

Uniwersytet SWPS  
Wydział Projektowania w Warszawie  
Informatyka  
studia pierwszego stopnia

autor: Tetiana Borozdina  
nr albumu studenta: 73795

Praca licencjacka

**[TYTUŁ PRACY DYPLOMOWEJ]**

tytuł w języku polskim: [wpisać]

Praca napisana pod kierunkiem  
[tytuł/stopień naukowy, imię i nazwisko promotora]

Warszawa 2025

## Spis treści

Streszczenie .....	3
Wprowadzenie .....	1
Metodologia .....	2
Projektowanie .....	2
Wyznaczenie wymagań do funkcjonalności aplikacji .....	2
Określenie przestrzeni projektowej .....	3
Organizacja baz danych .....	4
Architektura serwisowa .....	8
Implementacja.....	11
Organizacja serwisów z dostarczenia danych strukturalnych.....	11
Organizacja serwisu z dostarczenia danych multimedialnych .....	16
Organizacja serwisu z przekazywania danych według logiki biznesowej .....	18
Organizacja serwisów z prezentacji widoków .....	21
Organizacja serwisu z uwierzytelnienia użytkowników .....	26
Wyniki.....	28
Testowanie podstawowych zapytań.....	28
Wnioski.....	35
Bibliografia .....	37

## Streszczenie

...

## Wprowadzenie

Wzrost liczby dostępnych do konsumpcji treści multimedialnych przełożył się na znaczne rozszerzenie oferty dla celowych odbiorców. Odbiorcy z innej strony zyskali możliwość dopasowania treści do własnych zainteresowań, nie będąc ograniczonymi dostępnością materiałów. Według tego zasadne jest stosowanie narzędzi umożliwiających prowadzenie wyszukiwań według określonych kategorii filmowych. Dane o filmach przy tym gromadzą się w jednym miejscu, tworząc uporządkowaną bibliotekę.

Postawienie problemu: projektowanie aplikacji internetowej, celem której jest przedstawienie pomocy użytkownikom w znajdowaniu, filtrowaniu, porządkowaniu informacji o filmach na podstawie zainteresowań użytkowników. Aplikacja także musi prezentować i kategoryzować informacje powiązane, w tym dane o podmiotach zaangażowanych w produkcji, a także pozwalać dodawanie i przeglądanie ocen wystawianych przez użytkowników na pewne filmy.

W celu realizacji postawionego problemu niezbędne jest:

- a) projektowanie modułowej struktury projektu, opartej na niezależnych serwisach;
- b) projektowanie struktury bazy danych, która będzie przechowywać informacje o różnych obiektach;
- c) organizacja komunikacji między poszczególnymi serwisami;
- d) organizacja procesu projektowego obejmującego testowanie, wdrażanie części aplikacji.

Do każdego z powyższych punktów można przedstawić szereg bardziej szczegółowych zagadnień, opisanych dokładnie w następnych rozdziałach.

Dla implementacji rozwiązania wybrano podejście mikroservisowe, zakładające podział aplikacji na mniejsze, niezależne jednostki, komunikujące się ze sobą. Dane podejście umożliwia:

- a) wyodrębnienie jednostek według zakresów odpowiedzialności;
- b) skupienie się na odrębnych jednostkach podczas przeprowadzenia implementacji;
- c) przeprowadzenie testów jednostkowych bez konieczności uruchamiania całego rozwiązania;
- d) wprowadzenie przejrzystości w strukturę, co ułatwia znajdowanie oraz eliminację komponentów nadmiarowych.

## **Metodologia**

### **Projektowanie**

W niniejszym rozdziale opisano szczegóły, dotyczące formalizacji założeń projektowych z opisem decyzji podjętych wobec wyboru określonych podejść metodologicznych.

#### **Wyznaczenie wymagań do funkcjonalności aplikacji**

Pierwszym etapem formalizacji założeń stanowi określenie wymagań dotyczących funkcjonalności oraz logiki biznesowej aplikacji.

##### ***Funkcjonalność użytkownika***

Podstawowa funkcjonalność danej aplikacji z perspektywy użytkownika jest uzależniona od trybu użytkownika – anonimowego („gościa”) lub zalogowanego, oraz od jego roli w serwisie. Przewidziano dwa podstawowe typy ról – zwykłego użytkownika oraz administratora.

W trybie anonimowym użytkownik ma dostęp do następujących działań:

- przeglądanie informacji o filmach;
- nawigacja po stronach aplikacji przy użyciu odnośników;
- wyszukiwanie, filtrowanie informacji o filmach według kategorii;
- założenie konta lub logowanie do serwisu.

Opcja logowania udostępnia się użytkownikowi po założeniu konta. Po logowaniu wskazana powyżej funkcjonalność rozszerza się do następujących działań:

- ocenianie filmów w skali od jednego do dziesięciu;
- oznaczenie filmu etykietami: „ulubiony”, „zaplanowany”, „w trakcie oglądania”, „obejrzany”;
- podstawowa personalizacja własnego profilu z umieszczeniem opcjonalnych informacji o sobie i ulubionych filmach, gatunkach filmowych.

Dodatkowe działania obejmują możliwość wylogowania oraz usunięcie konta.

Tryb administratora nadaje użytkownikowi uprawnienia na dodawanie, edytowanie oraz usuwanie informacji o filmach, a także informacjach powiązanych – osobach, wytwórniach filmowych.

Nadawanie lub odbieranie roli administratora jest realizowane przez użytkownika posiadającego rolę superadministratora. Do jego uprawnień należy również zarządzanie kontami użytkowników o mniej przywilejowanych rolach z możliwością usuwania ich kont.

W taki sposób powstaje hierarchia ról użytkowników, każda z których cechuje się różnym zakresem możliwych czynności. Role decydują o to, czy informacje wyłożone na stronach są udostępnione dopiero do przeglądania, albo także do edytowania.

### ***Funkcjonalność aplikacji***

Z perspektywy działania systemu, aplikacja powinna realizować następujące funkcje:

- zwracanie stron internetowych w jakości odpowiedzi na zapytania klienckie;
- automatyczne generowanie widoków stron;
- pobieranie odpowiednich informacji z bazy danych;
- filtrowanie, sortowanie danych według ustalonych kategorii;
- wyszukiwanie danych według słów kluczowych;
- kontrolowanie spójności nadchodzących danych, w tym ich poprawności;
- logowanie obejmujące kontrolę procesów uwierzytelniania tożsamości oraz autoryzacji użytkownika z przydzieleniem dostępu do odpowiedniego zakresu działań.

### **Określenie przestrzeni projektowej**

Dla ułatwienia procesu przejścia od formalizacji do bezpośredniej implementacji przyjęto reguły podejścia Domain-Driven Design (DDD), zakładające projektowanie oprogramowania w ten sposób, aby jego system odzwierciedlał modele dziedziny jak najwiernie do rzeczywistych pojęć [1, s. 29]. Do tego, model dziedziny staje centralnym elementem procesu projektowego, gdy pozostałe części systemu zależą na jego określeniu.

Przestrzeń projektową formuje zbiór pojęć wykorzystywanych w ciągu procesu projektowego. Definicja tych pojęć nie tylko sprzyja współrozumieniu pomiędzy stroną zamawiającą a stroną wykonującą, ale również ułatwia komunikację pomiędzy wykonawcami należących do różnych zaangażowanych zespołów projektowych. Przestrzeń projektowa niesie nazwę, którym opisuje zbiór pojęć dziedziny, jednak także może utożsamiać się z nazwą samej aplikacji.

W przestrzeni projektowej aplikacji, której zadaniem jest gromadzenie materiałów o charakterze encyklopedycznym, większość pojęć wiąże się z treścią przedstawianych na stronach informacji. Wyodrębnia się w taki sposób dziedzina odpowiadająca tematyce, której w danym przypadku jest tematyka filmowa. Formują jej takie obiekty, jak „film“, „wytwórnia filmowa“, „artysta“, „reżyser“, „producent“, „scenarzysta“. Przeliczone obiekty stanowią jednocześnie kategorie informacji prezentowanych na stronach.

Pozostałymi pojęciami przestrzeni są subiekty wchodzące w interakcję z treściami na stronie. Są to kategorie osób – użytkownik niezalogowany („gość”), użytkownik zalogowany, administrator, superadministrator.

### **Organizacja baz danych**

Na wcześniejszym etapie zostały wyodrębnione obiekty przestrzeni projektowej. W niniejszej części odbywa się dalsza formalizacja problemu, ze skupieniem na strukturze wyznaczonych kategorii danych. Struktura opisanych modeli danych orientowana jest na ich przechowywanie w bazie danych typu NoSQL.

Projektowanie bazy danych umożliwia bardziej szczególne modelowanie struktury danych. Pojęcie bazy danych oznacza zbiór połączonych informacji o wartościach bezwzględnych, spójnych logicznie i możliwych do zapisania [2, s. 5]. Z kolei do projektowania bazy wchodzi:

- określenie encji, którymi są obiekty opisywane bazą danych, modelujące obiekty przestrzeni projektowej
- przeliczenie listy atrybutów, czyli cech, które opisują encje poprzez pewne przeznaczone wartości [2, s. 6].

Część atrybutów potrafi mieć charakter złożony. W przełożeniu na bazy danych, atrybuty opisujące encję przedstawiają sobą zapisy z innych tabel, które wiążą się z tabelą encji za pośrednictwem kluczy obcych. Pełny model, który powstaje przez połączenie informacji z różnych tabel jest nazywany agregatowym [1, s. 76].

Z kolei nie każdy obiekt wymaga reprezentacji przez osobną encję. Obiekt, który pełni jedynie funkcję opisową i nie posiada własnej tożsamości, nazywa się obiektem wartościowym (ang. „value object”) [2, s. 59].

Encje aplikacji odpowiadają określonym wcześniej pojęciom dziedziny projektowej. Modelowanie struktur danych rozpoczęto od formalizacji pojęcia użytkownika. Nie zważając na

to, że wprowadzone na wcześniejszym etapie pojęcia użytkowników różnią się według ról, ich pozostałe cechy są podobne. Z tego powodu przedstawione pojęcia zdecydowano połączyć pod wspólną encją „użytkownik” z atrybutami nadanymi niżej w tabeli.

User		
Nazwa	Typ danych	Opis
Id	String	Identyfikator zapisu.
Username	String	Imię unikatowe użytkownika.
Birthdate	Date	Data urodzenia (opcjonalnie).
Picture	String	Nazwa zdjęcia profilu.
CinemaRecord		
Id	String	Identyfikator zapisu.
CinemaId	String	Identyfikator filmu.
UserId	String	Identyfikator użytkownika.
Label	Label	Etykieta, którą oznaczono film.
AddedAt	DateTime	Czas etykietowania filmu.
Label		
Value	Integer	Znaczenie przeznaczone etykietce.
Name	String	Nazwa etykiety.
CinemaRating		
Id	String	Identyfikator zapisu.
CinemaId	String	Identyfikator filmu.
UserId	String	Identyfikator użytkownika.
RatingScore	Double	Znaczenie oceny.

Tabela 1 – Model agregatowy encji „użytkownik” („user”).

Pierwsze z pojęć dziedziny tematycznej odpowiada obiektowi „film” i cechuje się atrybutami, opisanymi w następującej tabeli:

Cinema		
Nazwa	Typ danych	Opis
Id	String	Identyfikator zapisu.
Name	String	Nazwa filmu.



ReleaseDate	Date	Data wyjścia.
Genres	Genre	Gatunki filmowe.
Language	Language	Język oryginału filmu.
RatingScore	RatingScore	Ocena użytkowników.
ProductionStudios	Array<StudioRecord>	Wytwórnie filmowe produkujące film.
Starrings	Array<Starring>	Osoby uczestniczące w produkcji.
Description	String	Opis filmu.
Picture	String	Nazwa zdjęcia plakatu.
Genre		
Value	Integer	Znaczenie przypisane gatunkowi.
Name	String	Nazwa gatunku.
Language		
Value	Integer	Znaczenie przypisane gatunkowi.
Name	String	Nazwa języka.
RatingScore		
N	Integer	Ogólna ilość ocen od użytkowników.
Score	Double	Średnia z ocen użytkowników.
StudioRecord		
Id	String	Identyfikator wytwórni.
Name	String	Nazwa wytwórni.
Picture	String	Nazwa zdjęcia wytwórni.
Starring		
Id	String	Identyfikator osoby.
Name	String	Imię osoby.
Jobs	Job	Rola osoby w produkcji filmu.
ActorRole	ActorRole	Rola, którą gra artysta (opcjonalnie).
Picture	String	Nazwa zdjęcia osoby.
Job		
Value	Integer	Znaczenie przypisane roli w produkcji.
Name	String	Nazwa roli w produkcji.
ActorRole		
Id	String	Identyfikator roli.
Name	String	Nazwa roli w filmie.
Priority	RolePriority	Priorytet roli.

RolePriority		
Value	Integer	Znaczenie przypisane priorytetowi.
Name	String	Nazwa poziomu priorytetu.

Tabela 2 – Model agregatowy encji „film” („cinema”).

Następną encją służy encja „osoba”, która reprezentuje jednocześnie pojęcia „artysta“, „reżyser“, „producent“, „scenarzysta“ – przeliczone nazwy zdecydowano wykorzystać dopiero dla oznaczenia zawodu pewnej osoby.

Person		
Nazwa	Typ danych	Opis
Id	String	Identyfikator zapisu.
Name	String	Imię osoby.
BirthDate	Date	Data urodzenia.
Country	Country	Kraj urodzenia.
Jobs	Job	Zawody osoby.
Filmography	Array<CinemaRecord>	Filmy, w których osoba uczestniża.
Description	String	Informacja o życiu osoby.
Picture	String	Nazwa zdjęcia osoby.
Country		
Value	String	Znaczenie przypisane kraju.
Name	String	Nazwa kraju.
Job		
Value	Integer	Znaczenie przypisane roli w produkcji.
Name	String	Nazwa zawodu.
CinemaRecord		
Id	String	Identyfikator wytwórni.
Name	String	Nazwa filmu.
Year	Integer	Rok wyjścia filmu.
Picture	String	Nazwa zdjęcia plakatu.

Tabela 3 – Model agregatowy encji „osoba” („person”).

Ostatnią encją opisuje schemat reprezentujący wytwórnię filmową.

Studio		
Nazwa	Typ danych	Opis
Id	String	Identyfikator zapisu.
Name	String	Nazwa wytwórni.
FoundDate	DateTime	Data założenia.
Country	Country	Kraj operowania się.
Filmography	Array<CinemaRecord>	Filmy, produkowane przez wytwórnię.
PresidentName	String	Imię prezesa firmy.
Description	String	Informacja o wytwórni.
Picture	String	Nazwa zdjęcia wytwórni.
Country		
Value	Integer	Znaczenie przypisane kraju.
Name	String	Nazwa kraju.
CinemaRecord		
Id	String	Identyfikator wytwórni.
Name	String	Nazwa filmu.
Year	Integer	Rok wyjścia filmu.
Picture	String	Nazwa zdjęcia plakatu.

Tabela 4 – Model agregatowy encji „wytwórnia” („studio”).

### Architektura serwisowa

Zamiast tego, aby projektować aplikację jako jeden monolityczny system, w jakości stylu projektowego zdecydowano się na wykorzystanie architektury mikroservisowej. Aplikację mikroservisową stanowi zbiór autonomicznych usług, wykonujących proste pojedyncze polecenia, które współpracują ze sobą, aby wykonywać bardziej złożone operacje [3, s. 4]. Każda usługa realizuje pewną funkcjonalność i komunikuje się z pozostałymi za pomocą zdefiniowanych protokołów. Zaletami architektury mikroservisowej są:

- łatwość w utrzymaniu każdej z usług
- niezależność wdrażania i skalowania poszczególnych komponentów
- mały rozmiar każdej z komponenty systemu [4, s. 14].

Określone zalety posłużyły motywatorem do opierania się na dany styl architektury w ciągu

procesu projektowania funkcjonalnych komponentów aplikacji.

Wymiana danych między serwisami oraz ich klientami realizowana jest zgodnie z zasadami architektury REST („Representational State Transfer”). Definiuje się jej jako mechanizm zapewniający zestaw ograniczeń, które umożliwiają skalowalność interakcji komponentów, ogólność interfejsów, niezależne wdrażanie komponentów pośredniczących oraz skuteczne wyegzekwowanie bezpieczeństwa [4, s. 73]. Nadaje się ona podczas implementacji niezależnych modułów, które zgodnie z tymi zasadami implementują i wystawiają pewny szereg poleceń dla korzystania przez serwisy klienckie. Tym formalizuje ona proces przebiegu komunikacji między komponentami.

Do przesyłania danych wykorzystano protokół HTTP, przy którym wszystkie operacje nad zasobami mapują się na ograniczony zestaw metod: „GET”, „POST”, „PUT”, „DELETE”. Specyfikacja działań nad danymi odbywa się za pośrednictwem ścieżek URL.

W projektowaniu poszczególnych serwisów zastosowano wzorec projektowy Model-View-Controller (MVC), który określa podstawę aplikacji z wydzieleniem trzech technicznych warstw: warstwy danych, warstwy logiki biznesowej oraz warstwy prezentacji [3, s. 52]. Na podstawie tego wzorca wyodrębniono następujące podstawowe serwisy:

- a) serwis, który organizuje bezpośrednią komunikację z serwerem bazy danych, jednocześnie opisuje interfejs programowy do wymiany danych z pozostałymi serwisami na zasadach REST;
- b) serwis, przedstawiający interfejs programowy do wymiany danych według zasad REST, w metodach którego implementują się ograniczenia logiki biznesowej;
- c) serwis, organizujący jedynie prezentację danych na stronach.

Serwis a) został rozdzielony na dwa oddzielne serwisy z dostarczenia danych w skutku wydzielenia dwóch poddziedzin z ogólnej dziedziny projektowej. W ten sposób powstał interfejs danych o użytkownikach, oraz interfejs samego materiału encyklopedycznego, który tworzą informacje o filmach, osobach i wytwórniach. Aby zniwelować nacisk na jedną bazę, oraz możliwe zakłócenia przy poleceniach HTTP, każdemu serwisowi została przypisana oddzielna baza danych.

Każdy obiekt dziedziny, zgodnie z opisem wcześniejszych tabel, posiada pole „Picture” dla oznaczenia nazwy pliku z obrazem. Aczkolwiek obiekty samych obrazów w postaci danych binarnych mogą być przechowywane w bazach danych, jednak rozwiązanie tego typu jest zbyt nieefektywne. Jednocześnie optymalnym a prostym wariantem jest przechowywanie tego typu

danych na dysku twardym maszyny hostującej serwer z organizacją hierarchii folderów ze zdjęciami. Alternatywą jest użycie serwisów dodatkowych, przedstawiających usługę przechowywania danych w chmurze. Przewagą takiego rozwiązania polega w obecności już wbudowanych narzędzi, powiązanych ze sprawdzaniem całościowości oraz bezpieczeństwa ładowanych plików. Dodatkową przewagą jest to, że serwis staje mniej zależny od swojego hosta. Przyjmując do uwagi wyłożone powody, wyodrębniany jest dodatkowy serwis na potrzebę obsługi plików binarnych. W implementacji zdecydowano korzystać z drugiego podejścia, czyli przechowywać obrazy w chmurze.

Podobnie do wcześniejszych manipulacji z serwisem a), serwis c) został podzielony na dwa odrębne komponenty prezentujące dane odpowiednio do każdego z serwisów obsługujących zapytania do bazy danych. Jeden z nich odpowiada za wyświetlanie stron encyklopedycznych, inny wyświetla strony związane z profilem użytkownika.

Z kolei serwis b) nie został podzielony w celu dostosowywania się do każdej pary jednostek, ponieważ reprezentuje unifikowany interfejs, łączący wszystkie kategorie danych. W nim także zostają integrowane prawa logiki, a także odbywa się kontrola dostępu do zasobów. Dane podejście z zapewnieniem pojedynczego punktu wejścia klienta zorientowanego na usługi z dostarczenia zasobów nazywa się wzorcem bramy interfejsu oprogramowania („Gateway API”) [3, s. 68]. Brama przekazuje zapytania do podstawowych serwisów i przekształca ich odpowiedzi a jednocześnie obsługuje powiązane zapytania klienta, takie jak uwierzytelnianie i podpisywanie zapytań.

Według przedstawionych wcześniej decyzji została określona architektura, podstawowymi komponentami której są następujące serwisy:

- “Encyclopedia Service” – serwis, który obsługuje strony prezentujące dane encyklopedyczne, inaczej główne strony informacyjne.
- “Cinema Data Service” – serwis, obsługujący zapis i odczyt danych związanych z filmami oraz powiązanymi informacjami (osoby, wytwórnie);
- “Access Service” – serwis realizujący proces logowania, rejestracji i autoryzacji użytkowników, nadając im odpowiednie role i uprawnienia.
- “Gateway API” – serwis, który pełni rolę pośrednika („proxy”) między serwisami. Zapewnia centralny punkt wejścia do systemu i egzekwuje prawa logiki biznesowej.

- “Profile service” – serwis, który odpowiada za zarządzanie kontem użytkownika z wyświetleniem odpowiednich stron. Także umożliwia edycję informacji profilowych zalogowanym użytkownikom.
- “User Data Service” – serwis, który obsługuje zapis i odczyt danych o użytkownikach oraz zarządza informacjami powiązanymi.
- „Image Service” – serwis zapewniający przetwarzanie i przechowywanie danych multimedialnych w postaci zdjęć.

## **Implementacja**

Serwisy o wspólnym przeznaczeniu mają podobną wewnętrzną organizację. Implementację samego rozwiązania rozpoczęto z założenia jednostek z dostarczenia danych, które z jednej strony implementują standardowe metody do pobrania danych z bazy, a z innej strony prezentują interfejs programowy, z którego korzystają pozostałe serwisy dla przeprowadzenia zapytań do danych.

Do przechowywania danych dziedziny używany jest serwer MongoDB. Serwisy współdzielą serwer bazy danych, ale każdemu serwisowi odpowiada własna baza danych do przechowywania obiektów dziedziny. Obiekty bazy danych zostają połączone w kolekcje reprezentujące określone pojęcia. Kolekcje baz danych przechowują obiekty w postaci BSON dokumentów.

## **Organizacja serwisów z dostarczenia danych strukturalnych**

Serwisy „Cinema Data” oraz „User Data” pełnią rolę pośredników między bazą danych oraz wewnętrznymi komponentami aplikacji. Ich architektura została opracowana na podstawie omówionych podejść projektowych: podejściu DDD oraz wzorcu projektowym MVC. Na tej podstawie każdy serwis został podzielony na trzy warstwy:

- a) „Domain” – warstwa definicji dziedziny projektowej;
- b) „Infrastructure” – warstwa odczytu i transferowania danych z bazy;
- c) „API” – warstwa definicji interfejsów programowych.

W warstwie „Domain” zostają definiowane modele obiektów dziedziny w wyglądzie klas, zgodnie z paradygmatem programowania obiektowego. Wyróżniono kilka przestrzeni nazw

porządkujących klasy w odpowiednie logiczne grupy.

Nazwa każdej z podprzestrzeni nazw zawiera imię odpowiedniego folderu, zawierającego plik z wyznaczeniem klasy, lub innej struktury. W taki sposób, hierarchia przestrzeni nazw odnosi się do hierarchii ścieżkowej projektu, co zapewnia dodatkową informację o miejscu przechowywania modeli przy importowaniu klas odpowiedniej przestrzeni na początku pliku.

Warstwa „Domain” zawiera jedną przestrzeń „Aggregates” ze wszystkimi modelami agregatowymi, każdy z których określa własne podprzestrzenie logiczne. Wśród nich wydziela się podprzestrzeń „Base”, która zawiera klasy abstrakcyjne, dziedziczone przez wszystkie modele – są to klasa „Entity” – podstawowa dla wszystkich encji, a także „Value” – podstawowa dla wszystkich obiektów wartościowych. Tym samym dodano poziom abstrakcji dla obiektów głównych i pobocznych. Klasa „Entity” definiuje wspólne pola obiektów głównych – identyfikator, nazwę własną, nazwę obrazu, opis, a także dodatkowe służbowe pola: „IsDeleted” – oznaczające usunięcie obiektu, „CreatedAt” – zawierające czas dodania obiektu, także służące jako pole odnośnie którego odbywa się domyślne sortowanie obiektów. Poboczne klasy dziedziczą identyfikator, nazwę własną oraz nazwę zdjęcia.

Każda z encji głównych posiada osobną przestrzeń, gdy poboczne obiekty albo stanowią część przestrzeni pewnego modelu agregatowego, albo znajdują się w wspólnej przestrzeni „Shared” w przypadku, gdy obiekt wartościowy wiąże się z kilkoma encjami jednocześnie.

W pozostałym definiowane modele danych odzwierciedlają struktury pojęć, opisanych w tabelach części *Organizacja baz danych*.

Oprócz obiektów głównych i pobocznych, opisanych za pomocą klas, obowiązują znaczenia reprezentowane za pomocą mechanizmu przeliczenia (z ang. „enumeration”). Użycie tego mechanizmu sprawia, że wszystkim polom wspólnego pojęcia różniącym ze sobą dopiero nominalnie nadawano jest pewne znaczenie liczbowe, do którego mapowany jest wiersz określający znaczenie w sposób sensowny. W ten sposób pojęcie reprezentowane jest ograniczonym zbiorem znaczeń całkowitych, które potrafią być jasnie odczytane i zinterpretowane przez model danych.

Do uwagi odnośnie układania serwisów z regułami DDD przyjmuje się niezależność warstwy dziedziny projektowej od innych komponentów.

Drugą warstwą jest „Infrastructure”, która opisuje trzy przestrzeni: kontekstu bazy danych „Context”, repozytoriów danych „Repositories” oraz modeli danych do transferowania

„Models”.

Przestrzeń kontekstu zawiera jedną klasę, która wyznacza używany system zarządzania bazą danych. Ta klasa wprowadza zależność od zewnętrznego pakietu „MongoDB Driver”, z którego pobrano narzędzia do zarządzania serwerem bazy danych MongoDB. Takimi narzędziami występują klasy, obiekty których reprezentują bazy danych oraz ich kolekcje. Każdemu pojęciu dziedziny odpowiada osobna kolekcja, dodatkowe kolekcje służą dla opisu relacji „Wiele-do-Wielu” między nimi. W ten sposób klasa kontekstu zawiera jeden obiekt bazy danych oraz kilka obiektów reprezentujących kolekcje tej bazy.

Następną przestrzenią logiczną stanowi przestrzeń obiektów, implementujących wzorzec projektowy „Repository”. Sens tego wzorca polega na oddzieleniu standardowych metod operujących się bezpośrednio nad obiektami bazy danych od logiki zewnętrznej z definicją dodatkowych metod, właściwych dopiero dla konkretnego pojęcia. Podstawą metod wszystkich klas wzorca służą operacje CRUD („Create Read Update Delete”), z których złożono metody tworzenia, odczytu po identyfikatorze, edytowania oraz usuwania obiektów pewnej kolekcji. Dane metody tworzą w serwisie wspólny interfejs `IEntityRepository<T>`, który dziedziczy abstrakcyjna klasa `EntityRepository<T>`. Zapis w „ $\diamond$ ” oznacza klasę jako generyczną, czyli zaimplementowaną bez specyfikacji konkretnej klasy – litera „T” przy nazwie interfejsu uogólnia typ obiektu encji, wobec którego odbywają się operacje. W ten sposób metody o podobnych implementacjach nie zostają wyznaczone ponownie dla każdego typu. Odbywa się tym stosowanie właściwości polimorfizmu.

W klasie abstrakcyjnej `EntityRepository` zaimplementowano wszystkie metody CRUD, oprócz metody edytowania obiektów, skoro wymaga ona osobistego podejścia do każdego typu danych. Dodatkowo w tej klasie wyznaczono metodę znalezienia mnóstwa obiektów zgodnie z przedstawionymi parametrami filtrowania, sortowania i paginacji. Parametrami sortowania są nazwa pola oraz kolejność sortowania. Parametrami paginacji są ilość obiektów do pobrania oraz ilość obiektów do omijania przed rozpoczęciem pobrania. Parametrem filtrowania występuje funkcja anonimowa, która przyjmuje obiekt i zwraca pole logiczne, wyznaczające to, czy należy wejściowy obiekt do grupy celowej. Wyznaczono także podobną metodę, służącą do pobrania pojedynczego obiektu za przedstawionym warunkiem filtrowania. Opisane funkcje wykorzystywane są w jakości podstawy bardziej specyficznych metod do odczytu danych.

Klasy implementujące wzorzec „Repository” napisano odpowiednio dla każdej klasy,



modelującej encję dziedziny. Z tego powodu główne odróżnienia między nimi zakluczają się w zbiorach metod do odczytu danych, filtrowanie w których odbywa się odnośnie specyficznych dla odpowiedniego modelu pól.

Dane, odczytywane bezpośrednio z bazy, zostają przetransformowane do obiektów odpowiednich klas należących przestrzeni „Domain”. Przy zapisywaniu informacji do bazy, nad danymi obowiązują odwrotne operacje.

Ostatnią część podziału „Infrastructure” formują modele obiektów do transferowania (DTO, „Data Transfer Object”). Zdefiniowano w niej klasy, obiekty których są wykorzystywane do przesyłania danych między serwisami. W większości opierają się one o struktury klas warstwy „Domain” opisujących obiekty dziedziny, z rozłożeniem pól na bardziej proste oraz skróceniem ilości tych pól. W ten sposób DTO zawierają tylko niezbędne informacje dopasowane do konkretnych zapytań. Modele do transferowania są podzielone logicznie według tego, czy odpowiadające im zapytanie dotyczy pobierania pojedynczego obiektu, lub grupy obiektów. W drugim przypadku klasom nadano lżejszą strukturę, co zapewnia większą wydajność przy dokonaniu zapytań na mnóstwo obiektów. Z powodu przeprowadzenia podobnych specyfikacji liczba modeli DTO we właściwy sposób potrafi kilkakrotnie przewyższać liczbę modeli dziedziny.

Szczegółem stosownie modeli DTO jest to, że ich klasy zostały umieszczone do współdzielonej przestrzeni, dostępnej dla wszystkich serwisów. Wyniesienie obiektów do oddzielnej przestrzeni pozwala uniknąć zależności między serwisami, gdyż proste korzystanie z obiektów DTO nie zyskuje takiej zależności. W celu obsługiwanego pozostałych jednostek, w istniejących modelach obiektów wprowadzać się zmiany, lub także mogą zostawać napisane odrębne klasy.

Ostatnią warstwą serwisów „Cinema Data” oraz „User Data” jest „Api”, która łączy funkcjonalność powiązaną z zewnętrzną logiką dostępu do danych. Do formalizacji i strukturyzacji pojęcia „zapytanie” używa się wzorca projektowego CQRS („Command Query Responsibility Segregation”), który cechuje się podziałem ogólnego zbioru zapytań na komendy powiązane ze zmianą stanu obiektów, oraz zapytania z prostym odczytem obiektów. Sens tego wzorca polega na rozdzieleniu obowiązków, według czego odbywa się zabezpieczenie optymalizacji z dostosowywaniem do potrzeb różnych modeli zapytań, jako tych powiązanych z wydajnością, skalowalnością, lub bezpieczeństwem [4, s. 235]. Według tego obsługiwanie zapytań obejmuje początkowe transformacje obiektów DTO typu „Request” do odpowiednich obiektów CQRS

przed korzystaniem z celowych metod dostępu do danych. Klasy każdego z podziałów zdefiniowane są w odpowiednich przestrzeniach nazw „Commands” oraz „Queries”, które zostały rozdzielone na podprzestrzeni zgodnie do odpowiedniego modelu dziedziny. W przestrzeni „Commands” także dokonano dodatkowego rozdzielenia według typu operacji celowej („Create”, „Update”, lub „Delete”).

W warstwie także zdefiniowano jednostki bezpośrednio zaangażowane w proces otrzymywania zapytań na dane oraz wysyłania odpowiedzi na nich. Za pomocą tych obiektów kształcono interfejs programowy do korzystania przez pozostałe serwisy. W celu ich definicji używano narzędzia frameworku ASP do implementacji API zgodnie z zasadami REST, z wykorzystaniem protokołu HTTP do wymiany danymi. Jednostki biorące udział w procesie wymiany danymi są reprezentowane przez klasy typu Controller, dopasowane osobiście do każdego modelu encji. Wewnątrz tych klas odbywa się kierowanie metodami „GET”, „POST”, „PUT”, „DELETE”, odpowiednio do każdej z podstawowych operacji protokołu HTTP, a także ich odmianami, specyficznymi dla różnych modeli.

Warto zaznaczyć, że przedstawianie interfejsu programowego do transferowania danych zostało jedyną funkcją klas typu Controller. Logika walidacji i przetwarzania danych została wyniesiona do metod określonych przez zewnętrzne klasy typu Handler. Do przekazywania danych pomiędzy kontrolerami a handlerami skorzystano z wzorca projektowego „Mediator”, polegającego na założeniu pewnej scentralizowanej jednostki, odgrywającej rolę pośrednika w wymianie danymi pomiędzy kilkoma niezależnymi warstwami oprogramowania. Gotowe rozwiązanie zaprezentowano pakietem „MediatR”, z którego także pobrano narzędzia do napisania handlerów przetwarzających zapytania.

W ten sposób obiekty DTO, otrzymywane przez kontrolery jako część zapytania, zostają przekształcone do obiektów CQRS, po czym zostają przekazane do handlerów odpowiedzialnych za określone operacje. Po przetworzeniu przez handlersy wyniki zostają zwracane w postaci obiektów DTO, które następnie stanowią odpowiedź metod kontrolerów wraz z odpowiednim statusem potwierdzającym wykonanie zapytania HTTP. Przy wystąpieniu problemów na poziomie przetwarzania przez handlersy, kontrolerami odbywa się zwracanie statusów HTTP oznaczających konkretne błędy.

W klasach handlerów wprowadzono zależność od obiektów implementujących wzorzec „Repository”, za pomocą których przekazują się polecenia na wywołania operacji nad obiektami

bazy danych. W trakcie działania metod handlerów dokonywano sprawdzenia pewnych warunków, na podstawie których metoda potrafi zgłaszać o błędach przez mechanizm zgłaszania wyjątków. W przypadku powodzenia operacji metody generują i zwracają obiekty DTO.

Operowanie w metodach obiektów „Repository” odbywa się wyłącznie na obiektach modeli przestrzeni „Domain”, z kolei handlery przyjmują i zwracają obiekty należące innym klasom, co wymaga przeprowadzenia konwersji danych. Aby scentralizować proces konwersji obiektów do innych typów, także używano implementacji wzorca „Mediator”, przedstawionego przez pakiet „AutoMapper”. Opis szczegółów konwersji wyniesiono do logicznej przestrzeni „Mappings”.

Ponadto klasy kontrolerów skonfigurowano w taki sposób, aby dostęp do ich metod wymagał autoryzacji. Dalsza kontrola uprawnień odbywa się w serwisie „GatewayAPI”, który pełni rolę nadzorcy logiki dostępu.

W taki sposób, opisane podejścia projektowe wyznaczają struktury serwisów „Cinema Data Service” oraz „User Data Service”. Wydzielenie logicznych części sprzyja kontrolowaniu procesu implementacji modułów zapewniających dane dziedziny projektowej wraz z przebiegiem operacji nad nimi przewidywane logiką biznesową.

### **Organizacja serwisu z dostarczenia danych multimedialnych**

Zadaniem serwisu „Image Service” jest zapewnienie przechowywania, przetwarzania i pobierania danych w postaci obrazów. Do przechowywania obrazów wykorzystano usługę „Blob Storage” z platformy Azure. Komunikacja z usługą odbywa się przez pakiet „Azure.Storage.Blobs”.

Serwis został oparty na tych samych zasadach, co serwisy „Cinema Data Service” oraz „User Data Service”, jednak w tym przypadku skorzystano z wersji uproszczonej. Decyzja o uproszczeniu została podjęta ze względu na specyfikę danych, którymi się operuje w kontekście tego modułu. Według tego dodawanie warstwy dziedziny „Domain” jest zbędne, z tym także implementacja wzorca CQRS – w przypadku danego serwisu operacje są jednolite, dlatego modele wszystkich zapytań są połączone we wspólnej przestrzeni „Requests”. W skutku zachowano dopiero podział na warstwy „Infrastructure” oraz „Api”.

Warstwa „Infrastructure” definiuje klasy DTO, modelujące zapytania z dodawania, zamieniania, usuwania oraz pobierania obiektów obrazów. Ponadto w tej warstwie wyznaczono

klasę implementującą wzorzec projektowy „Repository” jako warstwę abstrakcji nad obiektem klasy odpowiedzialnej za operacje na danych w chmurze.

Obiekty klas DTO dostarczają dane o zdjęciach w postaci wierszy. Przed przesłaniem zdjęcia, znajdującego się początkowo w postaci potoku pliku (obiekту klasy Stream), odbywa się jego konwersja do postaci bardziej nadającej się do transferowania – w tym celu wybrano stosować format Base64. Konwersje obiektów przeprowadzano z wykorzystaniem klasy Convert ze standardowej przestrzeni języka C#. W ten sposób obowiązki serwisu obejmują dekodowanie nadchodzących wierszy do formatu binarnego przed przekazaniem plików do zapisania w chmurze.

W celu dostosowania się do wymagań serwisów z prezentacji danych, które mogą potrzebować obrazów o różnych skalach i rozmiarach, serwis generuje i przechowuje kilka kopii pojedynczego obrazu jednocześnie. Dla określenia wymaganych rozmiarów obrazu wprowadzono typ przeliczeniowy ImageSize o wartościach „Tiny”, „Small”, „Medium”, „Big”, „Large”. Dane wartości pełnią rolę flag, co umożliwia ich dowolne łączenie między sobą. Odpowiednie pole reprezentujące rozmiar obrazu dodano do wszystkich modeli zapytań. Z tego powodu typ flag współdzielili z nimi jedną przestrzeń. Dla uproszczenia wszystkie kopie jednego zdjęcia mają tę samą nazwę, jednak ich lokalizacja różni się w kontenerze chmury. Odpowiednio każdy typ obrazu posiada przypisany osobny folder.

Do tworzenia kopii obrazów o różnych rozmiarach zastosowano skalowanie, realizowane za pomocą narzędzi przedstawianych przez pakiet „Magick.NET”. Wykorzystano metodę skalowania interpolacyjnego z użyciem algorytmu interpolacji dwuliniowej, który charakteryzuje się szybkością przetwarzania.

Dokładne wartości parametrów, decydujące o przynależności obrazów do poszczególnych kategorii typu przeliczeniowego ImageSize nie są bezpośrednio zapisane w kodzie, lecz zostają wyznaczone w pliku konfiguracyjnym „appsettings.json”. W tym pliku także określono parametr maksymalnego rozmiaru pliku oraz ścieżki do folderów wszystkich typów obrazów w kontenerze chmurowym. Rozmiar obrazu jest wyznaczany na podstawie długości jego krótszego boku, wartości graniczne którego odpowiadające każdej z kategorii także są definiowane w pliku konfiguracyjnym. Przy inicjalizacji serwisu informacje z pliku konfiguracyjnego są pobierane i przechowywane w obiekcie programowym w postaci słownika.

Interfejs programowy warstwy „Api” tego serwisu zaimplementowano przez jedną klasę

typu Controller. Logika przekształcenia plików obrazów w tym serwisie także została wyniesiona do metod handlerów, w których odbywa się sprawdzenie obecności określonych flag, skalowanie, generowanie ścieżek kontenera do obrazów, wywoływanie metod do zapisania danych według tych ścieżek w chmurze oraz sprawdzanie wyników wraz ze zgłaszaniem problemów w postaci wyjątków. W przypadku powodzenia handler generuje obiekt odpowiedzi, który zawiera nazwę zdjęcia, wartość z przeliczeniem obrobionych rozmiarów, oraz słownik w formie klucz-wartość, gdzie kluczem występuje rozmiar typu ImageSize, a wartością – wiersz ze ścieżką URL lokalizacji zdjęcia odpowiedniego rozmiaru. Aczkolwiek same obrazy nie są przesyłane bezpośrednio w odpowiedzi, zwracanych w odpowiedzi ścieżek jest wystarczająco dla wyświetlania zdjęć na stronach prezentacyjnych. Podobnie do wcześniej opisanych serwisów, metody kontrolera wymagają dostęp autoryzowany.

### **Organizacja serwisu z przekazywania danych według logiki biznesowej**

Następną jednostką jest serwis „Gateway API”, który pełni kluczową rolę w wymianie danymi pomiędzy serwisami. Ten serwis został zaimplementowany przy użyciu uproszczonych decyzji architektonicznych w porównaniu do wcześniej opisanych serwisów, jednak cechuje się metodami o bardziej złożonej logice przetwarzania danych.

Serwis jest podzielony na dwie główne warstwy – warstwę „Infrastructure” oraz „Api”. Warstwa „Infrastructure” zawiera przestrzeń „Services”, w której wyznaczono klasy modelujące klientów HTTP. Klasy te definiują metody do obsługi połączeń z pozostałymi serwisami architektury, co umożliwia przesyłanie danych. Zestaw zdefiniowanych metod każdej klasy odpowiada interfejsowi programowemu określonego serwisu, czym odbywa się modelowanie jego zewnętrznej struktury. W ten sposób klienci HTTP służą do reprezentacji samych serwisów, obecnych w architekturze.

Implementacje HTTP klientów zostały opracowane dopiero dla serwisów źródłowych, którymi występują „Cinema Data Service”, „User Data Service” oraz „Image Service”. Dla uproszczenia wewnętrznych struktur klas zdecydowano o dostosowaniu klientów do interfejsów poszczególnych encji dziedziny – w ten sposób klient serwisu „Cinema Data” rozbija się na „Cinema HTTP Client”, „Person HTTP Client” oraz „Studio HTTP Client”. Tym sposobem wprowadzono również poziom abstrakcji – nie określa się wprost, w jaki sposób dane encji projektowych są rozdzielone pomiędzy serwisy.

W warstwie „Api” zorganizowano klasy kontrolery zarządzające danymi poszczególnych encji systemu: „Cinema”, „Person”, „Studio” oraz „User”. Prawa logiki biznesowej zostały wbudowane bezpośrednio do metod interfejsu programowego, co zapewnia wprowadzenie kontroli logiki przy wymianie danymi. Z tego powodu zrezygnowano z implementacji dodatkowych klas handlerów, które odpowiadałyby za wyniesienie logiki. W ten sposób przestrzeń klas kontrolerów jest jedyną przestrzenią logiczną warstwy „Api” tego serwisu. Zapytania i odpowiedzi interfejsów zostają formowane z użyciem modeli DTO przedstawionych w ogólnej przestrzeni Shared.

Jeżeli metody interfejsów wcześniejszych serwisów operują wyłącznie na obiektach pojedynczego pojęcia projektowego, serwis „Gateway” zarządza procedurami, obejmującymi kilka pojęć jednocześnie. Łączenie odbywa się przy obsługiwaniu obu przedstawionych typów zapytań, czyli zarówno odczytujących i modyfikujących dane. W pierwszym przypadku obiekt transferowania danych jest tworzony na podstawie danych pochodzących z różnych źródeł, natomiast w drugim przypadku celowa modyfikacja pojedynczego obiektu może powodować poboczne działania na kilku obiektach pochodzących z różnych dziedzin jednocześnie.

Serwis także zajmuje się ograniczeniem dostępu do określonych zasobów. Jego zadanie polega na autoryzacji klienta, co przekierowuje niezalogowanego użytkownika na stronę z logowania. Przedstawione informacje autoryzacyjne wkluczają się jako część zapytań klienta do danych. Przy otrzymaniu odmowy dostępu odbywa się zwrócenie błędu o odpowiedniej treści.

Jednym z działań na poziomie procedury odczytu danych („GET”) jest formowanie dodatkowego zapytania w celu zwrócenia ścieżki do obrazu obiektu, o ile odpowiednie pole zawierające nazwę tego obrazu nie jest puste. Innym działaniem jest uzupełnienie niedostających informacji, które w głównym obiekcie występują jedynie jako identyfikatory. Wtedy zostaje formułowane dodatkowe zapytanie na podstawie tego identyfikatora w celu pobrania informacji o powiązonym obiekcie. Takie podejście jest stosowane przy obsługiwaniu zapytania na listy etykietowanych użytkownikiem filmów. W jego kontekście najpierw wysyłane jest zapytanie o listę identyfikatorów filmów za wydanym identyfikatorem użytkownika, a następnie formułowane zapytanie do metody API „Cinema” na szczegółowe informacje o wszystkich filmach odpowiadających zwróconej liście identyfikatorów. Ostatecznie ta lista zostaje zwrócona jako odpowiedź na oryginalne zapytanie.

Innym rodzajem bardziej złożonych działań są procedury z modyfikacji, które wkluczają

operacje dodawania, edytowania lub usuwania obiektów. Dla listy etykietowanych filmów lub ocen filmów użytkownika te działania obejmują jedynie sprawdzenie obecności obiektów za wskazanymi identyfikatorami oraz edytowanie samej informacji, reprezentowanego przez jedno zapytanie ze sprawdzeniem wyniku. W przypadku tych elementów, które zawierają informacje o powiązanych obiektach jako część własnych danych, oprócz wcześniejszych weryfikacji konieczne jest podejmowanie odpowiednich działań zapewniających logiczną integralności danych. Taką procedurą występuje aktualizacja odpowiednich list – jeżeli obiekt zostaje dodany do listy powiązanego elementu innego typu, ten obiekt musi również dodać odniesienie do bieżącego elementu poprzez umieszczenie jego we własnej liście. Analogicznie, usuwanie powiązanych obiektów z listy wymaga spójnej aktualizacji. W przypadku całkowitego usuwania obiektu, jego odniesienia muszą także zostać usunięte ze wszystkich list elementów powiązanych. Dotyczy to również edytowania – zmiany określonych pól obiektu zostają odzwierciedlone we wszystkich listach, w których dany obiekt występuje.

Przykładem takich operacji są działania związane z encją „Cinema”. Podczas edytowania obiektu tej encji, zapis o nim także zostaje aktualizowany dla encji „Person” i „Studio”, w których dany obiekt prezentowany jest w listach pola „Filmography”. Ponieważ te pojęcia są ze sobą powiązane, nie potrzebuje się wyszukiwania konkretnych obiektów, listy filmografii (pola „Filmography”) których zawierają potoczny film (obiekt „Cinema”) – odpowiednie identyfikatory takich obiektów są już zapisane w odpowiednich listach „Starrings” oraz „ProductionStudios” samego obiektu filmu. Dzięki temu możliwa jest bezpośrednia aktualizacja konkretnego zapisu z listy bez przeprowadzenia wyszukiwania. Przy obsługiwaniu zapytania o usunięcie filmu sprawdza się, czy element filmu zawiera odniesienia do obiektów innych encji. W przypadku zawierania, jego wpis usuwa się z list „Filmography” odpowiednich obiektów zanim sam film zostaje całkowicie usuwany.

W analogiczny sposób przebiegają procedury dotyczące encji „Person” i „Studio” z odwróceniem ról obiektów docelowych i zapisów. Podczas operacji na obiektach osób (obiekty encji „Person”) zmiany są odzwierciedlane w listach „Starrings”, natomiast przy działaniach na wytwórniach filmowych (obiekty encji „Studio”) zmiany można prześledzić w listach „ProductionStudios”.

Dodatkowe aktualizacje obiektu „Cinema” występują podczas zmiany lub dodawania oceny użytkownika dla filmu. Wtedy przed zapisaniem oceny użytkownika najpierw odbywa się

aktualizacja oceny bezpośrednio dla filmu.

W ten sposób danym serwisem zorganizowano warstwę logiki całej aplikacji, której prawa zostają implementowane w scentralizowanej jednostce pośredniczącej.

### **Organizacja serwisów z prezentacji widoków**

Niniejsza część opisuje serwisy, w których zaimplementowano widoki oraz powiązaną z nimi funkcjonalność, mającą na celu obsługiwanie interaktywności tych widoków. Przedstawiona zostaje zarówno struktura samego serwisu, jak i widoków, wraz z opisem ich przeznaczenia.

Dla formowania komponentów stron wykorzystano statyczną bibliotekę CSS „bootstrap.css”. Niektóre style z tej biblioteki zostały nadpisane we własnym pliku CSS, a dodatkowo z tym wprowadzono własne style prezentacji komponentów.

Wygląd widoków jest zależny od określonych warunków, dlatego zdecydowano się na mechanizm generowania widoków w celu dynamicznego napełnienia stron podczas ich ładowania. Jako mechanizm generowania widoków wybrano natywną dla frameworku .NET technologię Razor Pages, która umożliwia renderowanie znaczników HTML przy użyciu języka C# do sterowania logiką renderowania. Mechanizm ten wspiera generowanie jedynie statycznych stron, natomiast dynamiczne zmiany ich napełnienia zostają obsługiwane przez język JavaScript. Do stosowanych bibliotek języka JavaScript wchodzi „bootstrap.js”, „jquery.js”, oraz inne moduły rozszerzające ich funkcjonalność. Oprócz tego napisano własne skrypty, dostosowane do specyfiki każdej z poszczególnych stron.

### ***Struktura serwisów***

Serwis z organizacji widoków jest podzielony na warstwy „Api” oraz „Infrastructure”, przy czym główne funkcjonalności znajdują się w warstwie „Api”.

W warstwie „Infrastructure” wyznaczono jedną logiczną przestrzeń „Services” która zawiera definicję klienta odpowiedzialnego za wymianę danych. Klient ten służy do reprezentacji serwisu „GatewayAPI”, zdefiniowano jego w sposób analogiczny do wcześniej opisanych klientów HTTP. Za pośrednictwem tego klienta realizowane są zapytania o dane oraz polecenia dotyczące ich operacji, inicjowane bezpośrednio przez działania użytkownika na stronach. Klient udostępnia uogólniony interfejs, niezależnie od konkretnych encji poddziedzin. Polecenia przy tym różnią się w zależności od docelowej funkcjonalności serwisu widoków.

Organizacja warstwy „Api” opiera się na wzorcu projektowym „MVVM” („Model-View-



View Model”), który jest zbliżony do wzorca „MVC” z tą różnicą, że jednostka „View Model”, która oddziela logikę biznesową od interfejsu użytkownika i jest odpowiedzialna za dostarczanie danych, koncentruje się wyłącznie na logice poszczególnych stron, a nie na logice operowanych danych [6]. Modele widoków typu „View Model” znajdują się we wspólnej z widokami warstwy „View” przestrzeni logicznej. Odpowiednio do tego podejścia w warstwie „Api” powstały dwie przestrzeni nazw: „Views” oraz „Models”. Część „Views” zawiera strony o rozszerzeniu „.cshtml” utworzone przy użyciu technologii Razor, a także modele widoków zarządzające stronami i realizujące zapytania o dane poprzez klienta HTTP serwisu „GatewayAPI”. Część „Models” obejmuje modele przeznaczone do reprezentacji danych, dostosowane dla kategorii prezentowanych na stronach informacji. Po otrzymaniu odpowiedzi w postaci obiektów modeli transferowania danych, zostają one konwertowane na odpowiednie obiekty modeli widoków, które następnie wykorzystywane są do generowania stron.

Istotnym elementem struktury stanowi zawartość folderu „wwwroot”, w którym zostały umieszczone zasoby statyczne. Obejmują one biblioteki języka JavaScript, w tym „jquery.js”, „jquery.validate.js”, „bootstrap.js” oraz własny moduł funkcji „utils.js” wykorzystywany w plikach skryptów specjalizowanych. Ponadto folder zasobów zawiera bibliotekę stylów „bootstrap.css” wraz z plikami własnych stylów. Przechowuje także inne elementy statyczne – są to ikony w formacie „.svg” oraz obrazy.

### ***Komponenty stron i ich logika***

Część logiczna „Views” zawiera zarówno pliki widoków jak i wykorzystywane przez nich modele obiektów. W celu uproszczenia i strukturyzacji plików ze znacznikami HTML, niektóre strony zostały podzielone na odrębne komponenty – widoki częściowe – które są automatycznie składane w jeden wspólny widok podczas procesu renderowania.

Strony serwisów można podzielić na kategorie zgodnie z ich przeznaczeniem, są to np. strony prezentujące wiele obiektów określonej encji, strony z detalicznym widokiem pojedynczego obiektu oraz inne typy stron.

Istotnym komponentem odróżniającym się od pozostałych jest „Layout”, którym jest widok częściowy odpowiedzialny za wyświetlenie ramki widocznej na wszystkich stronach serwisu. Ramka zawiera górną nawigację z nazwą serwisu w lewym rogu, przycisk „Search” i pole wyszukiwania po środku oraz przycisk „Account” w prawym rogu. Po naciśnięciu na ostatni wyświetlana zostaje lista dostępnych operacji związanych z kontem użytkownika, zależnie od jego

statusu („zalogowany” lub „niezalogowany”) oraz roli w serwisie.

Strony prezentujące wiele obiektów zostają generowane osobno dla każdej encji encyklopedii –filmów, osób oraz wytwórni filmowych. Widoki prezentują obiekty na stronie w układzie siatki kartek, składającej się z kilku kolumn i wierszy. Każdy element siatki zawiera skrócone informacje oraz pomniejszony obraz. Elementy zawierają ukryte linki umożliwiające przejście do szczegółowego widoku obiektu.

Dodatkowo strony te posiadają opcje sortowania oraz filtrowania, umieszczone po prawej stronie od siatki elementów. Na dole każdej strony znajduje się numeracja, wskazująca numer potocznej strony oraz numery sąsiadujących z nią stron.

Nawigacja do tych widoków może odbywać się poprzez pole wyszukiwania, które umożliwia znalezienie obiektów za ich nazwą. Obok pola „Search” znajduje się lista do wybrania celowego typu obiektów do wyszukiwania („Cinema”, „Person”, lub „Studio”). Wprowadzenie pustej wartości powoduje wyszukiwanie wszystkich obiektów wybranego typu w kolejności, wyznaczanej na poziomie bazy danych.

Szczegółowe widoki obiektów są definiowane osobno dla każdej encji projektowej. Mają one bardziej złożoną strukturę, nadając możliwość wykonywania operacji różnego typu o ile użytkownik jest zalogowany i posiada na to odpowiednie prawa. Decyzja o przyznaniu praw dostępu podejmowana jest podczas renderowania widoków. Jeżeli użytkownik nie jest zalogowany lub posiada status zwykłego użytkownika, przedstawia się jemu możliwość w większości jedynie do przeglądania dostępnych na stronach danych. Natomiast w przypadku, gdy użytkownik posiada rolę uprzywilejowaną, odpowiednie widoki udostępniają opcje do edytowania lub usuwania strony opisującej obiekt. Dodatkowo w opcjach profilu „Account” w prawym górnym rogu nawigacji pojawia się opcja „Create” dodawania nowej strony z wybraniem typu obiektu, dla którego strona zostanie utworzona.

W przypadku braku odpowiednich praw opcje edytowania stron nie są ukrywane od użytkownika za pomocą stylizacji – dla użytkowników o mniej przywilejowanych rolach nie zostają one renderowane w ogóle. Wyjątkiem od tej reguły jest szczegółowy widok profilu osoby – największe przywileje przysługują właścicielowi konta tego profilu, niezależnie od jego roli w serwisie. Z kolei użytkownicy o najwyższych uprawnieniach mogą jedynie zmieniać role lub usuwać konta innych użytkowników. Także wyjątek dotyczy widoku list etykietowanych użytkownikiem filmów – możliwości dodania nowych zapisów, zmiany etykiet oraz ich usuwania

udostępniają się wyłącznie dla właściciela tych list.

Widoki szczegółowe zawierają obraz reprezentujący opisywany obiekt (plakat, zdjęcie, lub logotyp), podstawowe informacje, umieszczane po jego prawej stronie oraz sekcję na dole strony prezentującą powiązane wpisy. Wpisy te są ułożone w zestawach siatek kart, między którymi można się przełączać za pomocą strzałek znajdujących po obu stronach bloku tej sekcji. Elementy kart zawierają skrócone informacje i miniatury obrazów, a także ukryte linki, prowadzące do odpowiedniego szczegółowego widoku, w którym potoczny obiekt prezentuje się w dolnej sekcji jako element powiązany.

Dodatkowe informacje, dotyczące dopiero filmów, obejmują średnią ocenę użytkowników, zwizualizowaną w formie dziesięciu gwiazdek umieszczonych w górnej części strony. Ponadto pod obrazem plakatu filmu znajduje się przycisk dla dodawania filmu do listy „ulubionych”, obok którego umieszczono rozwijaną listę opcji umożliwiających dodawanie filmu do pozostałych list użytkownika.

Szczegółowe widoki obiektów integrują kilka widoków częściowych, takich jak formularze dla dodawania, lub edytowania obiektów, a także widoki przedstawiające obiekty powiązane (zawarte w listach „Starrings”, „ProductionStudios” oraz „Filmography”). Także zostały dodane widoki częściowe pozwalające dodawanie lub edytowanie wpisów z tych list. Dodawanie nowych zapisów odbywa się poprzez wyszukiwanie odpowiedniej encji według nazwy, wybór obiektu z ograniczonej listy znalezionych opcji oraz zaznaczenie dodatkowych informacji o wpisie.

Operacja przesyłania formularza o wypełnionych polach wywołuje zdarzenie, które przekazuje dane formularza do określonej metody modelu widoku. Następnie metoda ta wysyła zapytanie za pomocą klienta HTTP w celu przetworzenia przekazanych informacji. Po otrzymaniu od serwisu odpowiedzi z wynikiem przetwarzania, metoda przekazuje ją do kontekstu potocznej strony. Skrypty języka JavaScript pobierają końcowy wynik tej metody i wykorzystują do aktualizacji stanu strony bez jej pełnego przeładowania, lub do generowania prostego powiadomienia o wykonaniu operacji.

Przykłady zdarzeń wykonanych na stronach widoków szczegółowych obejmują dodawanie strony nowego obiektu, edytowanie informacji na stronie istniejącego obiektu oraz usuwanie strony obiektu. Podczas dodawania nowego obiektu strona tworzenia nadaje możliwość do wpisania wszystkich typów informacji, które po potwierdzeniu i walidacji zostają zbierane i

przesyłane razem w jednym obiekcie. Proces usuwania strony jest realizowany przez pojedynczą operację, która przedstawia się dla użytkownika w jakości wcześniej określonej opcji. Z kolei edycja informacji na stronie obiektu, w odróżnieniu od przypadku dodawania, ma charakter bardziej atomowy. Dane do zmiany są podzielone na konkretne kategorie: informacje główne, obraz, informacje o wpisach powiązanych. Do informacji głównych należą nazwa obiektu, jego opis, kategorie, do których obiekt podpada, inne. Obraz polega modyfikacji przez oddzielny przycisk z wybraniem pliku nowego obrazu. Wpisy o powiązanych obiektach zarządzane są przez należące odpowiadające im opcje usuwania lub edycji, dodawanie nowego wpisu reprezentowano oddzielnym przyciskiem. Na stronie profilu użytkownika dostępna jest edycja wyłącznie informacji podstawowych, takich jak imię, opcjonalne do wypełnienia pole daty urodzenia oraz obraz profilowy.

Na stronie filmu (widoku obiektu „Cinema”) zalogowani użytkownicy mają dostęp do dodatkowych operacji, takich jak zmiana oceny filmu oraz dodawanie wpisu o nim do etykietowanej listy. Zmiana oceny jest także zwizualizowana za pomocą ikon w kształcie gwiazdek – aby dokonać zmiany użytkownik najpierw klika na wartość oceny średniej, po czym wybiera jedną z dziesięciu dostępnych gwiazdek z rozwijanej listy. Po dokonaniu wyboru oceny zostaje wywołane zdarzenie strony z przesłaniem informacji o zaktualizowanej ocenie. Dodanie filmu do listy odbywa się po kliknięciu wybranej opcji z zestawu ikon pod plakatem filmu. Przy najechaniu kursorem nad ikonką wyświetlana jest nazwa tekstowa odpowiedniej etykiety. Kliknięcie na ikonkę inicjuje zdarzenie, analogiczne zdarzeniu zmiany oceny użytkownika.

Do kategorii pozostałych stron należą się listy etykietowanych filmów oraz strony związane z procesem uwierzytelniania, zarządzane przez jednostkę „Access Service”, kluczowymi z których są strony logowania, rejestracji oraz wylogowania użytkownika. Listy etykietowanych filmów są podzielone na wkładki kolorowane według rodzaju etykiety, między którymi możliwe jest przełączenie przez kliknięcie na odpowiednią ikonkę. Obiekty wpisów mogą zostać usunięte poprzez kliknięcie na odpowiednią opcję. Dodawanie filmów do listy jest możliwe wyłącznie na odpowiednich stronach widoków szczegółowych poprzez opisany wcześniej zestaw ikon.

Wydarzenia stron uwierzytelniania są bezpośrednio powiązane z mechanizmem autoryzacji, szczegóły którego są opisane w następnej części.

## Organizacja serwisu z uwierzytelnienia użytkowników

Celem jednostki „Access Service” jest uwierzytelnianie użytkowników, przechowywanie ich danych do logowania oraz realizacja mechanizmu autoryzacji. Serwis ten został podzielony na trzy warstwy: „Domain”, „Infrastructure” oraz „Api”.

W warstwie „Domain” zawiera jedynie wyznaczenie klasy „AccesProfileUser” modelujące profil użytkownika. Klasa ta dziedziczy właściwości klasy IdentityUser z frameworku .NET, lecz różni się od klasy modelującej użytkownika, zdefiniowanej w serwisie „User Data”. Obiekty tej klasy przechowują podstawowe informacje uwierzytelniające podczas gdy pozostałe dane dotyczą interakcji użytkownika z aplikacją. Kluczowym elementem profilu użytkownika jest jego rola, która jest sprawdzana podczas interakcji z serwisami.

Warstwa „Infrastructure” definiuje sposób przechowywania informacji uwierzytelniających użytkownika. Podobnie jak w przypadku innych serwisów, zawiera ona kontekst klasy umożliwiający bezpośrednie operowanie na bazie danych oraz konfigurację obiektów w tej bazie. Informacji uwierzytelniające podlegają przechowywaniu w wyglądzie strukturyzowanym przez system Microsoft SQL Server.

Warstwa „Api” obejmuje dwa typy jednostek do przesyłania danych. Pierwszym z nich są to modele stron, odpowiedzialne za generowanie widoków uwierzytelniających z realizacją procesu logowania. Drugim typem są klasy typu Controller, które obsługują proces autoryzacji z przyznawaniem użytkownikowi dostępu do zasobów po dokonaniu uwierzytelnienia przez proces rejestracji lub logowania.

Wybrany protokołem autoryzacji jest OIDC („OpenID Connect”), którego wymiana informacji odbywa się zgodnie z zasadami REST. OIDC opiera się na protokole OAuth, który definiuje podstawowe mechanizmy interakcji między serwerem autoryzacyjnym a jego klientem.

Końcowym wynikiem autoryzacji jest tzw. token dostępu (z ang. „Access Token”), który jest ciągiem znaków pełniącym rolę poświadczenia, umożliwiającego uzyskanie dostępu do chronionych zasobów użytkownika. Tokeny reprezentują określają zakresy dostępu, przyznawane przez właściciela zasobu i egzekwowane przez serwer autoryzacji [7, s. 10]. W protokole OIDC reprezentują się one w formacie JWT („JSON Web Token”), który składa się z zestawu oświadczeń w postaci obiektu JSON, zakodowanego przy użyciu określonego algorytmu, co umożliwia ich cyfrowe podpisywanie [8, s. 4]. Oświadczeniem w tym kontekście jest jednostka informacji podana na temat podmiotu, reprezentowana jako para klucz-wartość. Standardowymi

oświadczeniami protokołu OIDC są imię, nazwisko, adres e-mail, inne. Wykorzystywanymi oświadczeniami użytkownika danej aplikacji są jego identyfikator w serwisie, czyli identyfikator zapisu profilu w bazie „User Data Service”, nazwa jego profilu oraz przypisana jemu rola.

Podstawowy protokół autoryzacji obejmuje kilka etapów, które polegają na wymianie określonych danych w celu uzyskania tokena dostępu. Dane przesyłane przez klienta do procesu autoryzacji ze strony klienta obejmują jego identyfikator, zakres żądanych zasobów, adres URI do przekierowania po zakończeniu logowania oraz inne informacje. Protokół opisuje kilka metod dostarczania tokena. W przypadku jawnego mechanizmu autoryzacji przedstawiane przez klienta dane są zawarte bezpośrednio w zapytaniu, a token dostępu zwracany jest jako część URL przekierowania. Dla zapewnienia interaktywności oraz bezpieczeństwa procesu uwierzytelniania używana jest metoda kodu autoryzacyjnego. Ta metoda wyznacza przeprowadzenie dodatkowego zapytania na kod autoryzacyjny, którym jednocześnie przekierowuje użytkownika na stronę logowania. Po zakończeniu logowania generowany jest kod, który następnie zostaje wymieniony token dostępu za pomocą odrębnego zapytania [7, s. 24]. Po otrzymaniu tokena klient dokonuje jego walidację, wykorzystując podpis dostarczony przez serwer autoryzacyjny, po czym deszyfruje zawierające w nim informacje.

W części klienckiej, dokonującej zapytania o kod autoryzacyjny oraz token dostępu z walidacją oraz przechowywaniu ostatniego w HTTP cookie, wykorzystywany jest standardowy handler OIDC, implementację którego przedstawia framework ASP. Jego funkcjonalność integruje się z metodami serwisu „Gateway” w celu uzyskania tokena dostępu, który następnie jest dołączany do zapytań o dane. Serwisy odpowiedzialne za dostarczenie danych dokonują walidację przekazanego tokena na obecność wymaganych pól oraz ich wartości. Proces ten realizowany jest za pomocą handlera JWT, którego implementacja również pochodzi z ASP frameworku.

W celu dokładnego zbadania mechanizmu autoryzacji, zamiast rzeczywistego serwera autoryzacyjnego zastosowano własne rozwiązanie modelujące, implementacja którego znajduje się w warstwie „Api” w postaci klasy typu Controller, która udostępnia standardowe punkty końcowe zapytań protokołu OAuth.

W ten sposób serwis zapewnia informacje uwierzytelniające użytkownika, autoryzuje użytkownika przez obsługiwanie zapytań serwisu „Gateway”, a także zarządza informacjami ubezpieczającymi API innych serwisów z nadaniem tokenów umożliwiających dostęp do zapytań wymagających dostęp autoryzowany.

## Wyniki

### Testowanie podstawowych zapytań

Testowanie rozpoczęto z poziomu serwisów bezpośrednio zarządzających danymi. Testowaniu poddano metody HTTP różnego typu. Dla przeprowadzenia testowania został podłączony mechanizm Swagger, który umożliwia testowanie interaktywne.

Testowanie API danego typu serwisu „Cinema Data” rozpatrzono na przykładzie obiektów filmu. Na rysunku 1 przedstawiono początkowe zapytanie do tworzenia obiektu.



The screenshot displays the Swagger UI for a POST endpoint `/api/cinemas`. The request body is a JSON object representing a cinema entry. The response body, returned with a 200 status code, is a JSON object containing the created cinema's details, including its ID, name, release date, genres, language, rating, and description.

```

POST /api/cinemas

Parameters
No parameters

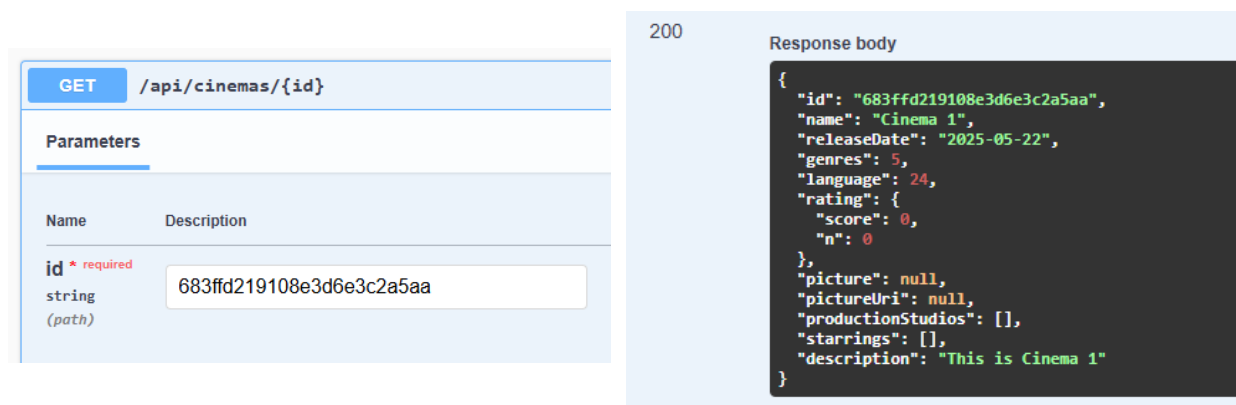
Request body
{
  "name": "Cinema 1",
  "releaseDate": "2025-05-22",
  "genres": 5,
  "language": 24,
  "picture": null,
  "productionStudios": null,
  "starrings": null,
  "description": "This is Cinema 1"
}

200
Response body
{
  "id": "683ffd219108e3d6e3c2a5aa",
  "name": "Cinema 1",
  "releaseDate": "2025-05-22",
  "genres": 5,
  "language": 24,
  "rating": {
    "score": 0,
    "n": 0
  },
  "picture": null,
  "pictureUri": null,
  "productionStudios": [],
  "starrings": [],
  "description": "This is Cinema 1"
}

```

Rysunek 1 – Zapytanie oraz wyniki wykonania zapytania POST

Na rysunku 2 przedstawiono zapytanie do pobrania obiektu za jego identyfikatorem, które jednocześnie sprawdza porządnosc wykonania poprzedniego zapytania POST.



The screenshot displays the Swagger UI for a GET endpoint `/api/cinemas/{id}`. The path parameter `id` is required and is a string. The response body, returned with a 200 status code, is a JSON object containing the details of the cinema identified by the provided ID.

```

GET /api/cinemas/{id}

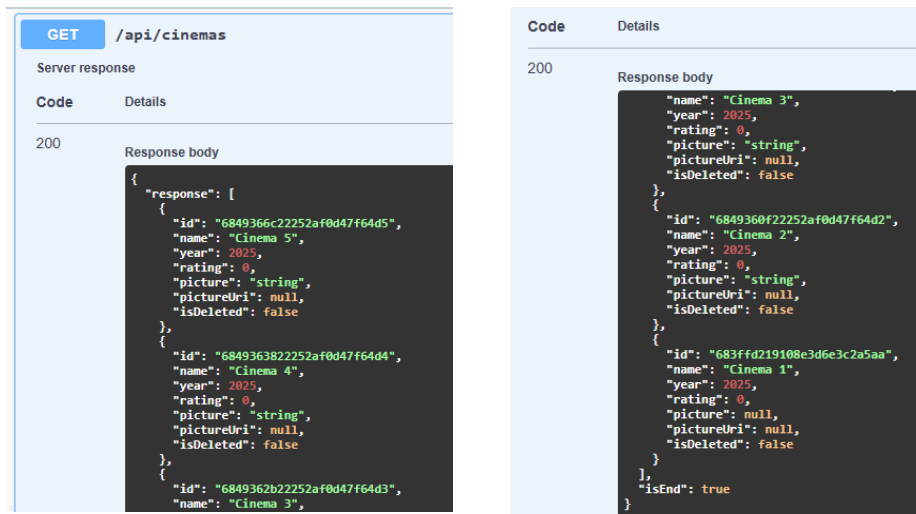
Parameters
Name      Description
id * required
string
(path)    683ffd219108e3d6e3c2a5aa

200
Response body
{
  "id": "683ffd219108e3d6e3c2a5aa",
  "name": "Cinema 1",
  "releaseDate": "2025-05-22",
  "genres": 5,
  "language": 24,
  "rating": {
    "score": 0,
    "n": 0
  },
  "picture": null,
  "pictureUri": null,
  "productionStudios": [],
  "starrings": [],
  "description": "This is Cinema 1"
}

```

Rysunek 2 – Zapytanie oraz wyniki wykonania zapytania GET dla pojedynczego obiektu

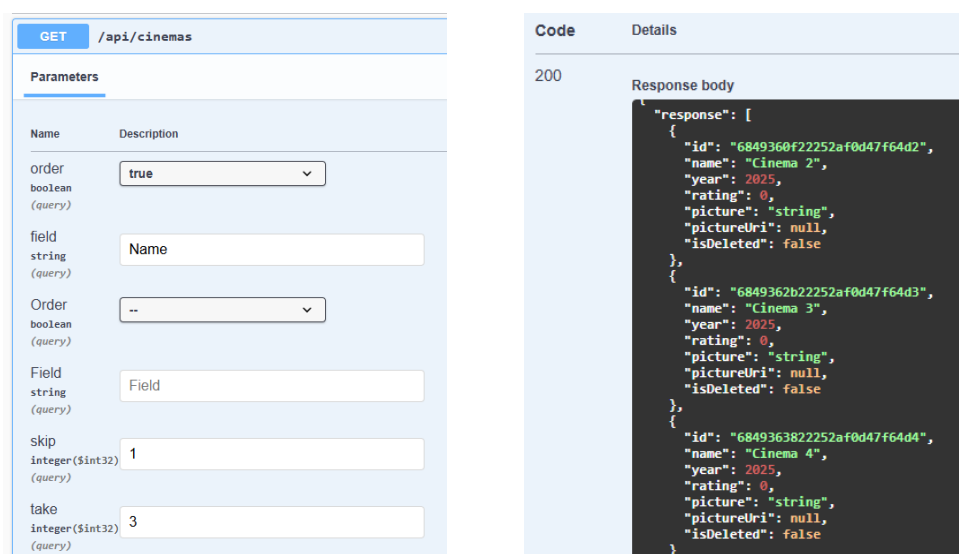
Następnie w podobny sposób dodano i pobrano kilka innych obiektów, aby przetestować pobieranie grupy obiektów. Testowanie przedstawione jest rysunkiem 3 (puste pola parametrów pomijano dla jasności).



Rysunek 3 – Zapytanie oraz wyniki wykonania zapytania GET dla kilku obiektów

Po tym przeprowadzono testowanie, podobne do wcześniejszego, z załączeniem parametrów paginacji – są to ilość obiektów do omijania oraz ilość obiektów do pobrania, a także parametrów sortowania w wyglądzie nazwy pola i kolejności. Testowanie jest wykazane na rysunku 4, gdzie przedstawiono zapytanie na omijanie jednego obiektu, pobraniu trzech, po sortowaniu kolekcji za imieniem (polem „Name”) w kolejności rosnącej (wyrażonej znaczeniem „true”). Oczekiwany rezultat: zostaną pobrane obiekty o nazwach „Cinema 2”, „Cinema 3”, „Cinema 4”, w zaznaczonej kolejności.

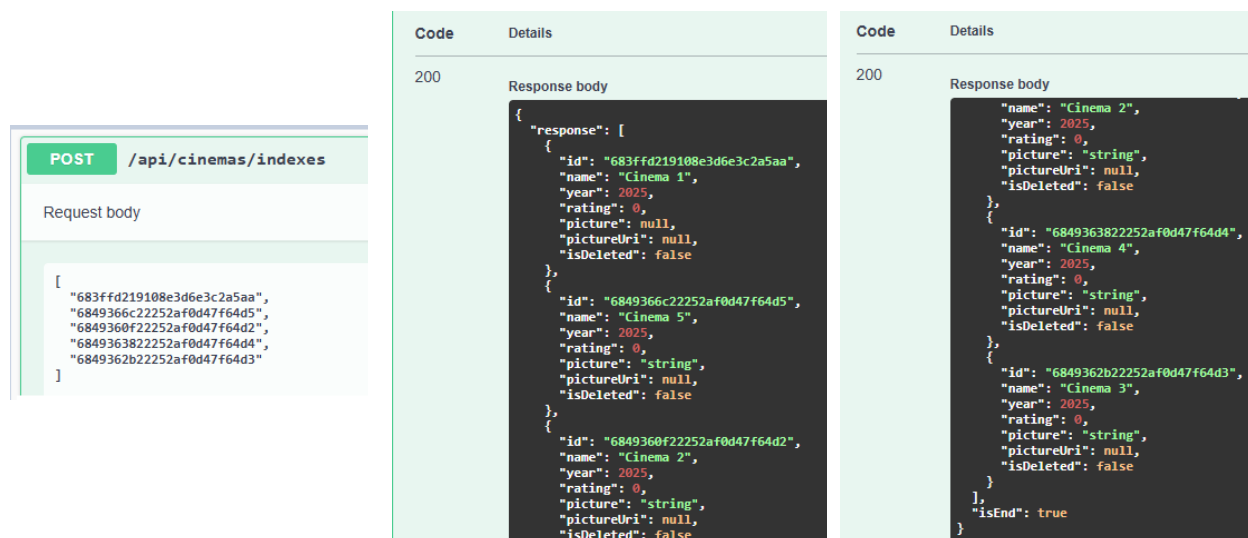




Rysunek 4 – Zapytanie oraz wyniki wykonania zapytania GET dla kilku obiektów z załączeniem parametrów paginacji i sortowania

Jak widać z rysunku 4, oczekiwania wobec wyniku zapytania zostały sprawdzone.

Innym zapytaniem jest pobieranie kilku obiektów według zaznaczonych identyfikatorów. Wpisane identyfikatory odpowiadają szeregu obiektów o nazwach „Cinema 1”, „Cinema 5”, „Cinema 2”, „Cinema 4”, „Cinema 3”. Kolejność w odpowiedzi musi się zgadzać z kolejnością w zapytaniu. Testowanie przedstawione jest rysunkiem 5 (pomijano puste pola parametrów).

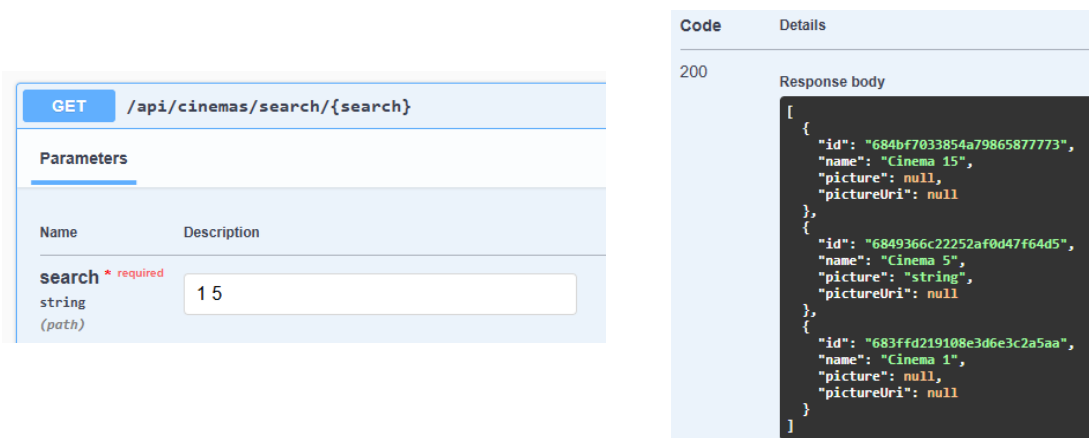


Rysunek 5 – Zapytanie POST do pobierania obiektów według szeregu identyfikatorów

Jak widać z rysunku 5, szereg zwróconych obiektów odpowiada zaznaczonym identyfikatorom, w tym i w kolejności ich zaznaczenia w zapytaniu wejściowym.

Sprawdzono także zapytania do pobierania listy obiektów ze sprawdzeniem filtrowania według określonych pól: nazwy, roku produkcji, gatunków filmowych, języka, identyfikatora wytwórni filmowej.

Zapytanie po nazwie przyjmuje wiersz symboli, który jest przetwarzany przez podział na tokeny według znaków odstępu. Po danym tokenom odbywa się wyszukiwanie z użyciem wyrażenia regularnego, które zapewnia wyszukiwanie niezależne od wielkości liter. W celu sprawdzenia danego zapytania został dodany obiekt o nazwie „Cinema 15”, sam wiersz wejściowy – „1 5”. Oczekiwany wynik: obiekty o nazwach „Cinema 15”, „Cinema 5”, „Cinema 1”. Testowanie jest przedstawione rysunkiem 6 (pomijano puste pola parametrów).



The image shows a REST client interface. On the left, a GET request is configured for the endpoint `/api/cinemas/search/{search}`. The parameters section shows a required string parameter `search` with the value `1 5`. On the right, the response details show a 200 status code and a JSON response body containing an array of three cinema objects.

```

Code    Details
200     Response body
[
  {
    "id": "684bf7033854a79865877773",
    "name": "Cinema 15",
    "picture": null,
    "pictureUri": null
  },
  {
    "id": "6849366c22252af0d47f64d5",
    "name": "Cinema 5",
    "picture": "string",
    "pictureUri": null
  },
  {
    "id": "683ffd219108e3d6e3c2a5aa",
    "name": "Cinema 1",
    "picture": null,
    "pictureUri": null
  }
]

```

Rysunek 6 – Zapytanie oraz wyniki wykonania zapytania GET dla kilku obiektów z filtrowaniem według nazwy

W podobny sposób przeprowadzono test na sprawdzenie tego, czy wyszukiwanie prawdziwie nie zależy od wielkości liter. W tym przypadku używano także dodatkowe parametry paginacji w celach ograniczenia długości odpowiedzi. Testowanie z wprowadzeniem wierszu „NEMA”, a także parametrów paginacji, jest przedstawione rysunkiem 7.

**GET** /api/cinemas/search/{search}

**Parameters**

Name	Description
<b>search</b> * required string (path)	NEMA
skip integer(\$int32) (query)	0
take integer(\$int32) (query)	3

**Code** **Details**

200

**Response body**

```
[
  {
    "id": "684bf7033854a79865877773",
    "name": "Cinema 15",
    "picture": null,
    "pictureUri": null
  },
  {
    "id": "6849366c22252af0d47f64d5",
    "name": "Cinema 5",
    "picture": "string",
    "pictureUri": null
  },
  {
    "id": "6849363822252af0d47f64d4",
    "name": "Cinema 4",
    "picture": "string",
    "pictureUri": null
  }
]
```

Rysunek 7 – Zapytanie oraz wyniki wykonania zapytania GET dla kilku obiektów z filtrowaniem według nazwy i załączeniem parametrów paginacji

Z rysunku 7 widać, że danym zapytaniem pobrano obiekty, nazwa których mieści ciąg „nema”, zapisany małymi literami, co oznacza prawidłowość oczekiwanego wyniku.

W testowaniu z rysunku 6 zapytanie z filtrowaniem po nazwie zostało sprawdzone z użyciem ciągu, który rozbija się na tokeny o długości jeden. Rzeczywiście, wyszukiwanie według tokenu dopiero o jednym znaku nie jest optymalne – z tego powodu warstwy zapytujące po dane wprowadzają ograniczenie na minimalną długość ciągu znaków, odmiennych od znaku odstępu (wybrano optymalną długość w trzy znaki).

Przed przejściem do testowania pozostałych zapytań filtrujących dane, zostały przetestowane zapytania edytujące dane. Testowanie rozpoczęto ze spróby edytowania obiektu po wszystkim polom, które wysyłają się zapytaniem tworzącym obiekt. Dla testowania został utworzony obiekt o nazwie „Cinema 61”. Jego początkowy stan, a także stan po testowanym zapytaniu, przedstawiono rysunkiem 8.

**PUT** /api/cinemas/{id}

**Parameters**

Name	Description
<b>id</b> * required string (path)	684c0bb3d045a4546a09d269

**Request body**

```
{
  "name": "Cinema 6",
  "releaseDate": "2025-05-29",
  "genres": 7,
  "language": 15,
  "picture": "string",
  "pictureUri": null,
  "productionStudios": [],
  "starrings": [],
  "description": "This is Cinema 6"
}
```

**Code** **Details**

200

**Response body**

```
{
  "id": "684c0bb3d045a4546a09d269",
  "name": "Cinema 6",
  "releaseDate": "2025-05-29",
  "genres": 7,
  "language": 15,
  "rating": {
    "score": 0,
    "n": 0
  },
  "picture": "string",
  "pictureUri": null,
  "productionStudios": [],
  "starrings": [],
  "description": "This is Cinema 6"
}
```

Rysunek 8 – Zapytanie oraz wyniki wykonania zapytania PUT z prezentacją stanów obiektu przed i po dokonaniu zapytania

Można dodatkowo upewnić się w tym, że stan obiektu po przeprowadzeniu zmian został zachowany, przez pobranie obiektu za identyfikatorem.

**GET** /api/cinemas/{id}

**Parameters**

Name	Description
<b>id</b> * required string (path)	684c0bb3d045a4546a09d269

**Code** **Details**

200

**Response body**

```
{
  "id": "684c0bb3d045a4546a09d269",
  "name": "Cinema 6",
  "releaseDate": "2025-05-29",
  "genres": 7,
  "language": 15,
  "rating": {
    "score": 0,
    "n": 0
  },
  "picture": "string",
  "pictureUri": null,
  "productionStudios": [],
  "starrings": [],
  "description": "This is Cinema 6"
}
```

Rysunek 9 – Zapytanie oraz wyniki wykonania zapytania GET za identyfikatorem zmienionego obiektu

Z rysunku 9 widać, że odpowiedź właściwie zawiera obiekt o zmienionych polach.

W analogiczny sposób działają pozostałe zapytania z edytowania obiektu z różnicą w ilości pól zmienianych od razu. Na rysunku 10 przedstawiono wyniki zapytań z edytowania dopiero głównych pól oraz zmiany nazwy obrazu.

The figure consists of four screenshots arranged in a 2x2 grid, showing REST client requests and responses for PUT operations on a cinema object.

**Top Left: PUT /api/cinemas/{id}/main**

**Parameters:**

Name	Description
<b>id</b> * required	
string (path)	684c0bb3d045a4546a09d269

**Request body:**

```
{
  "name": "Cinema 16",
  "releaseDate": "2025-05-29",
  "genres": 11,
  "language": 20,
  "picture": "string",
  "productionStudios": [null, null, null],
  "starrings": [null, null, null],
  "description": "This is Cinema 16"
}
```

**Top Right: Response body (200)**

```
{
  "id": "684c0bb3d045a4546a09d269",
  "name": "Cinema 16",
  "releaseDate": "2025-05-29",
  "genres": 11,
  "language": 20,
  "rating": {
    "score": 0,
    "n": 0
  },
  "picture": "string",
  "pictureUri": null,
  "productionStudios": [],
  "starrings": [],
  "description": "This is Cinema 16"
}
```

**Bottom Left: PUT /api/cinemas/{cinemaId}/picture**

**Parameters:**

Name	Description
<b>cinemaId</b> * required	
string (path)	684c0bb3d045a4546a09d269

**Request body:**

```
{
  "picture": null
}
```

**Bottom Right: Response body (200)**

```
{
  "id": "684c0bb3d045a4546a09d269",
  "picture": null,
  "pictureUri": null
}
```

Rysunek 10 – Zapytania oraz wyniki wykonania zapytań PUT na zmianę pól obiektu różnych zakresów

Różnica w zapytaniach ze zmiany wszystkich pól oraz dopiero głównych pól polega na tym, że znaczenia pól obiektów wtórnych (w tym przypadku pól „productionStudios” i „starrings”) są ignorowane, co widać na prawym górnym wyniku rysunku 10 – zamiast list o trzech znaczeniach „null” znaczenia pól nadal są reprezentowane pustą listą.

## Wnioski

...

Jednym z problemem powstałym w procesie ogarnięcia przesyłania danych był problem z zachowaniem integralności danych. Każda operacja zmiany stanu obiektu głównego lub jego usunięcie, wymagała przeprowadzenia ciągu zmian dla innych obiektów, co wyrażało się w relatywnie skomplikowanych prawach logiki na warstwie wymiany danymi. Przyczyna tego leży w sposobie zachowania obiektów powiązanych wyrażającego się w umieszczeniu tego rodzaju informacji jako część obiektów głównych. Podobne podejście skutkuje trudnością przeprowadzenia zmian w ciągu projektowania, optyimizacji, a w gorszych przypadkach, gdy nie jest zabezpieczona pełna aktualizacja obiektów, potrafi wyniknąć anomalia powiązana z przedstawieniem różnych informacji o jednym i tym samym obiekcie.

Rozwiązaniem powyższego problemu leżało by w zmianie struktury obiektów bazy, gdy by informacja o zapisach powiązanych przedstawiała z siebie dopiero indeksy, posyłające się na zapisy w oddzielnych kolekcjach. Przy takim podejściu czas ładowania strony może trochę się zwiększyć przez zapytania na niedostarczające informacje o zapisach, jednak pozwoliło by to na uniknięcie bardziej poważnych problemów w ciągu dalszej pracy nad aplikacją.

Ewentualnie można było by zmienić i typ bazy z NoSQL na tradycyjny relatywny SQL, skoro bazy danego typu uważają się być bardziej dostosowane do wykonania zapytań z użyciem operacji „JOIN”, niezbędnej dla encji w relacjach „Wiele-do-Wielu”. Z innej strony, niektóre bazy NoSQL potrafią wykonywać podobne zapytania z małą utratą wydajności, a ich jedna z głównych cech pozytywnych polega na elastyczności przy dostosowaniu się do zmian struktury encji, co jest trudniejsze do zabezpieczenia w przypadku baz SQL. Skoro materiał encyklopedyczny może z czasem przedstawiać więcej kategorii informacji o obiektach, na przykład, odnośnie statystyki, dana cecha elastyczności staje kluczową.

Oprócz problemu z warstwą dostarczenia danych, powstał także problem z organizacji widoków, a dokładnie wydarzeń dla nich. Powiązane to jest z wyborem technologii w wyglądzie Razor Pages, która ogarnia dopiero generację widoków statycznych, gdy niektóre wydarzenia (np. z aktualizacji danych strony) wymagają dynamicznej zmiany napełnienia strony. Przez takie ograniczenie niezbędne było wykorzystywać technologie czystego języka JavaScript do ogarnięcia dynamiki. Nie zważając na to, że za pomocą tego rozwiązania zostały osiągnięte żądane

wyniki, sam proces napisania był nieco niewygodny, z powodu operowania się na modelach języka C# w języku pozbawionym typizacji. W tym przypadku wygodniejsze było by użycie języka TypeScript, już wspierającego typizację danych. Jednak bardziej prostszą alternatywą stało by użycie technologii Blazor, która łączy generację widoków Razor a jednocześnie możliwości napisania wydarzeń z użyciem czystego C# i jest także przedstawiana przez .NET platformę. Oprócz tego używa ona pod spodem technologii WebAssembly do możliwości wykonania C# kodu w samej przeglądarce z jego optyimizacją, dzięki czemu zabezpiecza się jednocześnie wykorzystanie kodu natywnego dla serwisu widoków oraz wydajność podczas wykonania wydarzeń.

Ogarnięcie doskonałego zabezpieczenia API potrzebowała by oczywiste rozwiązanie w jakości dołączenia faktycznego dostawcy usług autoryzacyjnych, który mógł by także wykonywać funkcję zachowania zasobów użytkownika. Rozwiązaniem mogła by służyć usługa Identity z platformy Azure, zasoby której już są wykorzystane w danej aplikacji. Ogólny problem z załączeniem usług faktycznego nadawcy jest podobny do problemu z wykorzystaniem usług chmury obliczeniowej – rozwiązanie może okazać się kosztowne przy rozrastaniu aplikacji na większą ilość użytkowników. Z innej strony, dane użytkowników oraz API w tym przypadku są porządnie zabezpieczone oraz sam serwis staje prostszy w konfiguracji, w tym i organizacji bardziej złożonych ograniczeń na dane.

## Bibliografia

- [1] Evans, E. (2004). *Domain Driven Design – Tackling Complexity in the Heart of Software*. MA, Westford: Addison-Wesley Professional.
- [2] Elmasri, R. (2008). *Fundamentals of database systems*. Pearson Education India.
- [3] Pereira, P. A., & Bruce, M. (2019). *Microservices in Action*. NY, Shelter Island: Manning Publications.
- [4] Richardson, C. (2018). *Microservices patterns: with examples in Java*. Simon and Schuster.
- [5] Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media, Inc.
- [6] *Model-View-ViewModel (MVVM)*. (2024, Sep 10). Pobrano z <https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm>.
- [7] Hardt, D. (2012). *The OAuth 2.0 authorization framework* (No. RFC 6749).
- [8] Jones, M., Bradley, J., & Sakimura, N. (2015). *Json web token (jwt)* (No. RFC 7519).