**Міністерство ОСВІТИ І НАУКИ УКРАЇНИ**

**Чернівецький національний університет імені Юрія Федьковича**

**Кафедра комп’ютерних наук**

**КУРСОВА РОБОТА**

на тему:

**"Онлайн кінотеатр «FilmOS»"**

Виконав студент

групи 344СК \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ М.Ю.Максимович

(підпис)

Науковий керівник \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ М. Л. Ковальчук

(підпис)

Національна шкала \_\_\_\_\_\_\_ Кількість балів \_\_\_\_\_\_\_ Оцінка ECTS \_\_\_\_\_

Чернівці – 2022

**Зміст**

[Вступ 3](#_Toc105839758)

[1 ТеорЕТИЧНА ЧАСТИНА 5](#_Toc105839759)

[1.1 Актуальність розробки 5](#_Toc105839760)

[1.2 Огляд 5](#_Toc105839761)

[1.3 Властивості 6](#_Toc105839762)

[2 Проектування 7](#_Toc105839763)

[2.1 Опис предметної області 7](#_Toc105839764)

[2.2 Моделювання даних 8](#_Toc105839765)

[2.2.1 Моделювання даних (діаграми програмного продукту та БД) 8](#_Toc105839766)

[2.2.1.1 Favorite DB 8](#_Toc105839767)

[2.2.1.2 Shopping DB 11](#_Toc105839768)

[2.2.1.3 Rating DB 14](#_Toc105839769)

[2.2.1.4 Basket DB 15](#_Toc105839770)

[2.2.2 Моделювання даних (концептуальна, логічна) 16](#_Toc105839771)

[2.2.2.1 Favorite DB 16](#_Toc105839772)

[2.2.2.2 Shopping DB 17](#_Toc105839773)

[2.2.2.3 Rating та Basket DB 18](#_Toc105839774)

[2.2.3 Функціональне моделювання (діаграми прецедентів) 18](#_Toc105839775)

[2.3 Засоби та технології розробки 20](#_Toc105839776)

[3 РОБОЧИЙ ПРОЕКТ 23](#_Toc105839777)

[3.1 Інтерфейс веб додатку 23](#_Toc105839778)

[3.2 Інструкція програмісту 28](#_Toc105839779)

[Висновки 29](#_Toc105839780)

[Перелік джерел 30](#_Toc105839781)

[Додатки 31](#_Toc105839782)

# Вступ

Завданням курсової роботи є реалізація мікросервісів для онлайн кінотеатру «FilmOS».

Об’єктом дослідження курсової роботи є бази даних (БД) та програмно-апаратна частина сервісу (backend).

Предметом дослідження курсової роботи є проєктування баз даних та розробка мікросервісів.

Метою курсової роботи є поглиблення знань при роботі із базами даних та удосконалення практичних навичок при створенні та розробці мікросервісів. Необхідно забезпечити швидкий та вільний доступу до ресурсу для комфортного перегляду фільмів.

Основною задачею веб додатку є показ відео-продукту. До основних їх видів належать фільми, серіали та трейлери.

До принципових задач відносять:

* забезпечення збереження та ефективний пошук;
* зручний та інтуїтивний інтерфейс;
* збільшення швидкості обробки запитів пов’язаних з формуванням даних кошика;
* мінімізацію можливості виникнення помилок у роботі з даними, які може зробити оператор;
* контроль даних, що додаються у базу даних.

До функціональних задач відносять наступні:

* збереження даних в довідниках;
* пошук даних за різними критеріями;
* можливість розрахунків та збереження їх результатів;

До сервісних задач відносяться:

* організація головного меню;
* забезпечення маніпуляції з даними.

Вимоги надійності.

* програмний продукт повинен надійно працювати;
* Забезпечити різні рівні доступу, а саме користувач та адміністратор.
* розробник гарантує роботу програмного продукту без збоїв.

Веб додаток повинен підтримувати спільний режим роботи з основними браузерами, такими як Microsoft Edge, Google Chrome, Mozilla Firefox .

Для реалізації програмного продукту обрано середовище розробки Microsoft Visual Studio 2019, а також для роботи з базами даних використовувались Microsoft SQL Server 2019, MongoDB, Redis. Додатково були використані технології RabbitMQ, Ocelot, Docker, Git.

# 1 ТеорЕТИЧНА ЧАСТИНА

## 1.1 Актуальність розробки

На сьогоднішній день онлайн-кінотеатри не втратили свою актуальність, а здебільшого навіть її збільшили. Особливо в умовах карантину або неможливості виходу в людяні місця, такі як класичний кінотеатр. Саме тому було обрано розробити мікросервіс для онлайн кінотеатру, адже онлайн перегляд фільмів має велику популярність.

Поставлені задачі на розробку програмного забезпечення для користувачів та адміністратора.

## 1.2 Огляд

Мікросервіси — архітектурний стиль, за яким єдиний застосунок будується як сукупність невеличких сервісів, кожен з яких працює у своєму власному процесі та спілкується з рештою, використовуючи прості та швидкі протоколи передачі даних, зазвичай HTTP. Ці сервіси будуються навколо бізнес-потреб і розгортаються незалежно один від одного з використанням зазвичай повністю автоматизованого середовища. Існує абсолютний мінімум централізованого керування цими сервісами. Самі по собі вони можуть бути написані з використанням різних мов програмування і технологій зберігання даних.

Мікросервісна архітектура зручна для реалізації процесу безперервної поставки програмного продукту, на відміну від сервіс-орієнтовної архітектури, мікросервісна спрямована на створення одного застосунку, в той час як сервісно орієнтована система являє собою множину застосунків, які взаємодіють між собою.

## 1.3 Властивості

Основні властивості мікросервісів:

* Високий рівень незалежності: незалежна розробка, незалежне розгортання
* Незалежне масштабування
* Невелика кодова база зменшує кількість конфліктів та дозволяє швидко залучати нових розробників
* Простота заміни однієї реалізації сервісу іншою
* Простота додавання нової функціональності в систему
* Ефективне використання ресурсів
* Еластичність: вихід з ладу одного сервісу зазвичай не призводить до виходу з ладу всієї системи
* Сервіси організовані відносно бізнес логіки яку вони виконують
* Кожен сервіс незалежно від інших може бути реалізований за допомогою будь-якої мови програмування, СУБД, та ін.
* Архітектурно побудовані за симетричним принципом (виробник-споживач)

# 2 Проектування

## 2.1 Опис предметної області

Основним завданням реалізації програмного продукту було створення сукупності мікросервісів, куди входять три початкові бази даних, але під час розробки добавилось ще одна. Основною перевагою є можливість масштабування і додавання нових баз даних за їх необхідності.

Окрім розробки БД для кожної із них було покладено задачу розробити REST API. А саме CRUD (Create, Reade, Update, Delete) запити які будуть викликатись по http протоколу.

Заплановані можливості адміністратора:

* Додавання, редагування та видалення інформації фільму або серіалу.
* Перевірка коректності даних.

Заплановані можливості користувачів:

* Перегляд всіх доступних фільмів, серіалів.
* Пошук фільму або серіалу.
* Додавання коментаря, відгуку до фільму.
* Покупка підписки на платний фільм або серіал.
* Покупка прокату окремого фільму або серіалу.
* Додавання фільму в обрані або ж список «переглянути пізніше».

Для кожного користувача бази даних (Адміністратор, користувач) створити окремі логіни для входу в базу та надати відповідний доступ.

## 2.2 Моделювання даних

### 2.2.1 Моделювання даних (діаграми програмного продукту та БД)

Розглянемо покроково всі бази даних та їх структуру.

#### 2.2.1.1 Favorite DB

Вся інформація про обрані (улюблені) фільми зберігається в базі даних Favorite.

Таблиця Films призначення для зберігання інформації про фільми, її структура представлена на рисунку 1:

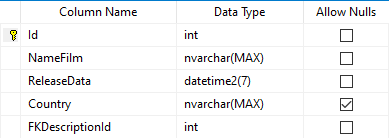


Рисунок 1 – Структура таблиці Films

Таблиця містить такі поля:

* Id – числове поле, ключ, включає в себе id фільму;
* NameFilm – короткий текст, призначено для збереження назви фільму;
* ReleaseData – поле із датою випуску фільму;
* Country – текстове поле, країна в якій проводились зйомки;
* FKDescriptionId – числове поле, id який вказує на таблицю Description із коротким описом фільму. Опис може бути унікальним для кожного фільму, а можна призначити один із початкових шаблонів.

Усі поля крім Country є обов’язковими для введення.

Таблиця Description призначення для зберігання короткого опису фільму, її структура представлена на рисунку 2.

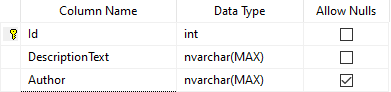


Рисунок 2 – Структура таблиці Description

Таблиця містить такі поля:

* Id – числове поле, ключ;
* DescriptionText – текстове поле, текст із описом;
* Author – текстове поле, містить ім’я або нікнейм автора опису;

Усі поля крім Author є обов’язковими для введення, так як автор може залишатись анонімним.

Таблиця Users призначення для зберігання інформації про зареєстрованих користувачів, її структура представлена на рисунку 3:

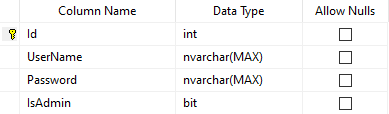


Рисунок 3 – Структура таблиці Users

Таблиця містить такі поля:

* Id– числове поле, id користувача;
* UserName – текст, ім’я або нікнейм користувача;
* Password –текстове поле яке зберігає пароль користувача;
* IsAdmin – поле bit, або ж boolean, вказує чи є користувач Адміном.

Усі поля є обов’язковими для введення.

Таблиця FilmsUsers реалізує зв'язок ManyToMany між таблицями Films та Users і призначена для зберігання улюблених фільмів, її структура представлена на рисунку 4.

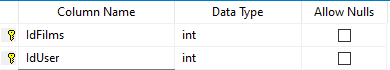


Рисунок 4 – Структура таблиці FilmsUsers.

Таблиця містить такі поля:

* IdFilms – ключове поле, вказує на id фільму;
* IdUser – ключове поле, вказує на id користувача;

Усі поля є обов’язковими для введення оскільки вони є складеним первинним ключем.

Таблиця Genres призначення для зберігання списку жанрів, її структура представлена на рисунку 5:

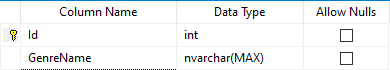


Рисунок 5 – Структура таблиці Genres.

Таблиця містить такі поля:

* Id – лічильник, ключове поле;
* GenreName – короткий текст, назва жанру для фільму;

Таблиця FilmsGenres реалізує зв'язок ManyToMany між таблицями Films та Genres і призначена для формування списку жанрів для одного фільму, її структура представлена на рисунку 6.

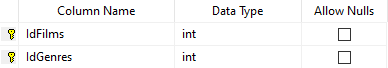


Рисунок 6 – Структура таблиці FilmsGenres.

Таблиця містить такі поля:

* IdFilms – ключове поле, вказує на id фільму;
* IdGenres – ключове поле, вказує на id жанру;

Усі поля є обов’язковими для введення.

#### 2.2.1.2 Shopping DB

Вся інформація про список фільмів, серіалів або підписок, які користувач хоче купити, зберігається в базі даних Shopping.

Таблиця Films призначена для зберігання інформації про фільми. Її структура представлена на рисунку 7:

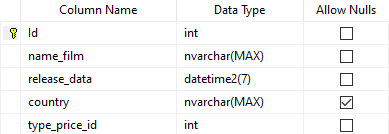


Рисунок 7 – Структура таблиці Films

Таблиця містить такі поля:

* Id – числове поле, ключ, включає в себе id фільму;
* NameFilm – короткий текст, призначено для збереження назви фільму;
* ReleaseData – поле із датою випуску фільму;
* Country – текстове поле, країна в якій проводились зйомки;
* type\_price\_id – числове поле, id який вказує на таблицю type\_price. Усі поля крім Country є обов’язковими для введення.

Таблиця Types\_prices є списком зі значеннями призначеними для того, щоб відрізняти цінову категорію фільму. Її структура представлена на рисунку 8.

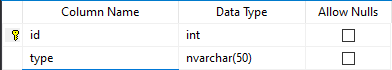


Рисунок 8 – Структура таблиці Types\_prices

Таблиця містить такі поля:

* Id – числове поле, ключ;
* type – текстове поле, тип цінової категорії;

Тип може бути: платний, безоплатний, лише за наявності підписки.

Усі поля є обов’язковими для введення.

Таблиця Users призначення для зберігання інформації про зареєстрованих користувачів, її структура представлена на рисунку 9:

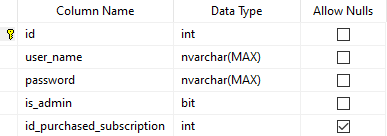


Рисунок 9 – Структура таблиці Users

Таблиця містить такі поля:

* Id– числове поле, id користувача;
* UserName – текст, ім’я або нікнейм користувача;
* Password –текстове поле яке зберігає пароль користувача;
* IsAdmin – поле bit, або ж boolean, вказує чи є користувач Адміном.
* id\_purchased\_subscription – числове поле, ключ, вказує на id купленої підписки, може бути null, тоді вважатиметься що користувач немає підписки. Усі інші поля є обов’язковими для введення.

Таблиця Type\_subscriptions призначення для зберігання інформації про підписки, її структура представлена на рисунку 10:

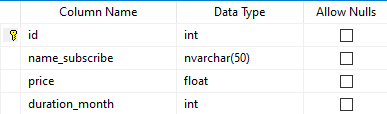


Рисунок 10 – Структура таблиці Type\_subscriptions

Таблиця містить такі поля:

* id – числове поле, id підписки;
* name\_subscribe – текст, офіційна назва підписки (VIP, Premium);
* price – числове поле із плаваючою крапкою, ціна підписки;
* duration\_month – числове поле, тривалість підписки.

Усі поля є обов’язковими для введення.

Таблиця Basket\_subscriptions реалізує зв'язок ManyToMany між таблицями Users та Type\_subscriptins і виконує роль корзини для покупки підписки, її структура представлена на рисунку 11.

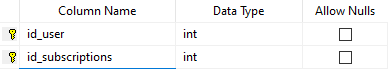


Рисунок 11 – Структура таблиці Basket\_subscriptions.

Таблиця містить такі поля:

* id\_user – ключове поле, вказує на id користувача;
* id\_subscriptions – ключове поле, вказує на id підписки;

Усі поля є обов’язковими для введення оскільки вони є складеним первинним ключем.

Таблиця Basket\_films реалізує зв'язок ManyToMany між таблицями Films та Users і виконує роль корзини для покупки фільму, її структура представлена на рисунку 12.

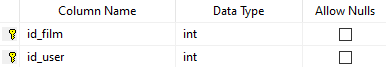


Рисунок 12 – Структура таблиці Basket\_films.

Таблиця містить такі поля:

* id\_film – ключове поле, вказує на id фільму;
* id\_user – ключове поле, вказує на id користувача;

Усі поля є обов’язковими для введення.

#### 2.2.1.3 Rating DB

В цій БД реалізований каталог коментарів, які користувачі можуть залишати під фільмами. В якості БД використовується Mongo. Для візуалізації рисунки із структурою будуть використовуватись із MongoDB Compass.

Колекція Comments містить в собі два поля, які відповідають ключам в реляційній базі, та текст коментаря. Її структура представлена на рисунку 13.



Рисунок 13 – Структура колекції Comments

Колекція містить такі поля:

* \_id– ObjectId, приймає текстове значення, id коментаря в базі;
* \_id\_sql\_film – int, id фільма в реляційній базі даних, для якого було написано коментар;
* \_id\_sql\_user – int, id користувача в реляційній базі даних, який написав коментар;
* Text – текстове поле, містить повідомлення яке написав користувач.

Усі поля є обов’язковими для введення.

#### 2.2.1.4 Basket DB

База даних Basket реалізована за допомогою Redis. Основний її функціонал це швидкий доступ до вмістимого кошика. Це прекрасно реалізовується завдяки тому що він зберігає дані в kay-value форматі.

В базі використовується 2 класи:

ShoppingCartItem:

* Quantity - int
* Price – decimal
* ProductName - string
* ProductId - string

ShoppingCart:

* UserName – string
* Items - List<ShoppingCartItem>

ShoppingCart зберігає список ShoppingCartItem, тобто об’єктів для купівлі.

### 2.2.2 Моделювання даних (концептуальна, логічна)

#### 2.2.2.1 Favorite DB

На рисунку 14 зображено логічну модель даних для favorite db:

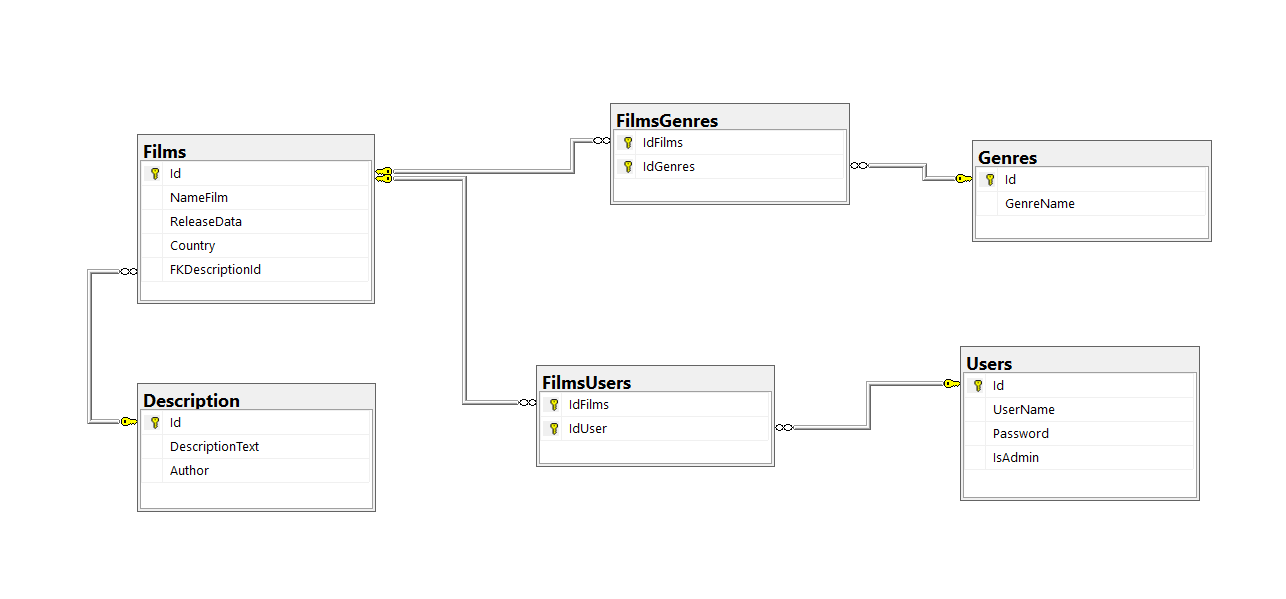


Рисунок 14 – Логічна модель даних

На схемі даних зображено таблиці та зв’язок між ними.

Поле FKDescriptionId в таблиці Films зв’язано багато до одного з ключовим полем Id таблиці Description для того, аби в таблиці Films мати інформацію про опис цього фільму.

Поле IdFilms таблиці FilmsGenres зв’язано багато до одного з полем Id в таблиці Films. Поле IdGenres таблиці FilmsGenres зв’язано багато до одного з полем Id в таблиці Genres. Таким чином реалізовується зв'язок багато до багатьох між таблицями Films та Genres.

Поле IdFilms таблиці FilmsUsers зв’язано багато до одного з полем Id в таблиці Films. Поле IdUsers таблиці FilmsUsers зв’язано багато до одного з полем Id в таблиці Users. Таким чином реалізовується зв'язок багато до багатьох між таблицями Films та Users.

#### 2.2.2.2 Shopping DB

На рисунку 15 зображено логічну модель даних для shopping db:

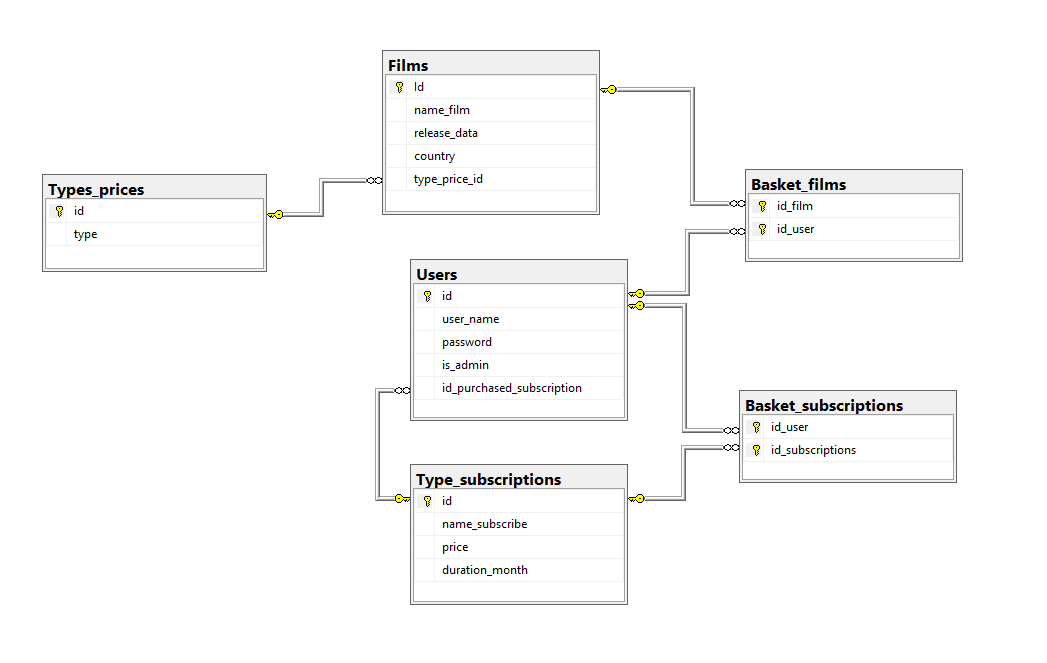


Рисунок 15 – Логічна модель даних

На схемі даних зображено таблиці та зв’язок між ними.

Поле type\_price\_id в таблиці Films зв’язано багато до одного з ключовим полем id таблиці Types\_prices для того, аби фільму присвоїти цінову категорію.

Поле id\_purchased\_subscription в таблиці Users зв’язано багато до одного з ключовим полем id таблиці Type\_subscriptions, щоб розуміти чи у користувача є куплена підписка і яка саме.

Поле id\_film таблиці Basket\_films зв’язано багато до одного з полем Id в таблиці Films. Поле id\_user таблиці Basket\_films зв’язано багато до одного з полем id в таблиці Users. Таким чином реалізовується зв'язок багато до багатьох між таблицями Films та Users.

Поле id\_user таблиці Basket\_subscriptions зв’язано багато до одного з полем id в таблиці Users. Поле id\_subscriptions таблиці Basket\_subscriptions зв’язано багато до одного з полем id в таблиці Type\_subscriptions. Таким чином реалізовується зв'язок багато до багатьох між таблицями Type\_subscriptions та Users.

#### 2.2.2.3 Rating та Basket DB

Для бази даних Rating та Basket немає логічної моделі, адже вони не мають зав’язків.

На рисунку 16 зображено концептуальна модель даних:

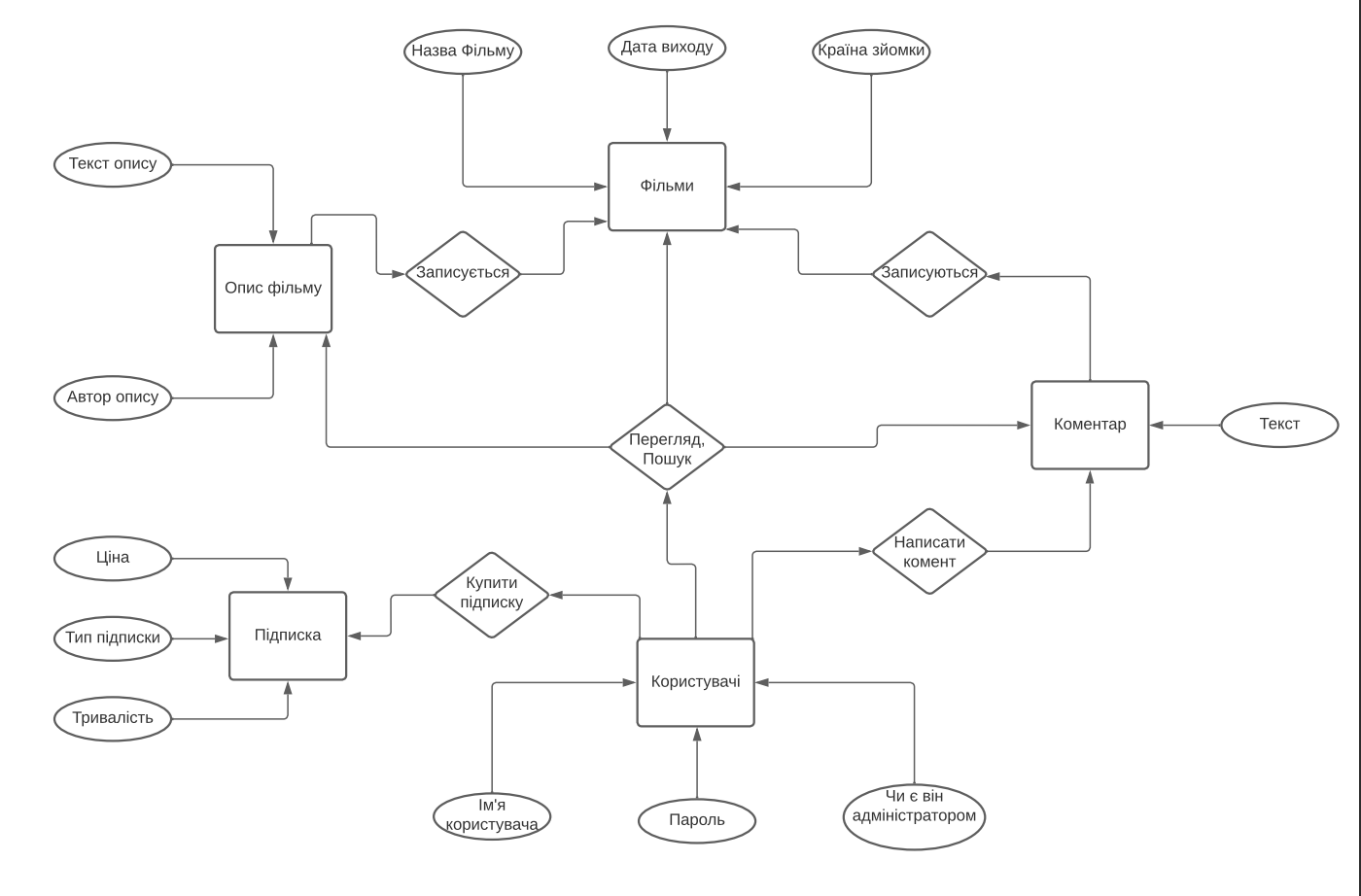


Рисунок 16 – Концептуальна модель даних

### 2.2.3 Функціональне моделювання (діаграми прецедентів)

Діаграма прецедентів – діаграма, на якій зображено відношення між користувачами та прецедентами в системі.

Актор – клієнт, який може переглядати дані, шукати, корегувати по кваліфікації, а також купувати, а адміністратор повинен спочатку авторизуватись, якщо він не пройде авторизацію, то він не зможе працювати з даними, до його можливостей роботи з даними входить перегляд, введення та видалення.

На рисунку 17 зображена діаграма прецедентів користувача:

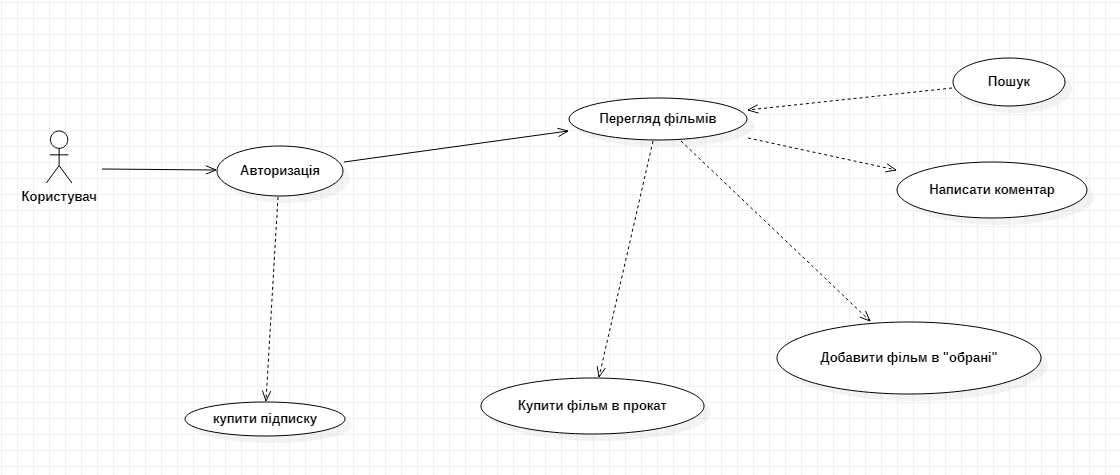


Рисунок 17 – Діаграма прецедентів користувача

На рисунку 18 зображена діаграма прецедентів адміністратора:

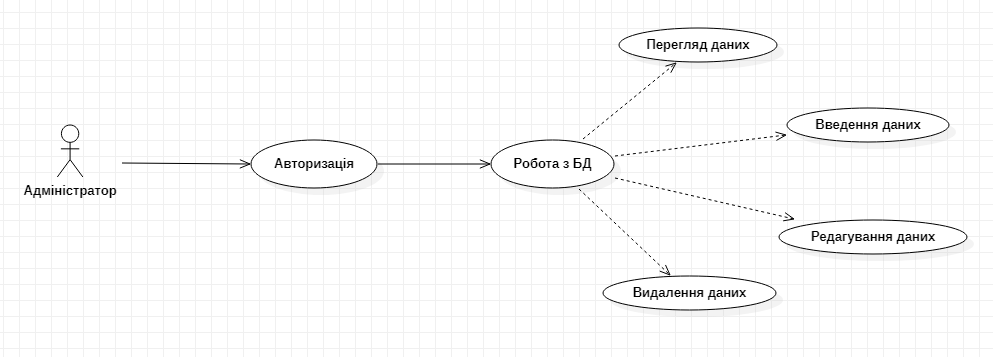


Рисунок 18 – Діаграма прецедентів адміністратора

## 2.3 Засоби та технології розробки

Для реалізації програмного продукту обрано середовище розробки додатків Microsoft Visual Studio 2019.

Інтегроване середовище розробки Visual Studio – це стартовий майданчик для написання, налагодження та складання коду, а також подальшої публікації програм. Він є досить гнучким та зручним для створення програмних продуктів. Його об’єкти та розширені бібліотеки можуть не тільки оформити отримання та динамічну обробку інформації, а й графічно і режимі реального часу відображати зміну вхідних параметрів, математичної моделі та системи і цілому.

Visual Studio – це надзвичайно потужна система розробки прикладних програм для Windows. Вона впевнено займає передові позиції як серед професійних програмістів, так і серед людей, що ніколи професійно не займались програмуванням. Професіонали використовують Visual Studio для побудови складних аплікацій з розподіленими базами даних і для багатьох інших цілей.

Які можливості пропонує Visual Studio?

* Розробка Швидкі навігація, написання та виправлення коду
* Налагодження, Просте налагодження, профіль та діагностика коду
* Комплексні інструменти тестування допомагають писати високоякісний код
* Співробітництво ...

Visual Studio включає редактор вихідного коду з підтримкою технології IntelliSense і можливістю найпростішого рефакторингу коду. Вбудований налагоджувач може працювати як налагоджувач рівня вихідного коду, так і відладчик машинного рівня.

Swagger це набір інструментів, які допомагають описувати API. Завдяки йому користувачі та машини краще розуміють можливості REST API без доступу до коду. За допомогою Swagger можна швидко створити документацію та надіслати її іншим розробникам чи клієнтам.

Microsoft SQL Server — система управління базами даних, яка розробляється корпорацією Microsoft. Як сервер даних виконує головну функцію по збереженню та наданню даних у відповідь на запити інших застосунків, які можуть виконуватися як на тому ж самому сервері, так і у мережі.

Мова, що використовується для запитів — Transact-SQL, створена спільно Microsoft та Sybase. Transact-SQL є реалізацією стандарту ANSI / ISO щодо структурованої мови запитів SQL із розширеннями.

MongoDB — документо-орієнтована система керування базами даних (СКБД) з відкритим вихідним кодом, яка не потребує опису схеми таблиць. MongoDB займає нішу між швидкими і масштабованими системами, що оперують даними у форматі ключ/значення, і реляційними СКБД, функціональними і зручними у формуванні запитів.

Код MongoDB написаний на мові C++ і поширюється в рамках ліцензії AGPLv3.

Redis — розподілене сховище пар ключ-значення, які зберігаються в оперативній пам'яті, з можливістю забезпечувати довговічність зберігання на бажання користувача. Це програмне забезпечення з відкритим сирцевим кодом написане на ANSI C. Розробка Redis фінансується VMware.

RabbitMQ — платформа, що реалізує систему обміну повідомленнями між компонентами програмної системи на основі стандарту AMQP (Advanced Message Queuing Protocol).

Docker — інструментарій для управління ізольованими Linux-контейнерами. Docker доповнює інструментарій LXC більш високорівневим API, що дозволяє керувати контейнерами на рівні ізоляції окремих процесів. Зокрема, Docker дозволяє не переймаючись вмістом контейнера запускати довільні процеси в режимі ізоляції і потім переносити і клонувати сформовані для даних процесів контейнери на інші сервери, беручи на себе всю роботу зі створення, обслуговування і підтримки контейнерів.

# 3 РОБОЧИЙ ПРОЕКТ

## 3.1 Інтерфейс веб додатку

В якості візуалізації проекту використано фреймворк для .net – Swagger.

Було реалізовано 4 REST API із базовими CRUD запитами до основних таблиць.

Swagger зібрав всі url на єдину сторінку, за допомогою яких можна швидко зорієнтуватись в реалізованих функціях.

Нище будуть наведені всі CRUD запити реалізовані для таблиць.

**Favorite**

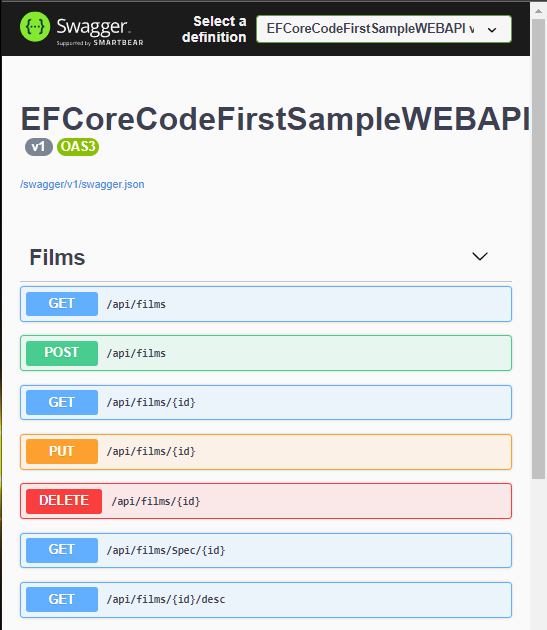


Рисунок 19

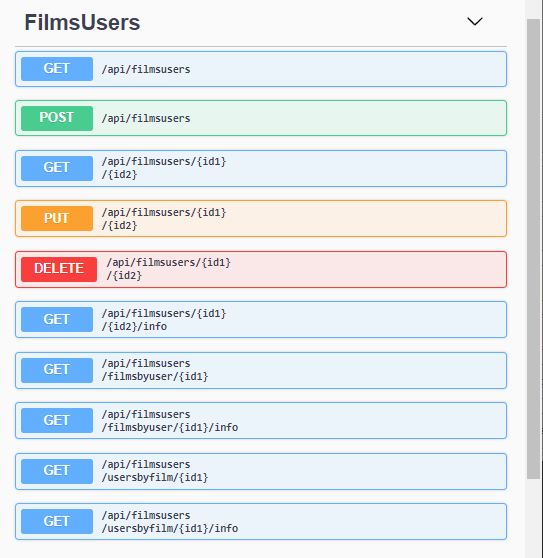


Рисунок 20

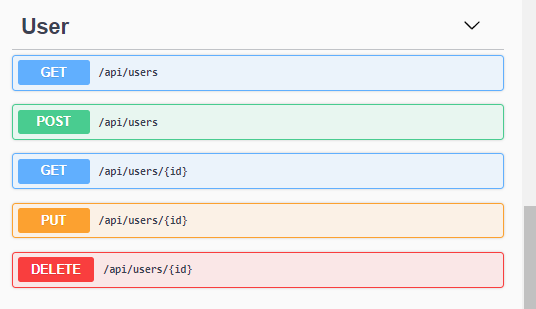


Рисунок 21

**Shoping**

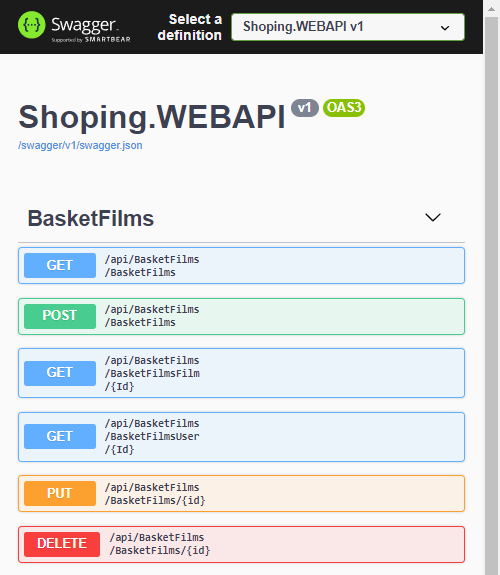


Рисунок 22

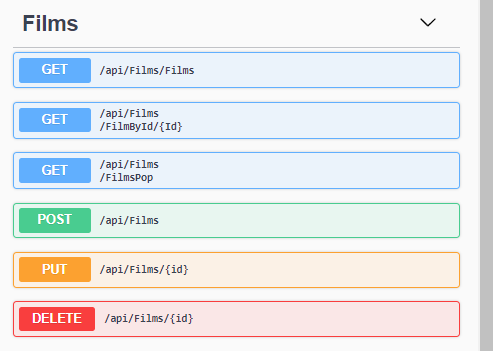


Рисунок 23

**Rating**

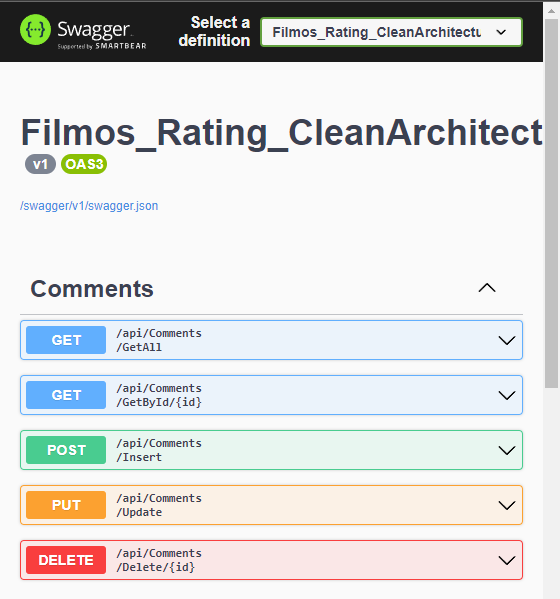


Рисунок 24

## 3.2 Інструкція програмісту

Розгорнути проект можна двома способами:

1. В локальному середовищі

Для Favorite проекту нам потрібно створити пусту базу даних, передати її назву в ConectionSstrin файлу appsettings, та сформувати базу із вже існуючих міграцій за допомогою команди Update-Database в консолі диспетчера пакетів.

Для Shoping ми можемо опублікувати готову модель бази даних натиснувши правою кнопкою по проекту із закінченням .DATABASE -> Публікація -> та вибрати локальний сервер.

Для Rating нам потрібно мати лише встановлений mongodb, передати рядок підключення.

Для Basket потрібно запустити базу даних Redis в Docker

1. З допомогою Docker-compose у віртуальному середовищі

Все що буде потрібно – це мати встановлений Docker на комп’ютері та запустити на виконання docker-compose, найлегше із Visual Studio.

# Висновки

Під час реалізації курсової роботи було здійснено розробку мікросервісу. Спершу було проведено дослідження предметної області та розроблено функціональну модель проекту. Так як дані зберігаються в базі даних, створено фізичну та логічну модель бази даних проекту.

Однією з вимог до кінцевого продукту була організація віддаленого доступу, тому базу даних було створено в Microsoft SQL Server Management Studio 2019, після чого створені логіни доступу (для віддаленого доступу до бази) та налаштовано локальний сервер на головному комп’ютері.

Засобами об’єктно-орієнтованого програмування реалізована операція пошуку за різними критеріями.

Також були розроблені CRUD запити для маніпуляції даними в таблицях.

За допомогою SQL запитів було реалізовано такі операції з таблицями, як додавання, оновлення запису, вибір з випадаючого списку.

З використанням Redis був реалізований кошик для швидкої обробки інформації про замовлення та подальшого зберігання його в кеші.

Був використаний MongoDB для створення каталогу із коментарями.

У проекті застосований Message Brocker RabbitMQ для спілкування та синхронізації даних між реалізованими мікросервісами.

Було сформовано налаштування кожного проекту для його запуску та розгортання в Docker.

За допомогою Swagger була сформована візуалізація та доступ до всіх API проекту.

Також, додатково було використано функціонал Ocelot для того щоб сформувати єдиний API шлюз із всіх проектів.

Працюючи над проектом, я поглибив знання у розробці баз даних та принципів роботи із мікросервісами, удосконалив знання на практиці, здобувши навички при створенні інформаційних систем.

# Перелік джерел

1. Форум программистов - Киберфорум <https://www.cyberforum.ru>
2. Популярна система питань і відповідей для професійних програмістів та ентузіастів «**Stack Overflow**» <https://stackoverflow.com/>
3. Відео-хостинг «YouTube» <https://www.youtube.com>
4. Короткий довідник sql запитів <http://databases.about.com/od/development/>
5. Реляционные базы данных и язык SQL (metanit.com) - <https://metanit.com/sql/>
6. Microsoft SQL documentation - <https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver16>
7. MongoDB Documentation - <https://www.mongodb.com/docs/>

# Додатки

Оскільки проект дуже масштабний – весь код можна знайти у вільному доступі на Github - [Release v1\_coursework · Kolyanuss/FilmOS-microservices](https://github.com/Kolyanuss/FilmOS-microservices/releases/tag/v1) (<https://github.com/Kolyanuss/FilmOS-microservices/releases/tag/v1>)

Нище будуть наведені зразки основних файлів.

## RepositoryBase

using EFCoreCodeFirstSampleWEBAPI.DAL.Interfaces;

using EFCoreCodeFirstSampleWEBAPI.DAL.Specifications;

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Expressions;

using System.Threading.Tasks;

namespace EFCoreCodeFirstSampleWEBAPI.DAL.Repository

{

public abstract class RepositoryBase<T> : IRepositoryBase<T> where T : class

{

protected MyAppContext MyAppContext { get; set; }

public RepositoryBase(MyAppContext myAppContext)

{

MyAppContext = myAppContext;

}

public IQueryable<T> GetAll()

{

return MyAppContext.Set<T>().AsNoTracking();

}

public IQueryable<T> GetByCondition(Expression<Func<T, bool>> expression)

{

return MyAppContext.Set<T>().Where(expression);

}

public IEnumerable<T> FindWithSpecificationPattern(ISpecification<T> specification = null)

{

return SpecificationEvaluator<T>.GetQuery(MyAppContext.Set<T>().AsQueryable(), specification);

}

public async Task Add(T entity)

{

await MyAppContext.Set<T>().AddAsync(entity);

MyAppContext.SaveChanges();

}

public void Update(T entity)

{

MyAppContext.Set<T>().Update(entity);

MyAppContext.SaveChanges();

}

public void Delete(T entity)

{

MyAppContext.Set<T>().Remove(entity);

MyAppContext.SaveChanges();

}

}

}

## MyAppContext

using EFCoreCodeFirstSampleWEBAPI.DAL.Models;

using Microsoft.EntityFrameworkCore;

using System;

namespace EFCoreCodeFirstSampleWEBAPI.DAL

{

public class MyAppContext : DbContext

{

public DbSet<Films> Films { get; set; }

public DbSet<User> Users { get; set; }

public DbSet<FilmsUsers> FilmsUsers { get; set; }

public DbSet<Genres> Genres { get; set; }

public DbSet<Description> Description { get; set; }

public MyAppContext(DbContextOptions options) : base(options)

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<User>().HasKey(u => u.Id);

modelBuilder.Entity<User>().Property(u => u.UserName).IsRequired();

modelBuilder.Entity<User>().Property(u => u.Password).IsRequired();

modelBuilder.Entity<User>().Property(u => u.IsAdmin).IsRequired();

modelBuilder.Entity<Genres>().HasKey(u => u.Id);

modelBuilder.Entity<FilmsUsers>().HasKey(u => new { u.IdFilms, u.IdUser });

modelBuilder.Entity<FilmsGenres>().HasKey(u => new { u.IdFilms, u.IdGenres });

modelBuilder.Entity<FilmsUsers>()

.HasOne(p => p.Films)

.WithMany(t => t.FilmsUsers)

.HasForeignKey(p => p.IdFilms);

modelBuilder.Entity<FilmsUsers>()

.HasOne(p => p.User)

.WithMany(t => t.FilmsUsers)

.HasForeignKey(p => p.IdUser);

modelBuilder.Entity<FilmsGenres>()

.HasOne(p => p.Films)

.WithMany(t => t.FilmsGenres)

.HasForeignKey(p => p.IdFilms);

modelBuilder.Entity<FilmsGenres>()

.HasOne(p => p.Genres)

.WithMany(t => t.FilmsGenres)

.HasForeignKey(p => p.IdGenres);

modelBuilder.Entity<Description>().HasData(

new Description

{

Id = 1,

DescriptionText = "Best film",

Author = "Husmant"

},

new Description

{

Id = 2,

DescriptionText = "Almost best film",

Author = "Husmant2"

});

modelBuilder.Entity<Films>().HasData(

new Films

{

Id = 1,

NameFilm = "Hellowin",

ReleaseData = new DateTime(1979, 04, 25),

Country = "USA",

FKDescriptionId = 1,

},

new Films

{

Id = 2,

NameFilm = "Strangers",

ReleaseData = new DateTime(2021, 10, 26),

Country = "Ukraine",

FKDescriptionId = 2,

});

modelBuilder.Entity<User>().HasData(

new User

{

Id = 1,

UserName = "Nikolay",

IsAdmin = true,

Password = "1234",

},

new User

{

Id = 2,

UserName = "Vasya",

IsAdmin = false,

Password = "111",

});

modelBuilder.Entity<FilmsUsers>().HasData(

new FilmsUsers

{

IdFilms = 1,

IdUser = 2

},

new FilmsUsers

{

IdFilms = 2,

IdUser = 2

});

}

}

}

## FilmsController

using EFCoreCodeFirstSampleWEBAPI.BLL.DataTransferObjects;

using EFCoreCodeFirstSampleWEBAPI.BLL.Exceptions;

using EFCoreCodeFirstSampleWEBAPI.BLL.Exceptions.Abstract;

using EFCoreCodeFirstSampleWEBAPI.BLL.Interfaces;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using System.Threading.Tasks;

namespace EFCoreCodeFirstSampleWEBAPI.Controllers

{

[Route("api/films")]

[ApiController]

public class FilmsController : ControllerBase

{

private readonly IServiceManager \_serviceManager;

public FilmsController(IServiceManager serviceManager) => \_serviceManager = serviceManager;

// GET: api/Films

[HttpGet]

[AllowAnonymous]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status500InternalServerError)]

public async Task<IActionResult> GetAll()

{

try

{

var Result = await \_serviceManager.FilmsService.GetAll();

return Ok(Result);

}

catch (System.Exception ex)

{

return StatusCode(500, ex.Message);

}

}

// GET: api/Films/5

[HttpGet("{id}", Name = "FilmById")]

[AllowAnonymous]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

[ProducesResponseType(StatusCodes.Status500InternalServerError)]

public async Task<IActionResult> GetById(int id)

{

try

{

var Result = await \_serviceManager.FilmsService.GetById(id);

return Ok(Result);

}

catch (FilmsNotFoundException)

{

return NotFound("No item found with index " + id);

}

catch (System.Exception)

{

return StatusCode(500, "Internal server error");

}

}

[HttpGet("Spec/{id}", Name = "FilmByIdSpec")]

public async Task<IActionResult> GetByIdSpec(int id)

{

try

{

var Result = await \_serviceManager.FilmsService.GetByIdSpec(id);

return Ok(Result);

}

catch (FilmsNotFoundException)

{

return NotFound("No item found with index " + id);

}

catch (System.Exception)

{

return StatusCode(500, "Internal server error");

}

}

// GET: api/Films/5/desc

[HttpGet("{id}/desc")]

[AllowAnonymous]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

[ProducesResponseType(StatusCodes.Status500InternalServerError)]

public async Task<IActionResult> GetWithDetailsById(int id)

{

try

{

var Result = await \_serviceManager.FilmsService.GetWithDetailsById(id);

return Ok(Result);

}

catch (FilmsNotFoundException)

{

return NotFound("No item found with index " + id);

}

catch (System.Exception)

{

return StatusCode(500, "Internal server error");

}

}

// POST: api/Films

[HttpPost]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

[ProducesResponseType(StatusCodes.Status500InternalServerError)]

public async Task<IActionResult> Post([FromBody] FilmsForCreationDto filmsDto)

{

try

{

var filmsDtoPrint = await \_serviceManager.FilmsService.Post(filmsDto);

return CreatedAtRoute(

"FilmById",

new { Id = filmsDtoPrint.Id },

filmsDtoPrint);

}

catch (BadRequestException ex)

{

return BadRequest(ex.Data);

}

catch (System.Exception)

{

return StatusCode(500, "Internal server error");

}

}

// PUT: api/Films/5

[HttpPut("{id}")]

[ProducesResponseType(StatusCodes.Status200OK)]

[ProducesResponseType(StatusCodes.Status400BadRequest)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

[ProducesResponseType(StatusCodes.Status500InternalServerError)]

public async Task<IActionResult> Put(int id, [FromBody] FilmsForCreationDto filmsDto)

{

try

{

if (!ModelState.IsValid)

{

return BadRequest("Invalid model object");

}

await \_serviceManager.FilmsService.Put(id, filmsDto);

return NoContent();

}

catch (FilmsNotFoundException)

{

return NotFound("No item found with index " + id);

}

catch (BadRequestException ex)

{

return BadRequest(ex.Data);

}

catch (System.Exception)

{

return StatusCode(500, "Internal server error");

}

}

// DELETE: api/Films/5

[HttpDelete("{id}")]

[ProducesResponseType(StatusCodes.Status204NoContent)]

[ProducesResponseType(StatusCodes.Status404NotFound)]

[ProducesResponseType(StatusCodes.Status500InternalServerError)]

public async Task<IActionResult> Delete(int id)

{

try

{

await \_serviceManager.FilmsService.Delete(id);

return NoContent();

}

catch (FilmsNotFoundException)

{

return NotFound("No item found with index " + id);

}

catch (System.Exception)

{

return StatusCode(500, "Internal server error");

}

}

}

}