: Usługa chmurowa do CI.

7.2 Konfiguracja pipeline testowego

- **Kroki**:
  1. **Checkout**: Pobranie kodu z repozytorium.
  2. **Instalacja zależności**:

     ```
     npm install
     ```

  3. **Uruchomienie testów jednostkowych**:

     ```
     npm test
     ```

  4. **Uruchomienie testów E2E**:

     ```
     npm run test:e2e
     ```

  5. **Generowanie raportów**.

7.3 Monitorowanie jakości kodu

- **Narzędzia**:
  - **SonarQube**: Platforma do analizy jakości kodu.
  - **Code Climate**: Usługa chmurowa monitorująca jakość i bezpieczeństwo kodu.
- **Metryki**:
  - **Pokrycie kodu testami**: Procent kodu, który jest testowany.
  - **Liczba błędów i ostrzeżeń**.
  - **Złożoność kodu**.

---

8. Strategie testowania w zespołach agile

W metodykach zwinnych testowanie jest integralną częścią procesu tworzenia oprogramowania.

8.1 Test-Driven Development (TDD)

- **Opis**: Podejście, w którym testy są pisane przed implementacją funkcjonalności.
- **Kroki**:
  1. **Napisz test**: Definiuje oczekiwane zachowanie.
  2. **Uruchom test**: Powinien nie przejść (czerwony).
  3. **Napisz kod**: Implementacja minimalna, aby test przeszedł.
  4. **Uruchom test**: Powinien przejść (zielony).
  5. **Refaktoryzacja**: Ulepszenie kodu przy zachowaniu przechodzenia testów.

8.2 Behavior-Driven Development (BDD)

- **Opis**: Koncentruje się na zachowaniu aplikacji z perspektywy użytkownika.
- **Narzędzia**:
  - **Cucumber**: Pozwala na pisanie testów w języku naturalnym (Gherkin).
- **Przykład scenariusza**:

```
Funkcja: Logowanie
  Jako użytkownik
  Chcę się zalogować
  Aby uzyskać dostęp do panelu

  Scenariusz: Użytkownik loguje się poprawnie
    Mając użytkownika z nazwą "Jan" i hasłem "tajne"
    Kiedy wpiszę nazwę "Jan" i hasło "tajne"
    I kliknę przycisk "Zaloguj"
    Wtedy powinienem zobaczyć stronę "Panel"
```

8.3 Continuous Testing

- **Opis**: Stałe testowanie na każdym etapie rozwoju aplikacji.
- **Zalety**:
  - Wczesne wykrywanie błędów.
  - Szybsze dostarczanie wartości dla klienta.
- **Praktyki**:
  - Automatyzacja testów.
  - Integracja testów z CI/CD.