AKADEMIA TARNOWSKA

Wydział Nauk Technicznych

Kierunek: ***Informatyka***

2025/2026

Szymon Masłoń

PRACA INŻYNIERSKA

Projekt i implementacja aplikacji webowej do zarządzania osobistymi kolekcjami multimediów

Promotor pracy:

mgr inż. Tomasz Gądek

Tarnów, *2026*

Spis treści

1. Wstęp 5
1.1 Motywacja 5
1.2 Cel Pracy 5
1.3 Zakres Pracy 6
2. Analiza biznesowa 7
2.1 Wymagania funkcjonalne 7
2.2 Wymagania niefunkcjonalne 8
2.2.1 Bezpieczeństwo 8
2.2.2 Wydajność 8
2.2.3 Czytelny interfejs użytkownika 8
2.3 Analiza rynku 9
3. Stos technologiczny 11
3.1 JavaScript 11
3.2 Node.js 11
3.3 Express 12
3.4 JWT 12
3.5 bcrypt 13
3.6 React 13
3.7 Tailwind CSS 14
3.8 REST API 14
3.9 JSON 14
3.10 PostgreSQL 15
3.11 Prisma 15
4. Implementacja systemu 17
4.1 Zakres funkcji 17
4.1.1 Gość (użytkownik niezalogowany) 17
4.1.2 Użytkownik zalogowany 17
4.2 Diagram przypadków użycia 18
4.3 Diagram ERD 20
4.4 Rejestracja użytkowników 21
4.5 Mechanizm weryfikacji tokenów JWT 23
4.6 Wyszukiwanie multimediów z API 24
4.7 Mechanizm głosowania na tagi 25
4.8 System oceniania multimediów 26
4.9 Organizacja kolekcji użytkownika 26
4.10 Integracja z usługą tłumaczeniową DeepL 26
5. Interfejsy użytkownika 27
5.1 Strona domowa 27
5.2 Rejestracja i logowanie 28
5.3 Wyszukiwanie 30
5.4 Zarządzanie kolekcją 32

5.5 Edycja multimediów 33
5.6 Profil użytkownika 34
5.7 Ustawienia konta 35
6. Testy 37
6.1 Testy jednostkowe 38
6.1.1 Test obsługi książek bez okładki 38
6.1.2 Test rejestracji użytkownika z istniejącym adresem e-mail 40
6.2 Testy integracyjne 41
6.2.1 Test integracji z API TMDB 41
6.2.2 Test wyszukiwania książek z Open Library 42
6.2.3 Test dodawania kolekcji 44
7. Podsumowanie i wnioski 47
Bibliografia 49
Spis rysunków 51
Spis listingów 53

1. Wstęp

Rozwój technologii cyfrowych istotnie zmienił sposób konsumpcji oraz organizowania treści rozrywkowych. Rosnąca liczba dostępnych filmów, gier i książek sprawia, że użytkownicy coraz częściej poszukują narzędzi, które pozwalają nie tylko na ich odkrywanie, ale również na świadome porządkowanie i zarządzanie zgromadzonymi materiałami.

1.1 Motywacja

Współczesny rynek rozrywki cyfrowej charakteryzuje się nadmiarem dostępnych treści, co prowadzi do zjawiska przeciążenia informacyjnego. Użytkownicy mają dostęp do tysięcy filmów, gier wideo oraz książek, a znaczna część czasu spędzanego na platformach rozrywkowych poświęcana jest na samo wyszukiwanie, a nie konsumpcję treści. Problem ten pogłębia fragmentacja rynku - poszczególne typy mediów funkcjonują w obrębie odrębnych, niezintegrowanych usług.

Istniejące rozwiązania, takie jak platformy streamingowe czy portale recenzenckie, zazwyczaj koncentrują się na jednej kategorii multimediów. W efekcie użytkownik traci możliwość całościowego spojrzenia na swoje zainteresowania, mimo że preferencje gatunkowe i tematyczne często przenikają różne formy treści. Brak jednego, spójnego systemu utrudnia efektywne zarządzanie własnymi zbiorami oraz wykorzystanie tych zależności w procesie odkrywania nowych materiałów.

Dodatkowym wyzwaniem pozostaje ograniczona możliwość uporządkowanego dzielenia się rekomendacjami. Choć platformy społecznościowe umożliwiają wymianę opinii, nie oferują narzędzi pozwalających na prezentowanie spójnych kolekcji obejmujących różne typy mediów.

1.2 Cel Pracy

Celem pracy jest zaprojektowanie i implementacja aplikacji webowej do zarządzania osobistymi kolekcjami multimediów, obejmującymi filmy, gry wideo oraz książki, w ramach jednego, spójnego systemu. Aplikacja umożliwia katalogowanie, ocenianie i organizowanie treści z wykorzystaniem mechanizmów wyszukiwania, filtrowania i tagowania.

Projekt zakłada integrację z zewnętrznymi interfejsami API w celu automatycznego pozyskiwania aktualnych metadanych. System wspiera także tworzenie i udostępnianie publicznych kolekcji, ułatwiając wymianę rekomendacji oraz odkrywanie nowych treści.

1.3 Zakres Pracy

Zakres pracy stanowi analiza, projekt oraz realizacja aplikacji webowej do zarządzania osobistymi kolekcjami multimediów. Praca składa się z następujących etapów:

- **Analiza biznesowa -** przeprowadzono analizę problemu nadmiaru i fragmentacji treści multimedialnych oraz zidentyfikowano potrzeby użytkowników w zakresie katalogowania, organizowania i udostępniania kolekcji filmów, gier wideo i książek. Na tej podstawie określono wymagania funkcjonalne i niefunkcjonalne systemu oraz zdefiniowano główne przypadki użycia aplikacji.
- **Dobór stosu technologicznego -** dokonano analizy dostępnych technologii frontendowych i backendowych, a następnie uzasadniono wybór narzędzi i frameworków wykorzystanych w projekcie, w tym technologii do budowy interfejsu użytkownika, warstwy serwerowej oraz bazy danych.
- **Implementacja systemu -** zrealizowano aplikację webową zgodnie z przyjętymi założeniami projektowymi, obejmującą warstwę kliencką, serwerową oraz bazę danych. System umożliwia zarządzanie kolekcjami multimediów, integrację z zewnętrznymi interfejsami API oraz obsługę mechanizmów uwierzytelniania użytkowników.
- **Projekt i realizacja interfejsów użytkownika -** opracowano responsywne i intuicyjne interfejsy użytkownika umożliwiające przeglądanie, wyszukiwanie, filtrowanie oraz organizowanie treści multimedialnych, a także tworzenie i udostępnianie kolekcji.
- **Testy systemu -** przeprowadzono testy funkcjonalne kluczowych elementów aplikacji w celu weryfikacji poprawności działania systemu oraz zgodności z określonymi wymaganiami funkcjonalnymi i niefunkcjonalnymi.

2. Analiza biznesowa

W tym rozdziale zaprezentowane zostaną podstawowe wymagania dotyczące aplikacji do zarządzania kolekcjami multimediów. Rozróżnienie wymagań funkcjonalnych oraz niefunkcjonalnych umożliwi precyzyjne określenie możliwości systemu, a także czynników mających wpływ na jego wydajność i zabezpieczenia.

2.1 Wymagania funkcjonalne

Aplikacja ma na celu umożliwienie użytkownikom pełnego zarządzania ich kolekcjami multimedialnymi. W ramach tego użytkownik będzie mógł:

1. Założyć konto użytkownika oraz uwierzytelnić się w systemie z wykorzystaniem danych w postaci adresu e-mail i hasła, co zapewnia bezpieczny dostęp do zgromadzonych materiałów.
2. Dodawać spersonalizowane listy, za pomocą których będzie w stanie organizować wybrane elementy zgodnie z własnymi upodobaniami.
3. Dodawać elementy multimedialne do swojego zbioru poprzez funkcję wyszukiwania. Aplikacja powinna w sposób automatyczny pobierać kluczowe dane o danym tytule, włączając w to nazwę, datę premiery, grafikę oraz opis.
4. Dodawać materiały ręcznie w przypadku, gdy nie znajdują się one w zewnętrznych źródłach.
5. Oceniać poszczególne tytuły w oparciu o ustaloną skalę wartości, z opcją dołączenia komentarzy.
6. Przeszukiwać zbiory na podstawie zróżnicowanych parametrów, takich jak nazwa, kategoria medium, ocena, tagi lub status pozycji.
7. Mieć dostęp do rozszerzonych kart informacyjnych każdego elementu, zawierających zarówno dane automatycznie zaimportowane, jak i ręcznie wprowadzone uzupełnienia.
8. Usuwać wybrane tytuły z kolekcji oraz kasować całe zbiory, gdy przestają być potrzebne.

2.2 Wymagania niefunkcjonalne

Wymagania niefunkcjonalne określają normy jakościowe platformy, które mają wpływ na jej zabezpieczenia, sprawność działania oraz wygodę użytkowania.

2.2.1 Bezpieczeństwo

Zabezpieczenie informacji użytkowników stanowi kluczowy element w projektowaniu systemu. Platforma powinna wykorzystywać mechanizm autoryzacji bazujący na tokenach JWT (ang. *JSON Web Token*). Rozwiązanie to eliminuje potrzebę przechowywania sesji po stronie serwera przy kolejnych operacjach, zachowując jednocześnie odpowiedni poziom bezpieczeństwa sesji.

2.2.2 Wydajność

Aplikacja powinna zapewniać płynne działanie niezależnie od rozmiaru gromadzonych zbiorów. Zapytania skierowane do zewnętrznych źródeł oraz procesy realizowane w lokalnej bazie danych powinny być zoptymalizowane w taki sposób, aby czas odpowiedzi systemu pozostawał na minimalnym poziomie. Efektywne zarządzanie zasobami oraz zoptymalizowana komunikacja z serwerem pozytywnie wpłyną na komfort użytkowania, zapewniając szybki dostęp do żądanych informacji.

2.2.3 Czytelny interfejs użytkownika

Interfejs systemu powinien być skonstruowany w taki sposób, aby umożliwiał szybką i bezproblemową nawigację, także osobom korzystającym z aplikacji po raz pierwszy. Formularze muszą charakteryzować się czytelnością i prostotą obsługi, a każdy etap wprowadzania informacji powinien być zrozumiały dla użytkownika. Intuicyjne rozmieszczenie elementów nawigacyjnych, precyzyjne etykiety oraz jasne podpowiedzi zagwarantują przyjazne doświadczenie podczas korzystania z platformy.

2.3 Analiza rynku

Rosnąca liczba serwisów oferujących dostęp do treści cyfrowych powoduje, że użytkownicy coraz częściej korzystają z wielu niezależnych platform jednocześnie. W praktyce prowadzi to do rozproszenia informacji o konsumowanych mediach oraz braku jednego narzędzia umożliwiającego spójne zarządzanie osobistymi zbiorami. Obserwowany trend wskazuje na rosnące zapotrzebowanie na aplikacje, które nie ograniczają się do jednej kategorii mediów, ale umożliwiają tworzenie zintegrowanych bibliotek osobistych.

Na rynku istnieje kilka popularnych rozwiązań o podobnej funkcjonalności. Wśród nich można wymienić serwisy takie jak IMDb w zakresie filmów i Goodreads w kategorii książek. Każda z tych platform skupia się jednak głównie na jednym typie treści, co ogranicza użytkownika do tworzenia odrębnych profili i list. Brakuje uniwersalnych rozwiązań, które pozwalałyby zarządzać wszystkimi rodzajami mediów w spójny sposób.

Analiza dostępnych narzędzi wskazuje również na istotne różnice w zakresie funkcjonalności i otwartości ekosystemu. Powstająca aplikacja wypełnia identyfikowaną lukę rynkową poprzez scalanie funkcjonalności dostępnych dotychczas wyłącznie w rozproszonych serwisach. Wykorzystanie integracji z uznanymi API zapewnia automatyzację procesu gromadzenia metadanych, jednocześnie umożliwiając użytkownikom elastyczne zarządzanie własnymi kolekcjami.

3. Stos technologiczny

3.1 JavaScript

JavaScript to wysokopoziomowy język programowania opracowany w 1995 roku, który stał się fundamentalnym narzędziem rozwoju aplikacji webowych. Język charakteryzuje się dynamicznym typowaniem, gdzie typ zmiennej określany jest w czasie działania programu na podstawie przypisanej wartości, co upraszcza proces implementacji, choć wymaga większej uwagi przy kontroli typów [8].

JavaScript obsługuje asynchroniczność poprzez callbacks, promises oraz async/await, co jest kluczowe dla operacji nieblokujących takich jak komunikacja z API. Mechanizm garbage collection automatycznie zarządza pamięcią, eliminując konieczność manualnej alokacji zasobów. Jego uniwersalność oraz bogaty ekosystem bibliotek sprawiają, że jest preferowanym wyborem przy tworzeniu aplikacji webowych.

3.2 Node.js

Node.js stanowi środowisko uruchomieniowe **JavaScript**, zaprojektowane do skalowalnych aplikacji webowych. Kluczową cechą **Node.js** jest architektura asynchroniczna oparta na zdarzeniach, pozwalająca na efektywną obsługę wielu równoczesnych połączeń [12]. Platforma umożliwia uruchamianie aplikacji na różnych systemach operacyjnych bez konieczności modyfikacji kodu. Integracja z menedżerem pakietów **npm** (ang. *Node Package Manager*) dostarcza rozległą bibliotekę modułów usprawniających implementację typowych funkcjonalności takich jak autoryzacja, walidacja danych czy obsługa baz danych.

Istotną zaletą **Node.js** jest możliwość wykorzystania **JavaScript** zarówno na backendzie, jak i frontendzie, co upraszcza proces deweloperski i przyspiesza tworzenie oprogramowania. **Node.js** znajduje zastosowanie w obsłudze żądań **HTTP** (ang. *Hypertext Transfer Protocol*), zarządzaniu bazami danych, przetwarzaniu plików oraz aplikacjach czasu rzeczywistego.

3.3 Express

Express to minimalistyczny framework webowy dla **Node.js**, dostarczający zestaw funkcjonalności do budowy aplikacji serwerowych i interfejsów API. Oferuje elastyczny system routingu, middleware oraz mechanizmy obsługi żądań **HTTP** i odpowiedzi. Framework nie narzuca sztywnej struktury projektowej, pozwalając programistom na swobodne kształtowanie architektury aplikacji.

Middleware w **Express** pełni kluczową rolę w przetwarzaniu żądań i odpowiedzi. Są to funkcje pośredniczące, które mogą modyfikować obiekt żądania (req), odpowiedzi (res) lub zakończyć cykl obsługi żądania [9]. Dzięki modułowej budowie middleware umożliwia łatwe dodawanie funkcji takich jak logowanie, autoryzacja, obsługa błędów czy parsowanie danych, co znacząco zwiększa elastyczność i czytelność kodu aplikacji.

3.4 JWT

JSON Web Token (JWT) jest otwartym standardem służącym do bezpiecznego przekazywania informacji między stronami w formacie JSON. Technologia ta znajduje zastosowanie głównie w systemach uwierzytelniania i autoryzacji użytkowników. W przeciwieństwie do tradycyjnych metod sesyjnych, **JWT** nie wymaga przechowywania danych o zalogowanych użytkownikach po stronie serwera - wszystkie niezbędne informacje zapisane są bezpośrednio w tokenie.

Każdy token składa się z trzech części: nagłówka (ang. *Header*), ładunku (ang. *Payload*) oraz podpisu (ang. *Signature*) [11]. Nagłówek definiuje algorytm szyfrowania i typ tokenu, ładunek zawiera dane takie jak identyfikator użytkownika czy czas wygaśnięcia, a podpis służy do weryfikacji integralności i autentyczności tokenu. Dzięki temu serwer może potwierdzić tożsamość użytkownika wyłącznie na podstawie przesłanego tokenu, bez potrzeby dodatkowych zapytań do bazy danych.

Token może być przechowywany po stronie klienta, np. w pamięci przeglądarki lub w ciasteczkach, a następnie przesyłany wraz z kolejnymi żądaniami **HTTP** w nagłówku. Dzięki temu użytkownik pozostaje zalogowany, dopóki token jest ważny.

3.5 bcrypt

Biblioteka **bcrypt** implementuje funkcję haszującą zaprojektowaną specjalnie do bezpiecznego przechowywania haseł. Jej działanie opiera się na koncepcji adaptacyjnego haszowania, w której poziom złożoności obliczeniowej może być regulowany za pomocą parametru kosztu. Parametr ten określa liczbę powtórzeń operacji haszowania, co pozwala dostosować odporność algorytmu na ataki brute-force do aktualnych możliwości sprzętowych.

Bezpieczeństwo haseł opiera się na dwóch technikach: "soleniu" (ang. *salt*) czyli dodaniu losowych danych uniemożliwiających użycie gotowych tabel haszów oraz "rozciąganiu klucza" (ang. *key stretching*) - wielokrotnym haszowaniu spowalniającym ataki metodą pełnego przeglądu [3]. Biblioteka bcrypt implementuje oba mechanizmy, automatycznie generując unikalną sól dla każdego hasła i zapewniając jednokierunkowe przekształcenie uniemożliwiające odzyskanie oryginału.

3.6 React

React to biblioteka **JavaScript** do budowy interfejsów użytkownika, opracowana i utrzymywana przez **Meta**. Implementuje komponentową architekturę, gdzie każdy element UI stanowi niezależną, reużywalną jednostkę z własnym stanem i logiką. Kluczowym mechanizmem **React** jest wirtualny DOM, który minimalizuje liczbę operacji na rzeczywistym drzewie DOM poprzez obliczanie i aplikowanie tylko niezbędnych zmian, co znacząco poprawia wydajność aplikacji.

React cechuje jednokierunkowy przepływ danych, gdzie stan aplikacji przekazywany jest z komponentów nadrzędnych do podrzędnych poprzez props. Do zarządzania stanem komponentów służą hooki, takie jak **useState** czy **useEffect**, umożliwiające obsługę dynamicznych danych i efektów ubocznych w komponentach funkcyjnych [1].

3.7 Tailwind CSS

Tailwind CSS to framework **CSS,** gdzie style tworzone są poprzez komponowanie małych, jednofunkcyjnych klas bezpośrednio w znacznikach HTML. Zamiast pisać niestandardowe style **CSS**, programista łączy gotowe klasy co przyspiesza proces stylowania.

Framework dostarcza system projektowy obejmujący spójną paletę kolorów, typografię, odstępy oraz responsywność. Narzędzie wspiera pełną personalizację poprzez plik konfiguracyjny, pozwalając dostosować każdy aspekt systemu projektowego do potrzeb aplikacji [14].

3.8 REST API

REST (ang. *Representational State Transfer*) **API** to styl architektoniczny definiujący zasady projektowania API w systemach rozproszonych. Interfejsy REST opierają się na bezstanowej komunikacji, gdzie każde żądanie zawiera wszystkie informacje niezbędne do jego obsługi, bez polegania na kontekście przechowywanym na serwerze [6].

REST wykorzystuje standardowe metody **HTTP** (GET, POST, PUT, DELETE) do wykonywania operacji na zasobach reprezentowanych przez unikalne adresy URL. Architektura promuje jednolitość interfejsu, oddzielenie klienta od serwera oraz możliwość cache'owania odpowiedzi, co wpływa na skalowalność i wydajność systemu.

3.9 JSON

JSON (JavaScript Object Notation) to lekki format wymiany danych oparty na podzbiorze składni **JavaScript**. Charakteryzuje się czytelnością dla człowieka oraz prostotą parsowania przez maszyny. Struktura **JSON** opiera się na dwóch typach danych: obiektach (kolekcjach par klucz-wartość) oraz tablicach (uporządkowanych listach wartości).

Format wspiera podstawowe typy danych takie jak stringi, liczby, wartości logiczne, null oraz zagnieżdżone struktury obiektów i tablic. **JSON** stał się standardem w komunikacji API ze względu na niezależność od języka programowania, mniejszy rozmiar w porównaniu do XML oraz natywne wsparcie w JavaScript [10].

3.10 PostgreSQL

PostgreSQL to obiektowo-relacyjny system zarządzania bazą danych o otwartym kodzie źródłowym, charakteryzujący się wysoką zgodnością ze standardem SQL oraz rozbudowanymi mechanizmami zapewniającymi integralność danych [7]. System wspiera zaawansowane typy danych, w tym **JSON**, umożliwiając elastyczne przechowywanie zarówno strukturyzowanych, jak i częściowo ustrukturyzowanych danych. **PostgreSQL** znajduje zastosowanie w aplikacjach wymagających niezawodności, skalowalności oraz zaawansowanych możliwości zapytań relacyjnych.

3.11 Prisma

Prisma jest narzędziem typu **ORM** (ang. *Object-Relational Mapping*), wykorzystywanym do komunikacji aplikacji z relacyjną bazą danych **PostgreSQL**. Umożliwia definiowanie struktury bazy danych w postaci schematu oraz generowanie silnie typowanego klienta, który upraszcza wykonywanie operacji CRUD (ang. *Create, Read, Update, Delete*) z poziomu kodu aplikacji.

Zastosowanie narzędzia Prisma pozwala na ograniczenie liczby zapytań SQL tworzonych ręcznie, co zmniejsza ryzyko błędów oraz poprawia czytelność i utrzymywalność kodu backendu. Dodatkową zaletą jest automatyczna walidacja typów danych oraz wsparcie dla migracji schematu bazy danych, co ułatwia rozwój i ewolucję struktury danych w trakcie trwania projektu.

4. Implementacja systemu

Niniejszy rozdział opisuje proces realizacji aplikacji webowej służącej do zarządzania osobistymi kolekcjami multimediów. Przedstawiono w nim szczegóły techniczne wybranego środowiska programistycznego, strukturę architektury systemu oraz podział na poszczególne warstwy logiczne. Głównym celem prac implementacyjnych było zaprojektowanie wydajnego rozwiązania, które integruje dane z wielu źródeł zewnętrznych, zapewniając jednocześnie spójność i responsywność interfejsu użytkownika.

4.1 Zakres funkcji

W ramach projektu wyodrębniono dwa poziomy dostępu do systemu zależne od statusu uwierzytelnienia. Podział ten determinuje zakres dostępnych funkcjonalności oraz możliwe interakcje z danymi.

4.1.1 Gość (użytkownik niezalogowany)

Jest to domyślny poziom dostępu dla każdej osoby odwiedzającej witrynę, która nie przeszła procesu autoryzacji.

- **Dostępne funkcje:**
  - Wyświetlanie strony powitalnej z ogólnymi informacjami o systemie.
  - Rejestracja nowego konta w celu uzyskania pełnego dostępu.
  - Logowanie do istniejącego profilu.
  - Przeglądanie publicznych kolekcji

4.1.2 Użytkownik zalogowany

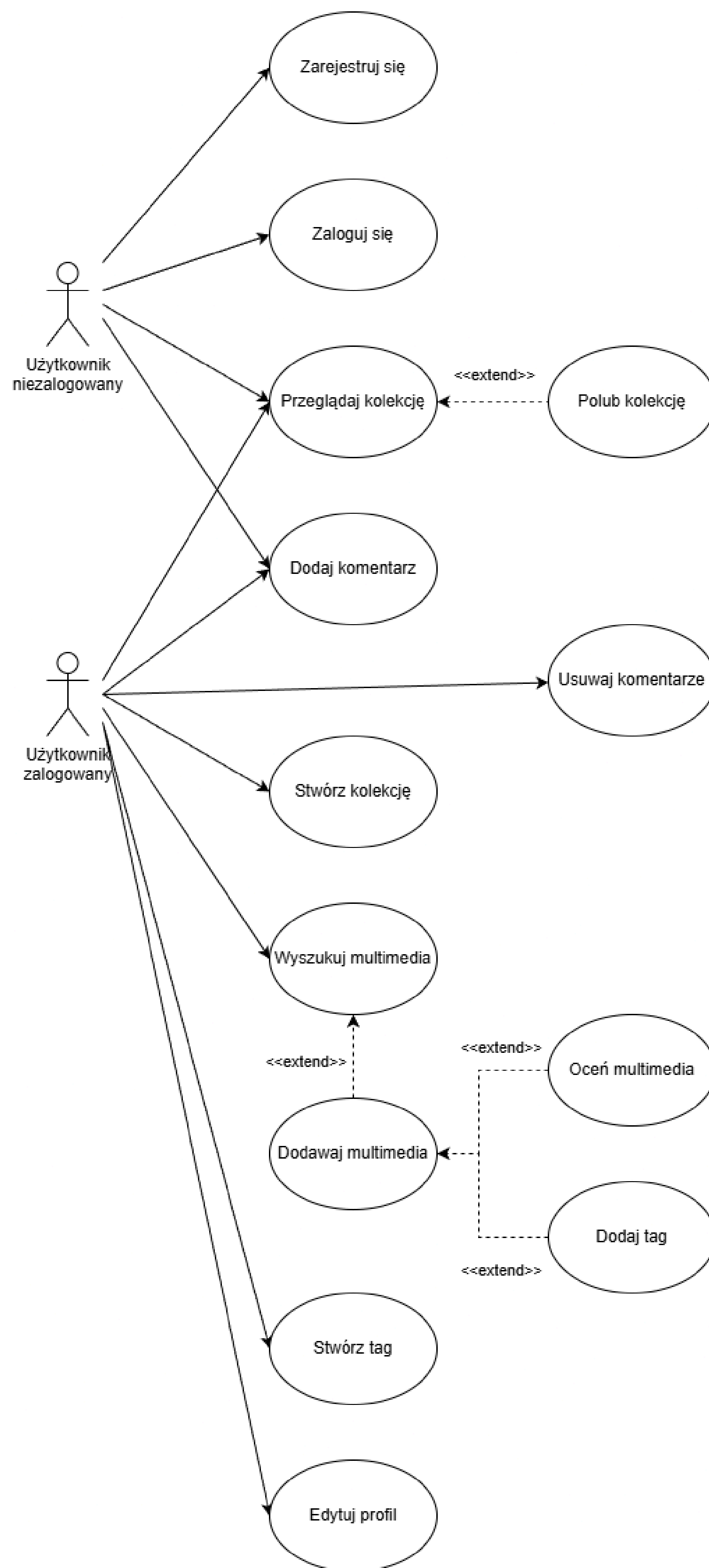
Rola ta jest przypisywana osobom, które poprawnie przeszły proces uwierzytelniania. Użytkownik ten jest głównym aktorem w systemie i posiada pełną kontrolę nad własnymi zbiorami danych.

- **Dostępne funkcje:**
  - **Agregacja danych:** Wyszukiwanie filmów, gier i książek przy użyciu zintegrowanych zewnętrznych źródeł.
  - **Zarządzanie kolekcjami:** Tworzenie nowych list, upublicznianie ich, dodawanie wyszukanych pozycji do katalogu i ich usuwanie.
  - **Organizacja:** Przypisywanie własnych tagów, przeszukiwanie zasobów.

- **Ocena mediów:** Wystawianie ocen i recenzji pozycjom znajdującym się w bibliotece.
- **Personalizacja:** Edycja ustawień konta.

4.2 Diagram przypadków użycia

Prezentowany diagram przypadków użycia, wykonany w standardzie UML, obrazuje zakres funkcjonalny aplikacji oraz interakcje zachodzące pomiędzy zdefiniowanymi aktorami a systemem. W projekcie wyodrębniono dwie kluczowe role: Gościa oraz Użytkownika zalogowanego. Podstawowy poziom dostępu pozwala Gościowi na realizację procesów związanych z uwierzytelnianiem, takich jak rejestracja nowego konta czy logowanie. Główny aktor, Użytkownik zalogowany, inicjuje kluczowe procesy, w tym tworzenie kolekcji, przeszukiwanie multimediów oraz kategoryzację zasobów poprzez tagowanie i ocenianie.



Rys. 4.1. Diagram przypadków użycia [opracowanie własne]

4.3 Diagram ERD

W celu wizualizacji struktury danych zaprojektowano diagram ERD (ang. *Entity-Relationship Diagram*), który ilustruje powiązania między poszczególnymi tabelami w bazie **PostgreSQL** (rysunek 4.2). W projektowanej aplikacji do zarządzania kolekcjami multimediów zaprojektowano model relacyjny w **PostgreSQL**, który obejmuje tabele użytkowników, kolekcji oraz pozycji multimedialnych. Każda tabela zawiera atrybuty niezbędne do katalogowania multimediów, a także mechanizmy z nimi związane.

Rys. 4.2. Diagram ERD głównych encji systemu [opracowanie własne]

4.4 Rejestracja użytkowników

Moduł rejestracji odpowiada za tworzenie nowych kont użytkowników oraz inicjalizację podstawowych zasobów wymaganych do korzystania z systemu. Implementacja obejmuje walidację danych wejściowych, weryfikację unikalności adresu e-mail, bezpieczne przechowywanie haseł oraz automatyczne dodanie głównej kolekcji użytkownika.

Proces rozpoczyna się od obsługi żądania **HTTP POST** skierowanego na endpoint /api/auth/register. Funkcja **signup()** pobiera z ciała żądania nazwę użytkownika, adres e-mail, hasło oraz nazwę wyświetlaną. Następnie wykonywana jest walidacja wstępna z wykorzystaniem funkcji validateSignupData(), która sprawdza kompletność danych, poprawność formatu adresu e-mail oraz spełnienie wymagań dotyczących minimalnej długości hasła.

Po pomyślnej walidacji system wykonuje zapytanie do bazy danych za pomocą ORM Prisma w celu sprawdzenia, czy konto z podanym adresem e-mail już istnieje. W przypadku wykrycia konfliktu zwracana jest odpowiedź **HTTP 409**, co zapobiega tworzeniu zduplikowanych kont użytkowników.

Hasło użytkownika jest zabezpieczane przy użyciu biblioteki **bcrypt** z parametrem kosztu ustawionym na wartość 10. Algorytm generuje unikalną sól dla każdego hasła i zapisuje w bazie danych wyłącznie jego skrót, co skutecznie chroni dane przed atakami typu brute-force oraz wykorzystaniem tablic tęczowych [13].

Po założeniu konta system automatycznie dodaje główną kolekcję multimediów, pełniącą rolę centralnego zbioru użytkownika. Rozwiązanie to upraszcza pierwsze użycie aplikacji i zapewnia spójną strukturę danych od momentu rejestracji.

Końcowym etapem procesu jest wygenerowanie tokenów **JWT**, które umożliwiają natychmiastowe uwierzytelnienie użytkownika po zakończeniu rejestracji. Serwer zwraca odpowiedź **HTTP 201** zawierającą podstawowe informacje o nowo założonym koncie, z pominięciem danych wrażliwych.

Listing 4.1. Fragment funkcji rejestracji użytkownika [opracowanie własne]

```
export const signup = async (req, res) => {
    const { name, email, username, password } = req.body;

    const validation = validateSignupData({ name, email, username,
password });
    if (!validation.valid) {
        return res.status(validation.status).json({ message:
validation.message });
    }

    const existingUser = await prisma.user.findUnique({
        where: { email }
    });
    if (existingUser) {
        return res.status(409).json({
            message: 'User with this email already exists'
        });
    }

    const hashedPassword = await bcrypt.hash(password, 10);
    const newUser = await prisma.user.create({
        data: {
            name,
            email,
            username,
            password: hashedPassword
        }
    });

    await prisma.collection.create({
        data: {
            name: "Moje media",
            isMainCollection: true,
            isPublic: true,
            userId: newUser.id
        }
    });

    setTokens(newUser.id, res);
    const { password: _, ...userWithoutPassword } = newUser;
    res.status(201).json({ user: userWithoutPassword });
};
```

4.5 Mechanizm weryfikacji tokenów JWT

Autoryzacja dostępu do chronionych zasobów systemu realizowana jest z wykorzystaniem middleware authenticate, odpowiedzialnego za weryfikację tokenu dostępowego **JWT** oraz identyfikację użytkownika wykonującego żądanie. W prezentowanym rozwiązaniu token przechowywany jest w plikach cookie po stronie klienta i przesyłany automatycznie wraz z każdym żądaniem **HTTP**.

Proces uwierzytelniania rozpoczyna się od odczytu tokenu accessToken z obiektu req.cookies. Jego brak skutkuje przerwaniem dalszego przetwarzania żądania i zwróceniem odpowiedzi **HTTP 401**. Następnie wykonywana jest kryptograficzna weryfikacja tokenu z użyciem funkcji verifyAccessToken(), która sprawdza poprawność podpisu oraz termin ważności. Po pomyślnej weryfikacji uzyskiwany jest zdekodowany ładunek zawierający identyfikator użytkownika zapisany w deklaracjach (ang. claims).

Na podstawie tych danych system pobiera aktualny rekord użytkownika z bazy danych. Dodatkowa weryfikacja istnienia konta zabezpiecza aplikację przed użyciem tokenów powiązanych z usuniętymi lub nieaktywnymi użytkownikami. Po poprawnym odnalezieniu rekordu dane użytkownika, z pominięciem hasła, dołączane są do obiektu req.user, co umożliwia ich dalsze wykorzystanie w warstwie kontrolerów.

Obsługa błędów obejmuje przypadki tokenów wygasłych, nieprawidłowych lub zmodyfikowanych i skutkuje zwróceniem odpowiedzi **HTTP 401**. Zastosowanie tokenów **JWT** w połączeniu z mechanizmem cookie umożliwia bezstanowe przetwarzanie żądań, zwiększa bezpieczeństwo sesji oraz ułatwia skalowanie aplikacji [4].

Listing 4.2. Fragment funkcji weryfikacji tokenu [opracowanie własne]

```
export const authenticate = async (req, res, next) => {
  try {
    const accessToken = req.cookies.accessToken;

    if (!accessToken) {
      return res.status(401).json({
        message: 'Access token required'
      });
    }

    const decoded = verifyAccessToken(accessToken);

    const user = await prisma.user.findUnique({
      where: { id: decoded.userId }
    });

    if (!user) {
      return res.status(401).json({
        message: 'User not found'
      });
    }

    const { password: _, ...userWithoutPassword } = user;
    req.user = userWithoutPassword;

    next();
  } catch (err) {
    console.error('Authentication error:', err);
    return res.status(401).json({
      message: 'Invalid or expired access token'
    });
  }
};
```

4.6 Wyszukiwanie multimediów z API

System wyszukiwania multimediów umożliwia przeszukiwanie trzech typów zasobów: filmów, książek oraz gier wideo. Na podstawie wybranego rodzaju medium zapytanie kierowane jest do odpowiedniego zewnętrznego źródła danych zintegrowanego z aplikacją.

Proces rozpoczyna się od wprowadzenia przez użytkownika frazy tekstowej, po czym system, w zależności od typu wyszukiwanego medium, kieruje zapytanie do odpowiedniego zewnętrznego interfejsu **API**: TMDB (ang. *The Movie Database*), będącego bazą danych filmów i seriali, Open Library, pełniącego rolę otwartego katalogu książek, lub RAWG, dostarczającego informacje o grach wideo. Otrzymane dane są następnie przetwarzane i ujednolicone w celu ich prezentacji w spójnej formie, niezależnie od źródła pochodzenia.

Liczba wyników jest ograniczana w celu zachowania czytelności interfejsu oraz zapewnienia płynności działania aplikacji. Każdy wynik wyszukiwania zawiera podstawowe informacje obejmujące tytuł, rok wydania, obraz okładki oraz dane twórcy medium, rozumiane jako autor, reżyser lub deweloper, w zależności od jego typu.

Szczegółowe informacje pobierane są dopiero w momencie dodania wybranej pozycji do kolekcji użytkownika. Takie rozwiązanie pozwala skrócić czas odpowiedzi podczas wyszukiwania oraz ograniczyć liczbę zapytań kierowanych do zewnętrznych serwisów.

4.7 Mechanizm głosowania na tagi

System tagowania został rozszerzony o mechanizm głosowania, którego celem jest poprawa jakości opisu multimediów. Użytkownicy mogą oddawać głosy na istniejące tagi, a ich znaczenie jest określane na podstawie aktywności społeczności.

Każdy tag przypisany do medium posiada licznik głosów, który wpływa na jego widoczność oraz kolejność prezentacji. System zapewnia, że jeden użytkownik może oddać tylko jeden głos na dany tag w obrębie konkretnego medium, co zapobiega wielokrotnemu naliczaniu głosów.

Operacje związane z dodawaniem i usuwaniem głosów są realizowane w sposób zapewniający spójność danych. Dodanie głosu oraz aktualizacja licznika wykonywane są w ramach jednej operacji zapisu do bazy danych, co ogranicza ryzyko niespójności danych przy równoczesnych żądaniach.

W sytuacji cofnięcia ostatniego głosu dany tag zostaje automatycznie usunięty z medium. Rozwiązanie to upraszcza zarządzanie danymi oraz zapobiega przechowywaniu nieaktualnych oznaczeń.

4.8 System oceniania multimediów

System oceniania został zaimplementowany jako pojedynczy punkt końcowy **API** obsługujący zarówno dodawanie nowej oceny, jak i jej modyfikację. Rozwiązanie to upraszcza komunikację z serwerem po stronie klienta.

Przed zapisaniem danych wykonywana jest walidacja wartości oceny, która musi mieścić się w zakresie od 1 do 5 oraz być liczbą całkowitą. System weryfikuje również istnienie ocenianego medium, co zapobiega tworzeniu nieprawidłowych powiązań w bazie danych. Każda ocena jest jednoznacznie identyfikowana przez parę użytkownik-medium, co gwarantuje możliwość posiadania maksymalnie jednej oceny danego medium przez użytkownika.

Po zapisaniu lub aktualizacji oceny system oblicza średnią oraz liczbę ocen, które są zwracane w odpowiedzi **API** i wykorzystywane w interfejsie użytkownika. Takie podejście zapewnia spójność danych oraz prostą obsługę mechanizmu oceniania.

4.9 Organizacja kolekcji użytkownika

System kolekcji zakłada istnienie jednej głównej kolekcji użytkownika, pełniącej rolę centralnego zbioru wszystkich multimediów. Pozostałe kolekcje mają charakter podkolekcji i służą do tematycznego grupowania zasobów.

Dodanie medium do podkolekcji nie powoduje jego usunięcia z kolekcji głównej. Natomiast usunięcie zasobu z kolekcji głównej skutkuje automatycznym usunięciem go ze wszystkich powiązanych podkolekcji. Przyjęte rozwiązanie eliminuje konieczność duplikowania danych oraz upraszcza zarządzanie relacjami pomiędzy kolekcjami.

4.10 Integracja z usługą tłumaczeniową DeepL

Do tłumaczenia opisów multimediów pobieranych z zewnętrznych interfejsów **API** wykorzystano usługę DeepL. Mechanizm tłumaczenia jest uruchamiany na żądanie użytkownika za pomocą dedykowanego przycisku dostępnego w interfejsie aplikacji.

Po stronie klienta inicjowane jest żądanie do warstwy serwerowej, która następnie komunikuje się z interfejsem API usługi DeepL. Takie rozwiązanie pozwala na wykonywanie tłumaczeń wyłącznie w momencie faktycznej potrzeby, ograniczając liczbę zapytań do usługi zewnętrznej oraz koszty jej wykorzystania.

5. Interfejsy użytkownika

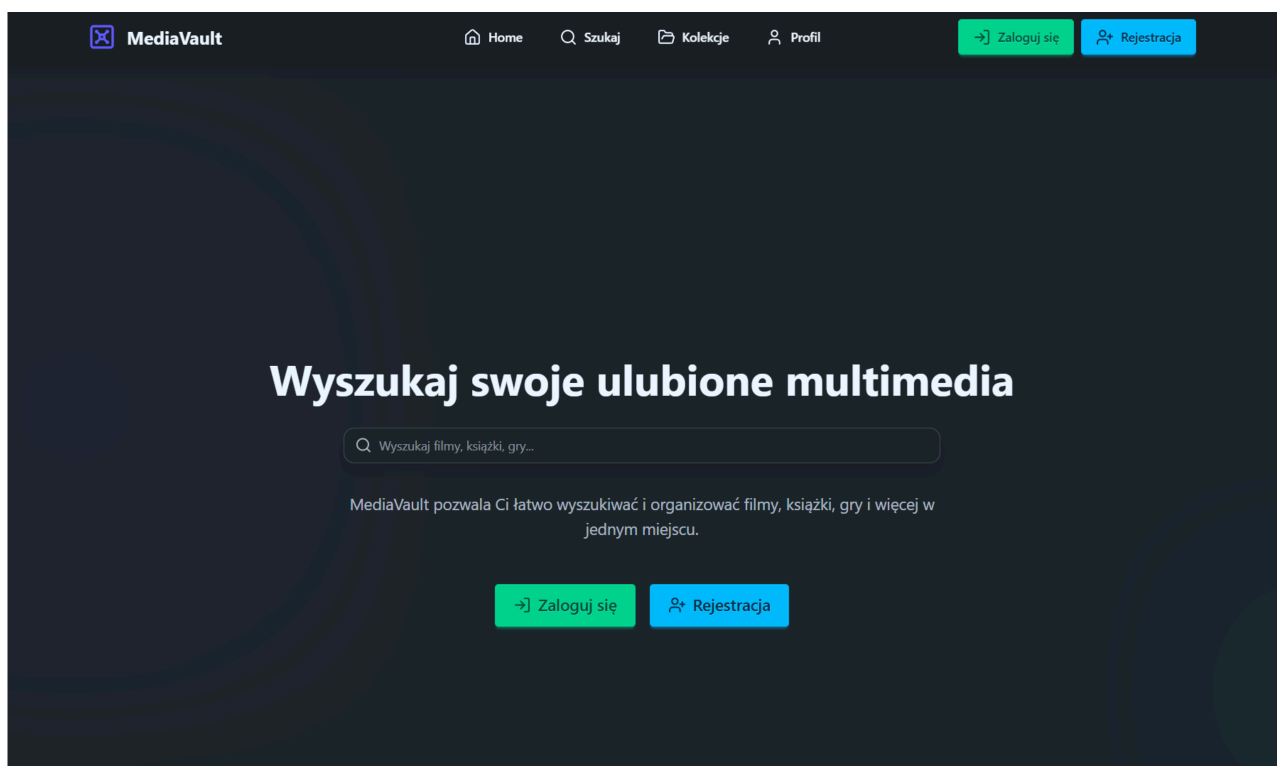
Interfejs użytkownika aplikacji został zaprojektowany w sposób zapewniający intuicyjną obsługę oraz przejrzysty dostęp do wszystkich funkcjonalności systemu. Projekt graficzny realizuje zasady responsywności i dostępności, uwzględniając różnorodne urządzenia końcowe oraz wymagania użytkowników o różnym poziomie doświadczenia.

5.1 Strona domowa

Strona domowa stanowi pierwszy punkt kontaktu użytkownika z aplikacją. Na środku interfejsu znajduje się pole wyszukiwania umożliwiające natychmiastowe rozpoczęcie eksploracji dostępnych multimediów. Poniżej umieszczono dwa przyciski: "Zaloguj się" oraz "Rejestracja", które kierują do odpowiednich formularzy uwierzytelniania. Dodatkowo, w dolnej części strony wyświetlany jest krótki opis funkcjonalności platformy.

Użytkownik niezalogowany otrzymuje dostęp wyłącznie do podstawowego wyszukiwania, podczas gdy pełny zakres możliwości aplikacji jak zakładanie kolekcji, dodawanie ocen i zarządzanie listami staje się dostępny po zarejestrowaniu konta.

W górnej części interfejsu umieszczono pasek nawigacyjny zawierający logo aplikacji oraz cztery główne sekcje: Home, Szukaj, Kolekcje i Profil. Po prawej stronie paska znajdują się przyciski "Zaloguj się" i "Rejestracja", umożliwiające szybki dostęp do funkcji uwierzytelniania z dowolnej podstrony. Nawigacja pozostaje widoczna na wszystkich stronach aplikacji, zachowując spójność interfejsu użytkownika i umożliwiając płynne przemieszczanie się pomiędzy poszczególnymi modułami systemu.



Rys. 5.1. Interfejs strony głównej [opracowanie własne]

5.2 Rejestracja i logowanie

Proces uwierzytelniania realizowany jest poprzez dwa dedykowane formularze: logowania (rysunek 5.1) oraz rejestracji (rysunek 5.2). Formularz logowania wymaga podania loginu oraz hasła, oferując dodatkowo możliwość przejścia do strony rejestracji za pomocą wyraźnie oznaczonego odnośnika. Interfejs modułu uwierzytelniania charakteryzuje się minimalistycznym układem, koncentrującym uwagę użytkownika na kluczowych polach formularza.

Rys. 5.2. Interfejs logowania [opracowanie własne]

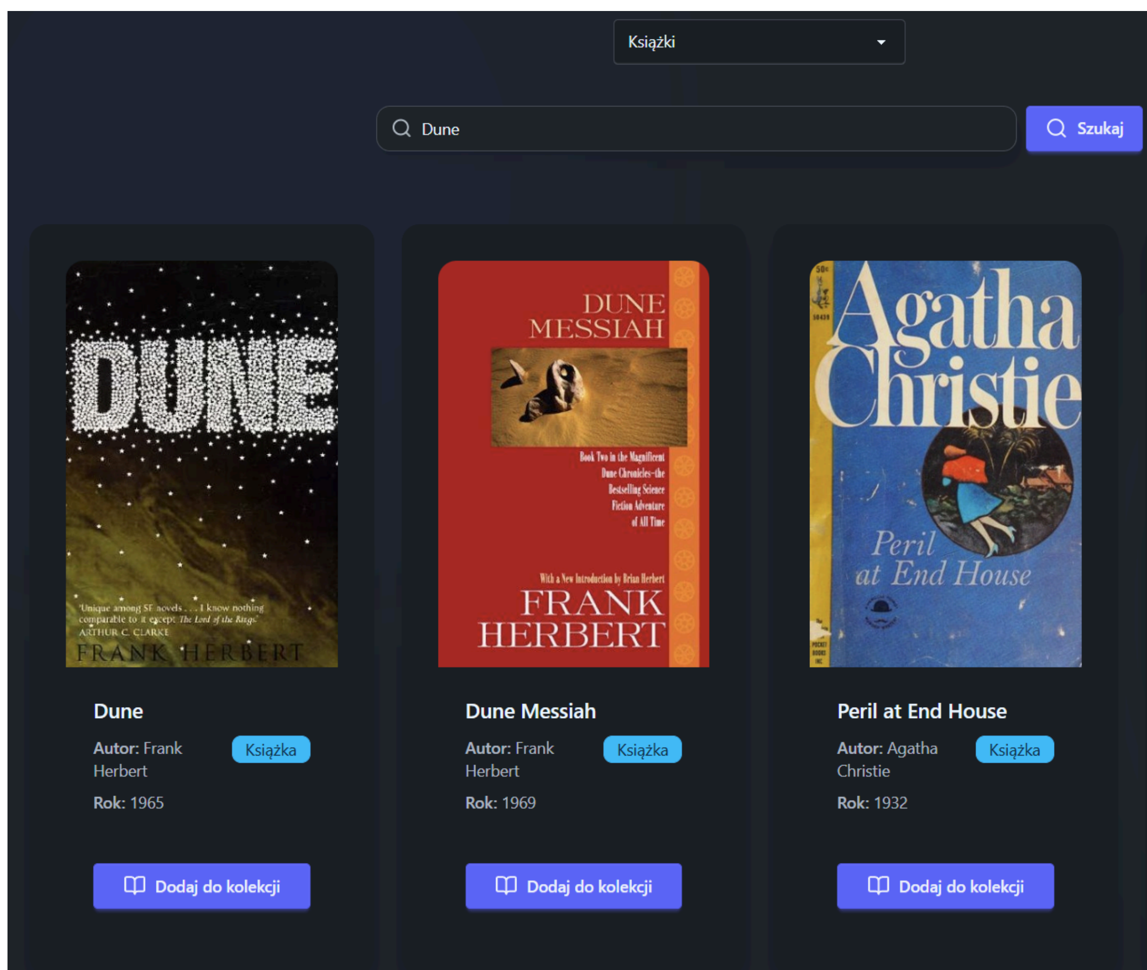
Formularz rejestracji (rysunek 5.2) obejmuje rozszerzony zestaw danych wymaganych do założenia konta: nazwę użytkownika, imię, nazwisko adres e-mail oraz hasło z polem potwierdzenia. Walidacja danych odbywa się zarówno po stronie klienta poprzez mechanizmy wbudowane w przeglądarkę, jak i po stronie serwera, co zapewnia integralność wprowadzanych informacji. Struktura formularza zapewnia logiczny przepływ wprowadzania danych, kierując uwagę użytkownika od danych identyfikacyjnych do danych uwierzytelniających.

Rys. 5.3. Interfejs rejestracji [opracowanie własne]

5.3 Wyszukiwanie

Moduł wyszukiwania stanowi centralny punkt dostępu do zewnętrznych baz danych multimediów. Interfejs umożliwia wybór źródła danych poprzez menu rozwijane mające opcje: książki, filmy oraz gry. Wybór źródła aktywuje odpowiednie API i dostosowuje wyświetlane rezultaty do specyfiki danego medium.

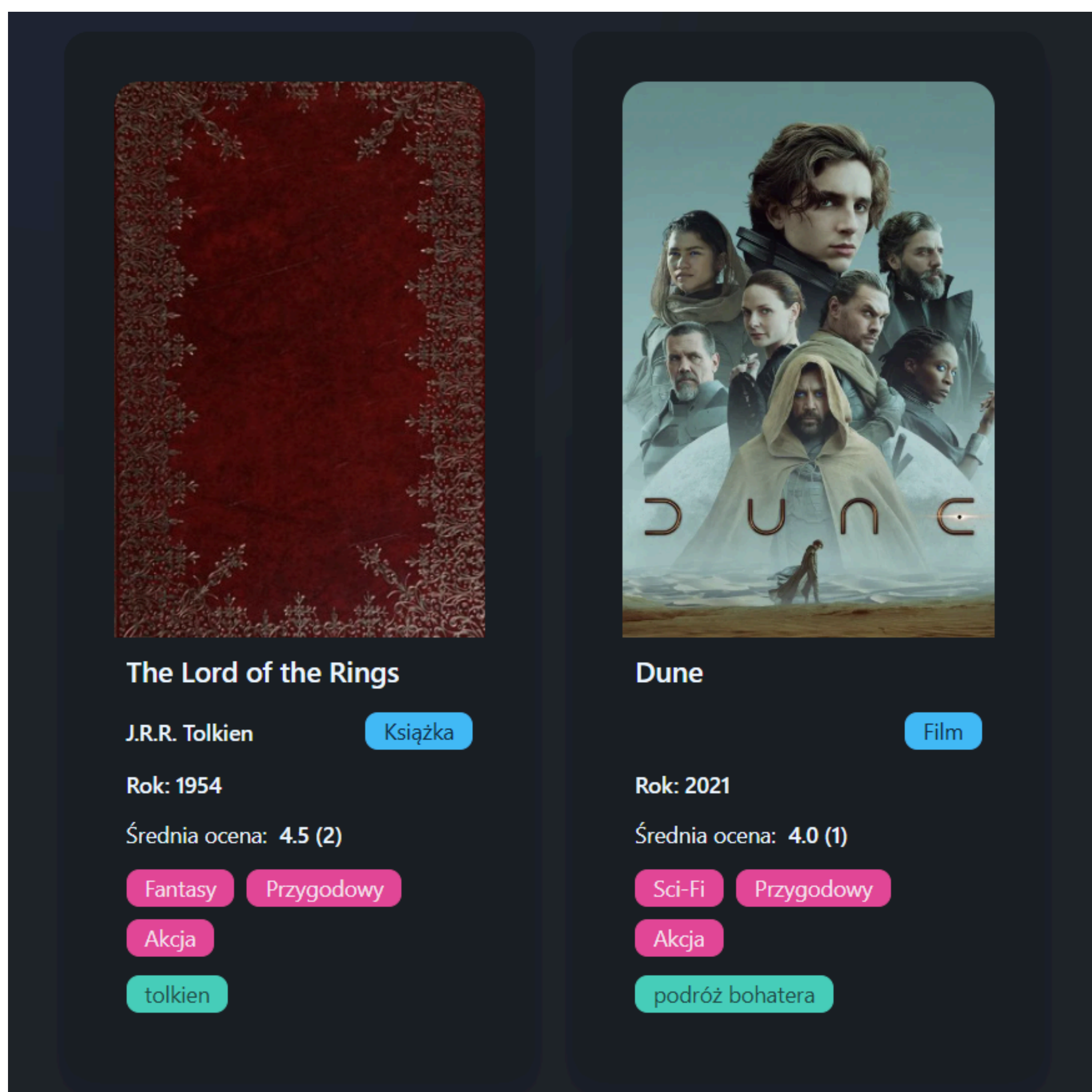
Wyniki wyszukiwania prezentowane są w układzie kafelkowym, zawierającym okładkę medium, tytuł, autora, rok wydania oraz przycisk dodania do kolekcji . Format ten umożliwia szybkie porównanie wyników i podejmowanie decyzji o dodaniu pozycji do kolekcji użytkownika. System dynamicznie pobiera dane z wybranego API, prezentując rezultaty w czasie rzeczywistym bez konieczności przeładowania strony.



Rys. 5.4. Interfejs wyszukiwania [opracowanie własne]

5.4 Zarządzanie kolekcją

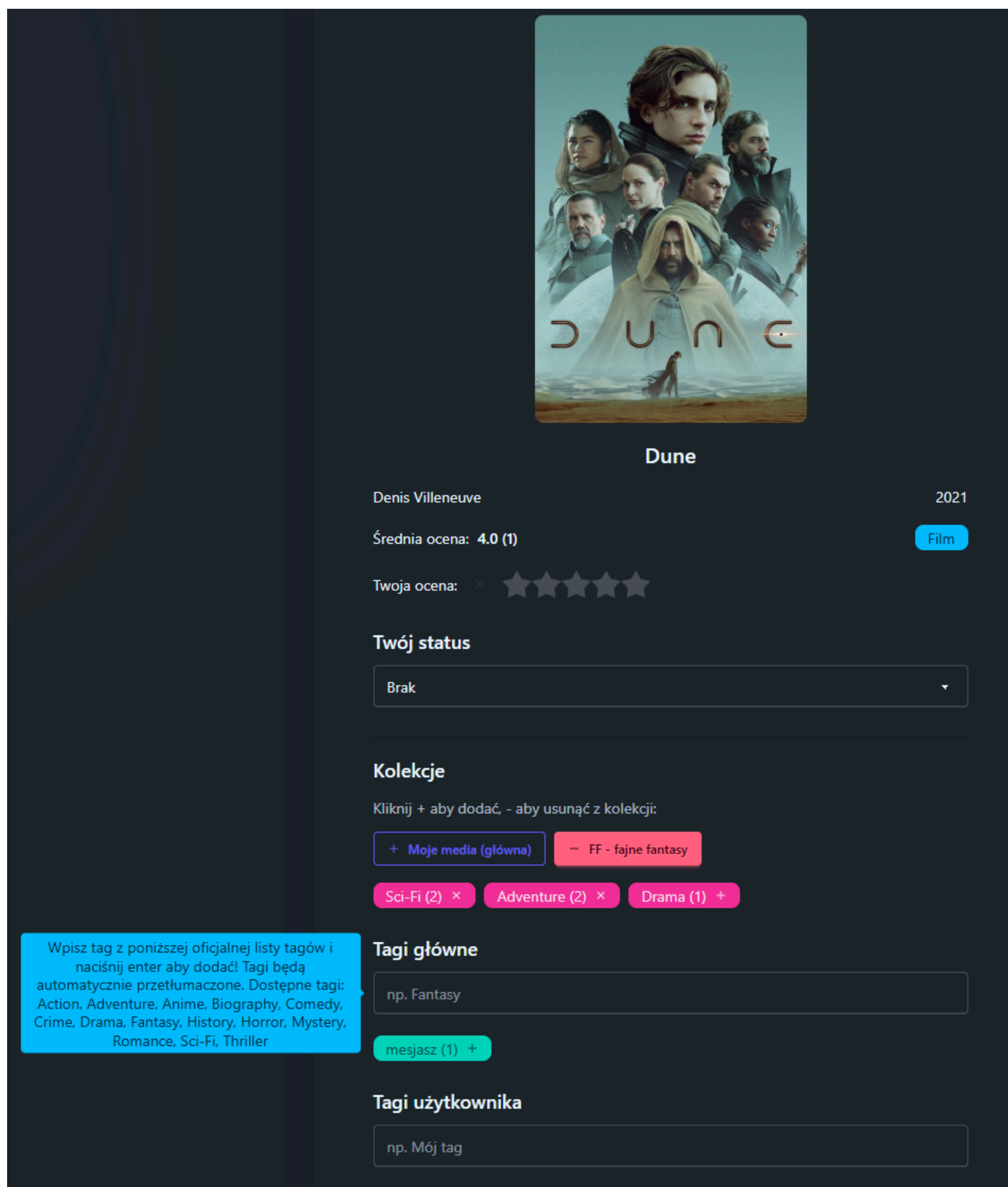
Widok kolekcji użytkownika prezentuje zgromadzone media w formie kart zawierających pełen zestaw metadanych. Każda karta obejmuje: okładkę medium, tytuł, opis, średnią z ocen wystawionych przez użytkowników, przypisane tagi główne, tagi stworzone przez użytkowników i typ multimediów. Można przejść do interfejsu edycji konkretnych mediów przez kliknięcie na nie.



Rys. 5.5. Interfejs własnej kolekcji [opracowanie własne]

5.5 Edycja multimediów

Formularz edycji medium umożliwia modyfikację wszystkich kluczowych atrybutów pozycji w kolekcji. Interfejs obejmuje pola: oceny w skali od 1-5 gwiazdek którą można zmieniać lub usuwać swoją ocenę, zmiana statusu, przynależność do kolekcji użytkownika, tagów głównych dostępnych na stronie i tagów użytkownika które można dowolnie wymyślać.

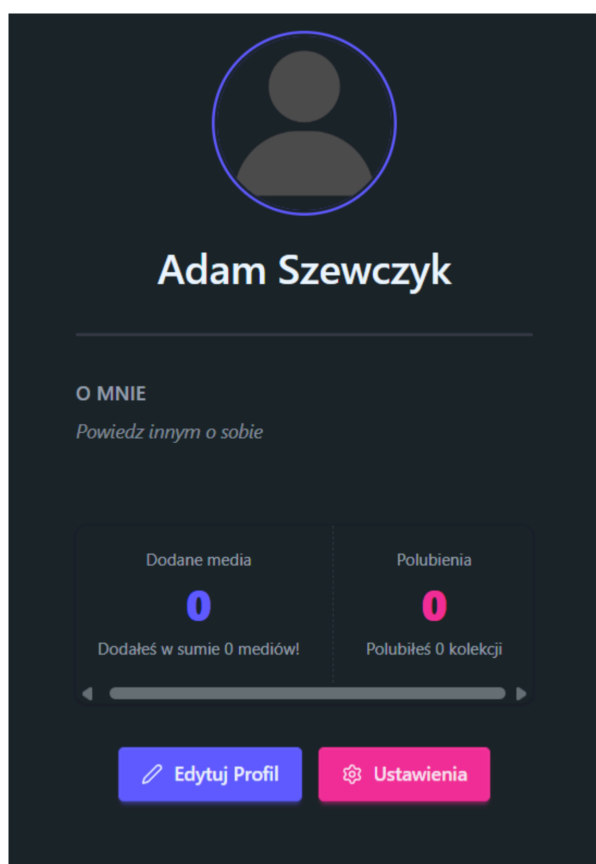


Rys. 5.6. Interfejs edycji multimediów [opracowanie własne]

5.6 Profil użytkownika

Strona profilu użytkownika prezentuje dane personalizacyjne konta oraz statystyki aktywności w aplikacji. W centralnej części widoku znajduje się awatar użytkownika oraz jego nazwa. Poniżej umieszczono sekcję "O MNIE" która umożliwia użytkownikowi dodanie krótkiego opisu biograficznego.

Interfejs wyświetla dwie metryki aktywności: liczbę dodanych mediów oraz liczbę polubionych kolekcji, prezentowanych w formie kolorowych ikon z wartościami numerycznymi. W dolnej części ekranu znajdują się dwa przyciski akcji: "Edytuj Profil" oraz "Ustawienia", które umożliwiają modyfikację danych osobistych i konfigurację parametrów konta. Struktura strony profilu została zaprojektowana zgodnie z zasadami projektowania interfejsów użytkownika, gdzie najważniejsze informacje umieszcza się w centrum uwagi, a funkcje zarządzania kontem pozostają łatwo dostępne w dolnej części widoku.



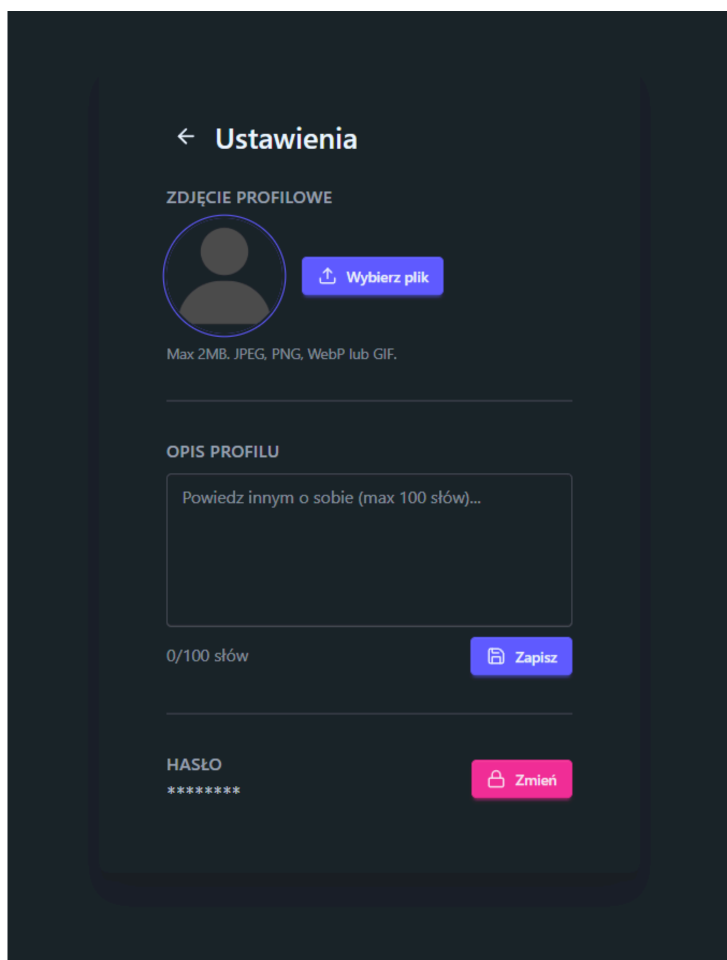
Rys. 5.7. Interfejs profilu użytkownika [opracowanie własne]

5.7 Ustawienia konta

Strona ustawień umożliwia użytkownikowi podstawową personalizację konta oraz zarządzanie wybranymi elementami profilu. W górnej części interfejsu znajduje się sekcja edycji zdjęcia profilowego, pozwalająca na wgranie pliku graficznego w określonym formacie i rozmiarze. Mechanizm ten umożliwia wizualne dostosowanie profilu użytkownika.

Poniżej umieszczono pole edycji opisu profilu, w którym użytkownik może wprowadzić krótki tekst biograficzny o ograniczonej liczbie słów. Interfejs informuje o aktualnym limicie znaków oraz udostępnia przycisk zapisu zmian.

W dolnej części strony znajduje się sekcja zarządzania hasłem, prezentowana w formie zaciemnionej w celu ochrony poufności danych. Zmiana hasła realizowana jest poprzez dedykowany przycisk, inicjujący odpowiednią procedurę aktualizacji danych uwierzytelniających.



Rys. 5.8. Interfejs edycji ustawień użytkownika [opracowanie własne]

6. Testy

W niniejszym rozdziale przedstawiono proces weryfikacji poprawności działania aplikacji z wykorzystaniem testów automatycznych. W projekcie zastosowano dwa uzupełniające się poziomy testowania: testy jednostkowe weryfikujące pojedyncze funkcje i moduły oraz testy integracyjne oceniające współdziałanie większych komponentów systemu, takich jak routing, middleware czy logika serwisowa.

Wszystkie testy automatyczne w projekcie zostały zorganizowane zgodnie z powszechnie stosowanym wzorcem **AAA** (Arrange-Act-Assert), który porządkuje proces testowania poprzez wyraźny podział na etap przygotowania danych i konfiguracji środowiska testowego, wykonanie testowanej operacji oraz weryfikację uzyskanych rezultatów. Takie podejście zwiększa czytelność testów oraz ułatwia ich utrzymanie i rozwój. Poprawność działania poszczególnych elementów systemu weryfikowana jest dzięki **asercjom**, które umożliwiają porównanie rzeczywistych wyników z wartościami oczekiwanymi, sprawdzanie typów danych, kodów odpowiedzi **HTTP** oraz struktury zwracanych obiektów. [5]

Drugim ważnym mechanizmem wspierającym proces testowania jest tworzenie atrap (ang. *mocking*) zależności, co pozwala testować logikę biznesową w izolacji od usług zewnętrznych, baz danych czy zapytań **HTTP**. Zastąpienie rzeczywistych wywołań kontrolowanymi atrapami umożliwia uzyskanie przewidywalnych wyników testów oraz eliminuje wpływ czynników niezależnych od aplikacji. Takie podejście jest szczególnie istotne w aplikacjach webowych, gdzie bezpośrednia komunikacja z zewnętrznymi serwisami mogłaby prowadzić do niestabilności testów oraz wydłużenia czasu ich wykonywania. [2]

Do implementacji testów wykorzystano framework **Vitest**, będący nowoczesnym narzędziem do testowania aplikacji opartych na JavaScript. Jego integracja z ekosystemem **Vite** umożliwia bardzo szybkie wykonywanie testów oraz pełne wsparcie modułów ESM, stosowanych w całym projekcie. [15]

Testy integracyjne zostały dodatkowo wsparte biblioteką **Supertest**, która umożliwia wykonywanie żądań **HTTP** bez konieczności uruchamiania rzeczywistego serwera. Umożliwia to przetestowanie pełnego przepływu żądania od routera, przez middleware, po końcową odpowiedź w sposób szybki i przewidywalny.

6.1 Testy jednostkowe

Celem testów jednostkowych jest wspieranie długoterminowego, zrównoważonego rozwoju oprogramowania poprzez wczesne wykrywanie błędów, co pozwala zachować stabilność systemu i utrzymać stałe tempo rozwoju projektu mimo rosnącej złożoności kodu. [5]

Testy jednostkowe koncentrują się na weryfikacji poprawności działania pojedynczych funkcji lub metod w izolacji od pozostałych komponentów systemu. Głównym celem tej kategorii testów jest zapewnienie, że każda jednostka kodu realizuje swoją odpowiedzialność zgodnie ze specyfikacją, niezależnie od stanu czy zachowania innych modułów. W projekcie testy jednostkowe obejmują serwisy odpowiedzialne za komunikację z zewnętrznymi **API**, weryfikując zarówno prawidłowe przetwarzanie odpowiedzi zawierających kompletne dane, jak i obsługę przypadków brzegowych.

W kontekście aplikacji webowej tworzenie atrap dotyczy przede wszystkim bibliotek realizujących zapytania **HTTP**, takich jak axios, których rzeczywiste wywołanie w trakcie testów byłoby nieefektywne i wprowadziłoby niepożądane zależności od dostępności zewnętrznych serwisów.

6.1.1 Test obsługi książek bez okładki

Jeden z testowanych scenariuszy weryfikuje poprawność obsługi danych zwracanych przez API Open Library w sytuacji, gdy książka nie zawiera informacji o okładce. W odpowiedzi **API** pole "cover_i" przyjmuje wartość null, co uniemożliwia wygenerowanie adresu URL okładki.

W teście wywołanie metody axios.get zostaje zastąpione atrapą, która symuluje odpowiedź **API** zawierającą rekord książki bez identyfikatora okładki. Następnie wywoływana jest funkcja openLibraryService.search, odpowiedzialna za pobieranie i przetwarzanie danych do formatu wewnętrznego aplikacji.

Zgodnie z aktualną logiką systemu rekordy książek, które nie zawierają informacji o okładce, są filtrowane na etapie przetwarzania danych i nie są uwzględniane w wynikach wyszukiwania. W konsekwencji funkcja search zwraca pustą tablicę wyników, mimo poprawnej odpowiedzi otrzymanej z zewnętrznego **API**.

Test potwierdza poprawne filtrowanie niekompletnych danych z zewnętrznego **API** oraz zapewnia spójność wyników prezentowanych użytkownikowi.

Listing 6.1. Test obsługi książek bez okładki [opracowanie własne]

```
 describe('search - no cover', () => {
    it('should return empty array when books have no cover', async
() => {
      axios.get.mockResolvedValue({
        data: {
          docs: [
            {
              key: '/works/OL12345W',
              title: 'Test Book Without Cover',
              author_name: ['Test Author'],
              first_publish_year: 2020,
              cover_i: null,
              isbn: ['1234567890']
            }
          ]
        }
      });

           const  results  =  await  openLibraryService.search('test
query');

      expect(results).toEqual([]);
      expect(axios.get).toHaveBeenCalledWith(
        'https://openlibrary.org/search.json',
        expect.objectContaining({
          params: expect.objectContaining({ q: 'test query' })
        })
      );
    });
```

6.1.2 Test rejestracji użytkownika z istniejącym adresem e-mail

Test weryfikuje obsługę sytuacji, w której użytkownik próbuje zarejestrować się z adresem e-mail już istniejącym w systemie. W scenariuszu testowym metoda prisma.user.findUnique zostaje zastąpiona atrapą, która symuluje istnienie użytkownika o adresie "existing@test.com" w bazie danych. Następnie wywoływana jest funkcja signup z poprawnymi danymi rejestracyjnymi przekazanymi w obiekcie żądania.

Test sprawdza, czy w przypadku wykrycia istniejącego adresu e-mail proces rejestracji zostaje przerwany, a serwer zwraca odpowiednią odpowiedź HTTP.

Weryfikowane są następujące warunki:

- serwer zwraca kod odpowiedzi **HTTP** 409,
- zwrócony komunikat **JSON** informuje o istnieniu użytkownika z podanym adresem e-mail,
- metoda sprawdzająca istnienie użytkownika (findUnique) zostaje wywołana z poprawnym parametrem,
- operacja tworzenia nowego użytkownika (prisma.user.create) nie zostaje wykonana.

Test potwierdza poprawną obsługę walidacji unikalności danych po stronie serwera oraz zabezpiecza system przed tworzeniem zduplikowanych kont użytkowników.

Listing 6.2. Test rejestracji użytkownika [opracowanie własne]

```
it('should return 409 for existing email', async () => {
    mockReq = { body: validSignupData };
    prisma.user.findUnique.mockResolvedValue({
      id: 1,
      email: 'existing@test.com',
      name: 'Existing User'
    });
    await signup(mockReq, mockRes);
    expect(mockRes.status).toHaveBeenCalledWith(409);
    expect(mockRes.json).toHaveBeenCalledWith({
      message: 'User with this email already exists'
    });
    expect(prisma.user.findUnique).toHaveBeenCalledWith({
      where: { email: 'existing@test.com' }
    });
    expect(prisma.user.create).not.toHaveBeenCalled(); });
```

6.2 Testy integracyjne

Testy integracyjne weryfikują poprawne działanie kodu w połączeniu z innymi komponentami systemu oraz zależnościami zewnętrznymi, takimi jak bazy danych czy usługi. Ich celem jest sprawdzenie, czy poszczególne części aplikacji poprawnie ze sobą współpracują w realistycznym scenariuszu użycia. [5] W przypadku aplikacji opartych na **Express** szczególnie istotne jest przetestowanie poprawności obsługi tras, komunikacji między warstwami oraz formatu zwracanych danych. Dzięki **Supertest** możliwe jest symulowanie pełnych żądań **HTTP** bez uruchamiania aplikacji na porcie, co upraszcza testy i pozwala na szybkie ich wykonywanie.

6.2.1 Test integracji z API TMDB

Test weryfikuje poprawność działania endpointu **GET** /api/media/search, odpowiedzialnego za wyszukiwanie filmów z wykorzystaniem zewnętrznego **API** TMDB do pobierania informacji o filmach i serialach. Żądanie **HTTP** wykonywane jest przy użyciu biblioteki Supertest i zawiera parametry zapytania określające typ medium oraz frazę wyszukiwania. Dodatkowo w nagłówku przesyłany jest token dostępu w postaci pliku cookie, co umożliwia autoryzację żądania w kontekście użytkownika zalogowanego.

Ze względu na komunikację z zewnętrznym serwisem test realizowany jest z wydłużonym limitem czasowym wynoszącym 15 sekund, co pozwala uwzględnić potencjalne opóźnienia odpowiedzi sieciowej.

W fazie asercji weryfikowane są następujące warunki:

- serwer zwraca kod odpowiedzi **HTTP** 200,
- pole **results** w odpowiedzi stanowi tablicę,
- tablica wyników zawiera co najmniej jeden element,
- każdy zwrócony obiekt filmu posiada właściwość title typu tekstowego.

Test potwierdza poprawną obsługę zapytań wyszukiwania, prawidłową integrację z **API** TMDB oraz poprawność struktury danych zwracanych przez endpoint aplikacji.

Listing 6.3. Test połączenia z API [opracowanie własne]

```
it('should return movie search results from TMDB', async () => {
   const response = await request(app)
     .get('/api/media/search')
     .query({ type: 'movie', query: 'Matrix' })
     .set('Cookie', ['accessToken=valid-test-token']);

   expect(response.status).toBe(200);

   expect(response.body.results).toBeInstanceOf(Array);

   expect(response.body.results.length).toBeGreaterThanOrEqual(1);

   response.body.results.forEach(movie => {
     expect(movie).toHaveProperty('title');
     expect(typeof movie.title).toBe('string');
   });
 }, 15000);
```

6.2.2 Test wyszukiwania książek z Open Library

W ramach testów integracyjnych zweryfikowano działanie endpointu **GET** /api/media/search, odpowiedzialnego za wyszukiwanie książek z wykorzystaniem danych pochodzących z **API** Open Library. W fazie przygotowania testu funkcja openLibraryService.search, wykorzystywana w warstwie kontrolera, zostaje zastąpiona atrapą zwracającą przygotowaną tablicę przykładowych wyników **mockBooks**.

Następnie przy użyciu biblioteki Supertest wykonywane jest żądanie **HTTP** typu **GET** z parametrami zapytania określającymi typ medium oraz frazę wyszukiwania. Żądanie zawiera również token dostępu przekazany w postaci pliku cookie, co symuluje kontekst użytkownika zalogowanego.

W fazie asercji weryfikowane są następujące elementy:

- serwer zwraca kod odpowiedzi **HTTP 200,**
- odpowiedź zawiera pola results oraz count,
- pole results stanowi tablicę obiektów, a count odzwierciedla liczbę zwróconych wyników,
- każdy obiekt książki posiada wymagane atrybuty, takie jak identyfikator, tytuł, autorzy, okładka, rok wydania,
- warstwa kontrolera poprawnie wywołuje funkcję serwisową z przekazaną frazą wyszukiwania.

Test potwierdza, że endpoint poprawnie obsługuje parametry zapytania, współpracuje z warstwą serwisową oraz zwraca dane w spójnym i oczekiwanym formacie **JSON**.

Listing 6.4. Test wyszukiwania książek [opracowanie własne]

```
 it('should return 200 with valid JSON for book search', async ()
=> {
    openLibraryService.search.mockResolvedValue(mockBooks);

    const response = await request(app)
      .get('/api/media/search')
      .query({ type: 'book', query: 'great book' })
      .set('Cookie', ['accessToken=valid-test-token']);

    expect(response.status).toBe(200);
    expect(response.body).toHaveProperty('results');
    expect(response.body).toHaveProperty('count');
    expect(response.body.results).toBeInstanceOf(Array);
    expect(response.body.count).toBe(2);

    const book = response.body.results[0];
    expect(book).toHaveProperty('id');
    expect(book).toHaveProperty('title');
    expect(book).toHaveProperty('authors');
    expect(book).toHaveProperty('coverImage');
    expect(book).toHaveProperty('year');
      expect(openLibraryService.search).toHaveBeenCalledWith('great
book');
  });
```

6.2.3 Test dodawania kolekcji

Test weryfikuje działanie endpointu **POST** /api/collections, odpowiedzialnego za tworzenie nowych kolekcji użytkownika. Scenariusze testowe obejmują zarówno przypadek błędnego żądania, jak i poprawne dodanie podkolekcji. W celu izolacji logiki kontrolera operacje na bazie danych realizowane przez Prisma ORM zostają zastąpione atrapami.

W pierwszym scenariuszu sprawdzana jest reakcja systemu na brak wymaganej nazwy kolekcji. Wysłanie żądania **POST** bez danych powoduje zwrócenie kodu odpowiedzi **HTTP 400**, co potwierdza poprawną walidację danych wejściowych po stronie serwera.

Drugi scenariusz testuje poprawny proces tworzenia podkolekcji. Po zasymulowaniu istnienia kolekcji nadrzędnej oraz braku powiązanych tagów wykonywane jest żądanie **POST** z nazwą nowej kolekcji. W fazie **asercji** weryfikowane jest wywołanie metody prisma.collection.create oraz kod odpowiedzi serwera.

Test potwierdza poprawne przetwarzanie żądań tworzenia kolekcji, działanie walidacji oraz współpracę warstwy kontrolera z warstwą dostępu do danych.

Listing 6.5. Test dodawania kolekcji [opracowanie własne]

```
 describe('POST /api/collections', () => {
    it('should return 400 when name missing', async () => {

prisma.collection.findFirst.mockResolvedValue(mockMainCollection);

      const response = await request(app)
        .post('/api/collections')
        .set('Cookie', ['accessToken=valid-token'])
        .send({});

      expect(response.status).toBe(400);
    });

    it('should create new subcollection', async () => {

prisma.collection.findFirst.mockResolvedValue(mockMainCollection);
      prisma.tag.findMany.mockResolvedValue([]);
      const newCollection = {
        id: 3,
        name: 'Nowa Kolekcja',
        isMainCollection: false,
        isPublic: false,
        parentId: 1,
        userId: 1,
        collectionTags: []
      };
      prisma.collection.create.mockResolvedValue(newCollection);

      const response = await request(app)
        .post('/api/collections')
        .set('Cookie', ['accessToken=valid-token'])
        .send({ name: 'Nowa Kolekcja' });

      expect(prisma.collection.create).toHaveBeenCalled();
      expect([200, 201]).toContain(response.status);
    });
```

7. Podsumowanie i wnioski

Celem pracy inżynierskiej było opracowanie funkcjonalnego narzędzia do zarządzania osobistymi kolekcjami multimediów, które w praktyczny sposób rozwiązuje problem rozproszenia informacji o preferencjach użytkowników. W ramach projektu udało się zrealizować wszystkie wymagania funkcjonalne i niefunkcjonalne.

System umożliwia użytkownikom bezpieczne tworzenie kont oraz uwierzytelnianie z wykorzystaniem mechanizmów haszowania haseł i zarządzania sesjami. Zaimplementowana integracja z zewnętrznymi API znacząco ułatwia proces dodawania nowych pozycji do kolekcji poprzez automatyczne pobieranie metadanych oraz okładek. Funkcjonalności tagowania, wyszukiwania i filtrowania działają efektywnie, zapewniając użytkownikom intuicyjny sposób nawigacji po własnych zbiorach. Mechanizm publicznych list pozwala na udostępnianie wybranych kolekcji innym użytkownikom. Poprawność działania systemu zweryfikowano poprzez testy obejmujące główne scenariusze użytkowania.

Możliwe kierunki rozwoju aplikacji w przyszłości:

- **Rozwój funkcji społecznościowych aplikacji** - implementacja systemu obserwowania innych użytkowników, tworzenia grup tematycznych oraz wysyłania wiadomości bezpośrednich. Rozszerzenie to pozwoliłoby na budowanie społeczności wokół wspólnych zainteresowań medialnych oraz odkrywanie nowych pozycji na podstawie aktywności innych użytkowników.
- **Opracowanie aplikacji mobilnej** - implementacja natywnych aplikacji na systemy iOS oraz Android, umożliwiających pełny dostęp do funkcjonalności systemu z urządzeń mobilnych.
- **Integracja modeli sztucznej inteligencji** - wykorzystanie algorytmów uczenia maszynowego do tworzenia spersonalizowanych rekomendacji na podstawie historii ocen użytkownika oraz wzorców konsumpcji mediów. System mógłby również zastosować przetwarzanie języka naturalnego do automatycznej analizy recenzji oraz inteligentnego sugerowania tagów podczas dodawania nowych pozycji.

Realizacja projektu pozwoliła na praktyczne zastosowanie wiedzy z zakresu inżynierii oprogramowania oraz projektowania interfejsów użytkownika. Zaprojektowany system stanowi funkcjonalną bazę, która może być rozwijana zgodnie z przedstawionymi kierunkami oraz potrzebami użytkowników.

Bibliografia

[1] Alex Banks, Eve Porcello. *Learning React. 2nd Edition*, O'Reilly Media, 2024

[2] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley Professional, 2002

[3] Niels Ferguson, Bruce Schneier, Tadayoshi Kohno. *Cryptography Engineering: Design Principles and Practical Applications.* Wiley, 2010

[4] Joseph Ingeno. *Software Architect's Handbook.* Packt Publishing, 2018

[5] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns.* Manning, 2020

[6] Sam Newman. B*uilding Microservices: Designing Fine-Grained Systems.* 2nd Edition, O'Reilly Media, 2021

[7] Abraham Silberschatz, Henry F. Korth, S. Sudarshan. *Database System Concepts.* 7th Edition, McGraw-Hill Education, 2020

[8] Kyle Simpson. *You Don't Know JS Yet: Get Started.* Wydanie własne, 2020

[9] Express [Online] https://expressjs.com/

[10] JSON [Online] https://www.json.org/json-en.html

[11] JWT [Online] https://www.jwt.io/introduction

[12] Node.js [Online. https://nodejs.org/en/about

[13] OWASP [Online] https://cheatsheetseries.owasp.org/cheatsheets

[14] TailwindCSS [Online] https://tailwindcss.com/docs

[15] Vitest [Online]. https://vitest.dev/guide/

Spis rysunków

Rys. 4.1. Diagram przypadków użycia [opracowanie własne]

Rys. 4.2. Diagram ERD głównych encji systemu [opracowanie własne]

Rys. 5.1. Interfejs strony głównej [opracowanie własne]

Rys. 5.2. Interfejs logowania [opracowanie własne]

Rys. 5.3. Interfejs rejestracji [opracowanie własne]

Rys. 5.4. Interfejs wyszukiwania [opracowanie własne]

Rys. 5.5. Interfejs własnej kolekcji [opracowanie własne]

Rys. 5.6. Interfejs edycji multimediów [opracowanie własne]

Rys. 5.7. Interfejs profilu użytkownika [opracowanie własne]

Rys. 5.8. Interfejs edycji ustawień [opracowanie własne]

Spis listingów

Listing 4.1. Fragment funkcji rejestracji użytkownika [opracowanie własne]

Listing 4.2. Fragment funkcji weryfikacji tokenu [opracowanie własne]

Listing 6.1. Test obsługi książek bez okładki [opracowanie własne]

Listing 6.2. Test rejestracji użytkownika [opracowanie własne]

Listing 6.3. Test połączenia z API [opracowanie własne]

Listing 6.4. Test wyszukiwania książek [opracowanie własne]

Listing 6.5. Test dodawania kolekcji [opracowanie własne]