Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI

KIERUNEK STUDIÓW
INFORMATYKA

MATERIAŁY DO ZAJĘĆ LABORATORYJNYCH

Szkielety programistyczne w aplikacjach internetowych

dr Mariusz Dzieńkowski

Lublin 2022

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

LABORATORIUM 10. TESTOWANIE APLIKACJI

Cel laboratorium:

Poznanie podstawowych sposobów testowania aplikacji i ich praktyczna realizacja. Nabycie umiejętności debugowania aplikacji.

Zakres tematyczny zajęć:

- Poznanie narzędzi do realizacji testów aplikacji internetowych.
- Wykonanie podstawowych testów aplikacji www.
- Praktyczna realizacja testów na aplikacji opartej na bibliotece React: testy migawkowe, testy stanu (ang. state) oraz właściwości (ang. props), test komponentów z zaczepem (ang. hook).

Pytania kontrolne:

a) Wymień rodzaje testów przeprowadzanych na aplikacjach internetowych.
b) Wymień narzędzia do realizacji testów aplikacji www.
c) Na czym polegają testy migawkowe?

Testowanie polega na weryfikacji poprawności działania aplikacji. Testy mają na celu naśladowanie sposobu działania, sprawdzają czy zaimplementowane funkcjonalności aplikacji działają zgodnie z wcześniejszymi założeniami i oczekiwaniami twórców oprogramowania.

Istnieje kilka rodzajów testów. Najważniejsze z nich to:

- Testy jednostkowe - polegają na weryfikacji działania pojedynczych elementów. Wykonywany jest fragment aplikacji, a otrzymany wynik jest porównywany z oczekiwanym. Testom podlega zarówno pozytywny jak i negatywny scenariusz.
- Testy integracyjne - sprawdzają interakcje pomiędzy komponentami aplikacji.
- Testy end to end - łączą zazwyczaj wiele testów jednostkowych i integracyjnych. Zazwyczaj odbywają się w środowisku przeglądarki i testują działanie aplikacji od początku do końca.

Zadanie 10.1. Testowanie kodu JavaScript za pomocą biblioteki Jest

Etap 1. Czynności przygotowawcze

- Utworzenie pliku package.json oraz jego modyfikacja oraz instalacja biblioteki Jest

```
npm init –y
npm i jest --save-dev
//package.json
"scripts": {
"test": "jest"
},
```

- Utworzenie pliku filterTest.js zawierającego implementecję funkcji oraz blok *describe* testu

Listing 10.1 *Funkcja filterByTerm*

```
function filterByTerm(inputArr, searchTerm) {
  return inputArr.filter(function(arrayElement) {
    return arrayElement.url.match(searchTerm)
  })
}

describe("Funkcja filtrująca", () => {
  test("filtrowanie na podstawie wyszukiwanego terminu (link)", () => {
    const input = [
      { id: 1, planeta: "Merkury", url: "https://pl.wikipedia.org/wiki/Merkury" },
      { id: 2, planeta: "Wenus", url: "https://pl.wikipedia.org/wiki/Wenus" },
      { id: 3, planeta: "Ziemia", url: "https://pl1.wikipedia.org/wiki/Ziemia" }
    ]

    const output = [{ id: 3, planeta: "Ziemia", url: "https://pl1.wikipedia.org/wiki/Ziemia" }]

    expect(filterByTerm(input, "Ziemia")).toEqual(output)
  })
})
```

describe – metoda, blok zawierający jeden lub więcej powiązanych testów; przyjmuje ciąg znaków do opisu zestawu testów oraz funkcję wywołania zwrotnego

test – funkcja, która jest właściwym blokiem testowym

expect – sprawdzenie czy funkcja filtrująca zwraca oczekiwany wynik po wywołaniu

- Uruchomić testowanie

  ```
  npm test
  ```

- Wykonać modyfikację funkcji *filterByTerm()*, tak aby nie była wrażliwa na wielkość liter szukanego ciągu *searchTerm*
- Zaimplementować funcję *filterByTerm2()*, która będzie przeszukiwała pole "planeta". Następnie do bloku *describe* należy dodać drugi test, w który zostanie sprawdzona zdefiniowana funkcja

Zadanie 10.2. Testowanie API aplikacji opartej na szkielecie Express za pomocą bibliotek Jest oraz SuperTest

Etap 1. Struktura projektu aplikacji oraz konfiguracja narzędzia Jest

```
|-- models
  |-- Post.js
|-- index.js
|-- package-lock.json
|-- package.json
|-- routes.js
```

```
|-- index.test.js
|-- jest.config.js
```

Na początku należy zorganizować środowisko testowe poprzez zainstalowanie zależości takich jak:

- SuperTest - bibliotekę testującą pozwalającą wykonać zautomatyzowany kod testowy
- Jest – jendą z najprostszych bibliotek testujących języka JavaScript

```
npm i --save-dev jest supertest
```

Ustawienie polecenia testowego w pliku package.json

```
 "scripts": {
   "test": "jest"
 },
```

Ustawienie biblioteki Jest, które spowoduje, że testy będą uruchamiane wyłącznie w środowisku Node (plik jest.config.js)

```
module.exports = {
 testEnvironment: "node",
}
```

Etap 2. Przygotowanie testu – implementacja funkcji pomocniczej, która pozwoli na utworzenie instancji aplikacji expressa bez jej uruchamiania

Listing 10.2 *Plik server.js*

```
const express = require("express")
const routes = require("./routes")

function createServer() {
        const app = express()
        app.use(express.json())
        app.use("/api",routes)
        return app
}

module.exports = createServer
```

Etap 3. Przygotowanie testu – plik server.test.js

Utworzenie pliku testowego JavaScript. Narzędzie Jest domyślnie szuka plików, które pasują do *.test.js. Na listingu 10.3 wykorzystano funkcje beforeEach i afterEach, które będą wywoływane przed i po każdym przypadku testowym. W tym przypadku będą dwie czynnośći: podłączenie do bazy MongoDB oraz usunięcie wszystkich danych i zakończenie testu. Na końcu zostanie zainicjowany serwer w zmiennej *app*.

Listing 10.3 *Plik testowy server.test.js*

```
const mongoose = require("mongoose")
```

```
const createServer = require("./server")

beforeEach((done) => {
	mongoose.connect(
		"mongodb://localhost:27017/postsdb",
		{ useNewUrlParser: true },
		() => done()
	)
})

afterEach((done) => {
	mongoose.connection.db.dropDatabase(() => {
		mongoose.connection.close(() => done())
	})
})

const app = createServer()
```

Etap 4. Testowanie tras

Do pliku testowego server.test.js należy dodać kod z listingu 10.4, za sprawą którego zostanie dodany do bazy danych nowy dokument. Następnie po wysłaniu żądania GET za pomocą SuperTest do punktu końcowego /api/posts należy oczekiwać odpowiedzi mającej status 200. Na koniec należy sprawdzić czy odpowiedź jest zgodna z danymi w bazie MongoDB.

Listing 10.4 *Przypadek testowy GET /api/users – pobieranie wszystkich postów*

```
//...
const Post = require("./Post")
// ...

test("GET /posts", async () => {
	const post = await Post.create({
		title: "Post 1",
		content: "Tekst postu 1",
	})

	await supertest(app)
		.get("/api/posts")
		.expect(200)
		.then((response) => {
			// Sprawdzenie typu i długości odpowiedzi
			expect(Array.isArray(response.body)).toBeTruthy()
			expect(response.body.length).toEqual(1)

			// Sprawdzenie danych odpowiedzi
			expect(response.body[0]._id).toBe(post.id)
			expect(response.body[0].title).toBe(post.title)
			expect(response.body[0].content).toBe(post.content)
```

```
        })
})
```

Po zaimplementowaniu powyższego przypadku testowego można uruchomić test:

```
npm test
```

Następnie należy zaimplementować kolejne przypadki testowe. Na listingu 10.5 znajduje się kod odpowiedzialny za utworzenie pojedynczego posta.

Listing 10.5 *Przypadek testowy POST /api/posts - generowanie pojedynczego posta*

```
test("POST /api/posts", async () => {
    const data = {
        title: "Post 1",
        content: "Tekst postu 1",
    }

    await supertest(app)
        .post("/api/posts")
        .send(data)
        .expect(200)
        .then(async (response) => {
            // Sprawdzenie odpowiedzi
            expect(response.body._id).toBeTruthy()
            expect(response.body.title).toBe(data.title)
            expect(response.body.content).toBe(data.content)

            // Sprawdzenie danych w bazie
            const post = await Post.findOne({ _id: response.body._id })
            expect(post).toBeTruthy()
            expect(post.title).toBe(data.title)
            expect(post.content).toBe(data.content)
        })
})
```

- Sugerując się powyższym kodem należy zaimplementować jeszcze jeden przypadek testowy - na przykład usunięcie postu z określonym id z bazy. W tym celu najpierw trzeba utworzyć post, następnie wywołać żądanie DELETE do punktu końcowego pojedynczego postu oraz na koniec sprawdzić czy post został usunięty z bazy.

Zadanie 10.3. Testowanie aplikacji opartej na szkielecie React

Etap 1. Testowanie migawek

Po utworzeniu aplikacji poprzez create-react-app, aplikacja posiada domyślną konfigurację testową.

Testowanie migawek polega na utworzeniu tzw. migawki, która zapisuje stan komponentu i przy każdym wykonaniu testu porównuje ją z aktualnym jego stanem. Pozwala to

kontrolować, w jaki sposób zmienia się komponent pomiędzy kolejnymi testami. Zabezpiecza to też przez niespodziewaną zmianą interfejsu użytkownika.

Listing 10.6 obrazuje komponent, na którym zostanie przeprowadzony test.

***Listing 10.6** Testowany komponent*

```
const Snapshot = () => {
  return (
    <div>
      <h1>Test migawkowy</h1>
      <h2>test</h2>
    </div>
  )
}
export default Snapshot
```

Dodatkowo należy utworzyć plik *Snapshot.test.js*

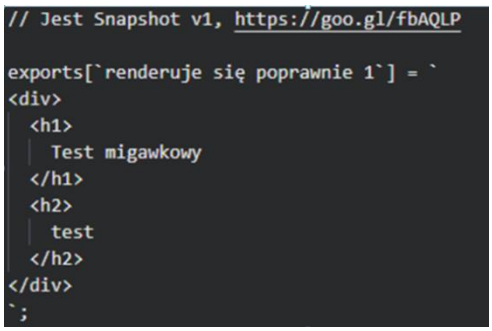
***Listing 10.7** Kod testu zawarty w pliku Snapshot.test.js*

```
import Snapshot from '../Shapshot'
import renderer from 'react-test-renderer'

it('renderuje się poprawnie', () => {
  const tree = renderer.create(<Snapshot/>).toJSON()
  expect(tree).toMatchSnapshot()
})
```

Każdy test umieszczany jest w bloku *it* lub *test* - mogą być one używane zamiennie. Bloki testowe pomagają grupować testy. Blok posiada dwa argumenty. Pierwszy to nazwa testu, którym jest ciąg znaków określający, co powinno się wydarzyć. Drugim argumentem jest funkcja wykonująca test.

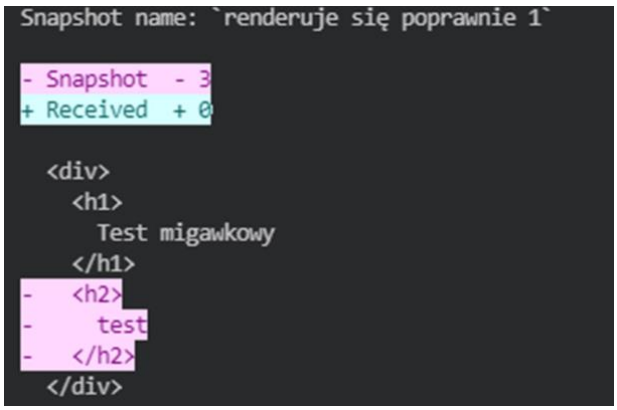
Po uruchomieniu testowym poleceniem *npm test* utworzony zostanie automatycznie folder *_snapshots_* zawiejący migawki. Rys. 10.1 przedstawia migawkę utworzoną na podstawie testowanego komponentu.



Rys. 10.1 Przykład migawki

Podczas kolejnych uruchomień testowych *Jest* porównuje wyrenderowany wynik z wcześniej przygotowaną migawką. Jeśli wszystko się zgadza, test przechodzi pomyślnie. Jeśli test zakończy się niepowodzeniem spowodowanym zmianą zawartości komponentu, należy naprawić błąd lub zaktualizować migawkę.

Usuwając z kodu z Listingu 10.6 linię: `<h2>test</h2>`
po ponownym uruchomieniu testu pokaże się informacja, że test zakończył się niepowodzeniem. Wskazana również zostanie różnica między migawką, a nowo wyrenderowanym widokiem.



Rys. 10.2 Test zakończony niepowodzeniem

W oknie wynikowym pojawi się również informacja, że po naciśnięciu przycisku ‘u’ nastąpi aktualizacja migawki, tzn. zostanie przygotowana nowa i to do niej od tej pory będą porównywane nowe rendery wykonywane w kolejnych testach.

Etap 2. Testy stanu (ang. state) oraz właściwości (ang. props)

Do testów zostanie użyty kod komponentu z Listingu 10.7. Użytkownik za pomocą przycisków zmienia stan – w pierwszym przypadku stan lokalny komponentu, a w drugim stan przekazany za pomocą właściwości props.

Listing 10.7 *Przykładowy komponent do testowania stanu*

```
import { useState } from "react"
const HookUseState = (props) => {
  const [state, setState] = useState("Wartość początkowa")

  const changeState = () => {
    setState("Inna wartość")
  }

  const changeName = () => {
    props.changeName()
  }

  return (
  <div>
        <button onClick={changeState}>Zmień stan</button>
        <p>{state}</p>
        <button onClick={changeState}>Zmień stan</button>
        <p>{props.state}</p>
  </div>
)
}
export default HookUseState
```

Zmodyfikowany zostanie również komponent *App.js*

***Listing 10.8** Kod komponentu głównego - App.js*

```
const [name, setName] = useState("zielony")

const changeName = () => {
   setName("czerwony")
}

return (
   <div clasName="App">
      <h1>Migawka</h1>
      <Snapshot />
            <h1>Hook useState</h1>
           <HookUseState name={name} changeName={changeName}>
```

Realizacja testu będzie wyglądała tak jak na Listingu 10.9.

***Listing 10.9** Kod procesu testowania*

```
import {render, fireEvent, cleanup} from '@testing-library/react'
import HookUseState from '../HookUseState'
import App from '../../App'
afterEach(cleanup)

it('po wciśnięciu przycisku zmienia sie wartość state', () => {
   const { getByText } = render(<HookUseState />)

   expect(getByText(/Wartość początkowa/i).textContent).toBe("Wartość początkowa")
   fireEvent.click(getByText("Zmień stan"))

   expect(getByText(/Inna wartość/i).textContent).toBe("Inna wartość")
})

it('po wciśnięciu przycisku zmienia sie wartość props() => {
   const { getByText } = render(<App> <HookUseState /> </App>)
   expect(getByText(/zielony/i).textContent).toBe("zielony")
   fireEvent.click(getByText("Zmień nazwę"))
   expect(getByText(/czerwony/i).textContent).toBe("czerwony")
})
```

Często podczas pisania testów pojawia się jakaś czynność, którą należy wykonać za każdym razem przed lub po ukończeniu testu. *Jest* udostępnia funkcje pomocnicze do obsługi tego przypadku. Jeśli istnieje czynność, którą potrzeba wykonać wielokrotnie dla wielu testów, można użyć funkcji *beforeEach* i *afterEach*. W przypadku tego testu za każdym razem po skończeniu testu wywołuje się funkcję *cleanup*, która odmontowuje komponent z kontenera i niszczy ten kontener.

Pisząc testy, często trzeba sprawdzać, czy wartości spełniają określone warunki. Wykorzystuje się do tego dopasowania (ang. matchers). *Jest* używa dopasowań, aby

umożliwić testowanie wartości na różne sposoby. Jednym z takich dopasowań jest *.toBe(),* które sprawdza, czy sprawdzany element posiada oczekiwaną wartość.

Innym przykładem dopasowania może być *.toMatch(reqex | string)*. Służy ono do sprawdzenia, czy dany ciąg pasuje do wyrażenia regularnego. Wiele innych doposowań wraz z opisem ich działania można znaleźć w oficjalnej dokumentacji *Jest*.

Inna metoda *getByText()* spowoduje wyszukanie wszystkich elementów, które zawierają określony węzeł tekstowy. Na przykład: *(/przykład/i)* jest wyrażeniem regularnym, które zwraca pierwszy węzeł zawierający co najmniej tekst „przykład”.

Metoda *fireEvent()* służy do uruchamiania zdarzeń DOM. W tym przypadku wyszukiwany jest odpowiedni przycisk - przy pomocy metody *getByText()* i następnie wywoływane jest jego kliknięcie.

Zintegrowany Program Rozwoju Politechniki Lubelskiej - część druga

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego