Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система створення соціальної мережа для синефілів
(тема)

Виконав:
здобувач 4 року навчання,
групи ПЗПІ-21-2

Михайло ТКАЧЕНКО
(власне ім'я, прізвище)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник доц. кафедри ПІ Олександр ВЕЧУР
(посада, власне ім'я, прізвище)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
Кафедра Програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва)
Тип програми освітньо-професійна
Освітня програма Програмна Інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Ткаченко Михайлу Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Програмна система створення соціальної мережа для синефілів. затверджена наказом по університету від № 397Ст від 19.05.2025
2. Термін подання студентом роботи до екзаменаційної комісії 12.06.2025р.
3. Вихідні дані до роботи Розробити серверну частину веб-системи соціальної мережі для синефілів з можливістю контентної та колаборативної рекомендацій. Використовувати: MacOS 15.5, платформу .NET, мову C#, фреймворк ASP.NET Core, СУБД PostgreSQL, хмарний сервіс Azure та Superbase, середовище розробки JetBrains Rider 2025.
4. Перелік питань, що потрібно опрацювати в роботі

Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування програмного забезпечення, висновки, перелік джерел посилання, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	20.05.2025	*виконано*
2	Створення специфікації ПЗ	22.05.2025	*виконано*
3	Проектування ПЗ	24.05.2025	*виконано*
4	Розробка ПЗ	28.05.2025	*виконано*
5	Тестування ПЗ	30.05.2025	*виконано*
6	Оформлення пояснювальної записки	05.06.2025	*виконано*
7	Підготовка презентації та доповіді	06.06.2025	*виконано*
8	Попередній захист	23.06.2025	*виконано*
9	Нормоконтроль, рецензування	23.06.2025	*виконано*
10	Здача роботи у електронний архів	23.06.2025	*виконано*
11	Допуск до захисту у зав. кафедри	25.06.2025	*виконано*

Дата видачі завдання 16 травня 2025р.

Здобувач _____ Михайло ТКАЧЕНКО
(підпис)

Керівник роботи _____ доц. кафедри ПІ Олександр ВЕЧУР
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра, 69 сторінок, 41 рисунків, 12 джерел.

КІНО, СОЦІАЛЬНА МЕРЕЖА, РЕЦЕНЗІЇ, РЕКОМЕНДАЦІЇ, УКРАЇНОМОВНИЙ КОНТЕНТ, ВЕБ-СИСТЕМА, C#, .NET 8, ASP.NET CORE, POSTGRESQL, MODULAR MONOLITH, CQRS

Об’єкт розробки – серверна частина веб-системи «Movie Captain» для створення україномовної соціальної платформи кінолюбителів з функціями рецензій, взаємодії між користувачами та генерації персоналізованих рекомендацій.

Мета розробки – розробка програмної системи створення соціальної мережі для синефілів. Програмна система є повноцінним програмним інструментом для об’єднання українських глядачів у межах однієї платформи, забезпечення можливості створення контенту, коментування, оцінювання фільмів, додавання друзів, а також перегляду рекомендацій на основі вподобань і соціальних зв’язків.

Метод рішення – використання мови програмування C# та платформи .NET 8, фреймворку ASP.NET Core для створення RESTful API, архітектурного підходу Modular Monolith з реалізацією патерну CQRS, використання реляційної бази даних PostgreSQL (Supabase) та розгортання через хмарний сервіс Azure App Service.

У результаті створено серверну частину веб-застосунку, яка реалізує описаний функціонал, включаючи структуру бази даних, систему взаємодії з користувачем, рекомендаційний механізм, а також документацію з архітектурою та діаграмами.

ABSTRACT

CINEMA, SOCIAL NETWORK, REVIEWS, RECOMMENDATIONS, UKRAINIAN CONTENT, WEB SYSTEM, C#, .NET 8, ASP.NET CORE, POSTGRESQL, MODULAR MONOLITH, CQRS

Object of development – server-side part of the «Movie Captain» web system designed to create a Ukrainian-language social platform for film lovers with functionality for reviews, user interaction, and personalized recommendation generation.

Purpose of development – to deliver a comprehensive software solution that unites Ukrainian film viewers on one platform, enabling them to create content, comment, rate films, add friends, and receive recommendations based on their preferences and social connections.

Solution method – application of the C# programming language with the .NET 8 platform, ASP.NET Core framework for RESTful API development, Modular Monolith architecture with CQRS pattern, PostgreSQL (Supabase) as the relational database, and deployment via Azure App Service cloud hosting.

As a result, a backend part of the web application has been developed, implementing the described functionality including the database structure, user interaction system, recommendation engine, and full architectural documentation.

ЗМІСТ

ВСТУП

У сучасному цифровому середовищі платформи для спілкування навколо кіно займають важливе місце в культурному житті користувачів. Проте більшість із них орієнтовані на глобальну, переважно англомовну аудиторію, і не враховують локальні особливості, зокрема потреби україномовних глядачів. Українському користувачу бракує зручного, локалізованого інструменту для ведення рецензій, взаємодії з іншими кінофілами, обговорення вражень та отримання персоналізованих рекомендацій у звичному для нього мовному та культурному середовищі.

У межах кваліфікаційної роботи було розроблено серверну частину веб-системи Movie Captain, яка покликана стати соціальною платформою для українських шанувальників кіно. Основною метою проєкту є створення функціонального, масштабованого та захищеного API, що дозволяє користувачам створювати власні блогові публікації, коментувати та оцінювати фільми, формувати коло спілкування, а також отримувати рекомендації на основі власних вподобань та активності друзів.

Розробка здійснювалася із застосуванням сучасних технологій: мова програмування C#, платформа .NET 8, фреймворк ASP.NET Core, архітектурний підхід Modular Monolith із використанням патерну CQRS, реляційна база даних PostgreSQL на Supabase та хмарне розгортання через Azure App Service. Під час виконання роботи було також створено повний набір документації, включаючи UML-діаграми, структуру бази даних та опис архітектури системи.

Результати кваліфікаційної роботи підтверджують можливість реалізації локалізованих платформ із високим рівнем персоналізації, що відповідають сучасним вимогам до зручності, безпеки та продуктивності вебсервісів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

У сучасному інформаційному середовищі кіномистецтво продовжує займати вагоме місце в культурному житті суспільства. Глядачі активно обговорюють фільми в соціальних мережах, діляться враженнями, читають рецензії та шукають рекомендації. Тим не менш, український користувач залишається недостатньо представленим у цьому просторі. Більшість платформ, які існують на ринку, або не мають україномовної локалізації, або орієнтовані на глобальну аудиторію без урахування локального контенту – українських фільмів, серіалів, режисерів, акторів.

Культурна специфіка, мовна ідентичність, та попит на авторський підхід до контенту формують потребу у створенні окремої соціальної платформи, яка б забезпечувала функціональну та емоційну залученість саме україномовних кінолюбителів. Це має бути місце, де користувач може не лише прочитати загальні рейтинги чи короткі коментарі, а створити власний блог, написати повноцінну рецензію, вести дискусію в ком’юніті й отримувати рекомендації, засновані на реальних інтересах.

1.2 Виявлення проблем

Після дослідження потреб цільової аудиторії та аналізу наявних рішень було виявлено такі ключові проблеми:

- відсутність україномовної платформи – більшість існуючих сервісів не пропонують повноцінного інтерфейсу українською мовою;
- неможливість створення авторського контенту – платформи часто обмежуються коментарями чи оцінками, без можливості публікувати аналітичні статті чи рецензії;
- слабка соціальна взаємодія – рідко реалізована модель «друг-друг», особисті стрічки новин, або тематичні спільноти;

– інфраструктурна складність – нові платформи часто не мають зручних API, не масштабуються та не мають захисту, що ускладнює інтеграцію і розвиток;
– відсутність інтелектуальних рекомендацій – пропозиції на основі популярності не враховують інтересів конкретного користувача, на відміну від сучасних підходів колаборативної та контентної фільтрації.

1.3 Аналіз існуючих рішень

На сучасному цифровому ринку існує велика кількість платформ, присвячених фільмам, серіалам і спільнотам кінолюбителів. Проте жодне з цих рішень не враховує специфіку української аудиторії або не забезпечує повноцінну інтеграцію соціальних функцій, орієнтованих на обмін думками та рекомендаціями у локалізованому контексті.

IMDb (Internet Movie Database) [1] є найбільшою міжнародною базою даних про фільми, серіали, акторів і знімальні групи. Платформа надає потужний інструментарій для пошуку фільмів за різними критеріями, підтримує оцінювання та базові коментарі. Втім, її основний фокус – зберігання енциклопедичної інформації, а не соціальна взаємодія. Інтерфейс повністю англомовний, україномовний контент не пріоритетний, а можливість створювати блоги, особисті підбірки з розгорнутими описами чи вести діалоги з іншими користувачами – практично відсутня.

Letterboxd [2] – сучасна англомовна соціальна мережа для кіноманів, яка дозволяє створювати особисті добірки, вести щоденник переглядів, публікувати рецензії, лайкати та коментувати контент інших користувачів. Платформа активно використовується на Заході, особливо серед англомовних користувачів, однак вона не має української локалізації, не пропонує українських добірок або тематичних українських спільнот. Український користувач часто змушений використовувати транслітерацію або англійську мову для інтеграції в англомовне середовище, що послаблює культурну ідентичність.

Facebook-групи та Reddit-сабредіти, присвячені фільмам, є ще одним варіантом для обговорення. Проте вони не структуровані під цільову тематику: часто теми змішані, немає централізованого механізму оцінювання чи рекомендацій. Такі спільноти існують у вигляді потокових обговорень, що не дозволяє ефективно зберігати та фільтрувати корисний контент. Крім того, вони не інтегруються з базами фільмів і не дозволяють створювати власні профілі активності кіноглядача (наприклад, історію переглядів, персональні добірки, блоги тощо).

Жодне з наведених рішень не надає можливості формування персоналізованої стрічки рекомендацій, яка враховує вподобання користувача, соціальні зв’язки (наприклад, уподобання друзів), або локальний контекст — українські фільми, серіали, режисерів, акторів.

Таким чином, можна зробити висновок, що існуючі рішення:

- надто вузькі за функціоналом (наприклад, IMDb як інформаційна енциклопедія);
- універсальні, але не мають глибокої адаптації під потреби конкретної аудиторії (як Letterboxd, Reddit чи Facebook);
- не пропонують достатньої соціальної складової, що робить платформу інструментом перегляду, а не середовищем комунікації та творчості.

Усе це створює передумови для розробки локалізованого, сучасного та функціонального сервісу, який поєднує в собі сильні сторони вищенаведених рішень, одночасно компенсуючи їхні ключові недоліки в контексті українського кінопростору.

1.4 Постановка задачі

Програмна система буде побудована з урахуванням сучасних архітектурних принципів, які дозволяють досягти гнучкості, масштабованості та розділення відповідальностей у коді. Основним підходом є побудова Modular Monolith [3] — архітектурного стилю, що поєднує в собі переваги монолітної структури (простота розгортання, ефективність при невеликій команді) з логічною декомпозицією на окремі функціональні модулі.

Програмна система буде розділена на такі основні модулі:

- identity module – обробляє профілі користувачів, соціальні зв'язки (додавання друзів), відповідає за реєстрацію, логін, JWT-автентифікацію;
- content module – управління постами, коментарями, оцінками;
- recommendation module – надає рекомендації на основі інтересів користувача.

Для розмежування запитів і змін даних буде реалізовано паттерн CQRS [4].

Це означає, що:

- команди (commands) відповідають за зміну стану (наприклад, створення поста, оновлення профілю);
- запити (queries) – лише для читання даних без побічних ефектів (наприклад, отримання списку блогів чи рекомендацій).

Це дозволяє:

- легше масштабувати читання та запис окремо;
- оптимізувати читання для конкретних сценаріїв;
- підвищити контроль за логікою бізнес-процесів.

У системі буде реалізована внутрішня публікація подій (Event Publishing) у рамках патернів DDD (Domain-Driven Design) [5] та event-driven communication між модулями.

Події будуть використовуватись в асинхронному режимі через event dispatcher, що дозволяє уникати жорсткого зв'язування між модулями та спрощує підтримку.

Зважаючи на відокремлення модулів та використання подій, система не гарантуватиме миттєву узгодженість усіх даних у всіх частинах платформи. Натомість буде реалізована кінцева узгодженість (Eventual Consistency) [6]:

- після створення/зміни об’єкта, пов’язані компоненти оновлюються з деякою затримкою;
- читачі можуть короткочасно бачити неактуальні дані, проте загалом система підтримує цілісність.

Це дозволяє масштабувати систему без необхідності транзакцій між модулями, спрощує підтримку незалежного зростання навантаження на окремі функціональні частини (наприклад, читання блогів чи рекомендацій).

Інші архітектурні рішення:

- dependency injection (DI) – для ізоляції компонентів та легкого тестування;
- validation pipeline – валідація вхідних команд перед обробкою;
- mapping layer (наприклад, з використанням AutoMapper) – для перетворення між DTO та доменними об'єктами;
- data Access layer – через ORM (Entity Framework Core [7]) із чітким розділенням доменної логіки та доступу до бази.

Для створення API використовується ASP.NET Core Web API, з подальшим розгортанням на Azure App Service. У якості СУБД обрано PostgreSQL [8], а для спрощення інфраструктури — хостинг на платформі Supabase, яка також пропонує REST та Realtime API для зручності розробки.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Кваліфікаційна робота передбачає створення сучасної веборієнтованої соціальної платформи для україномовних шанувальників кіно під назвою «Movie Captain». Для забезпечення цілісності розробки та відповідності очікуванням користувачів і адміністрації, на цьому етапі сформульовано функціональні та нефункціональні вимоги до програмної системи. Усі вимоги базуються на аналізі предметної галузі, очікуваному функціоналі та обраній архітектурі платформи.

2.1 Функціональні вимоги до програмної системи

Функціональні вимоги системи Movie Captain зосереджені виключно на забезпеченні зручної та інформативної взаємодії для звичайних користувачів – кіноглядачів, які прагнуть ділитися своїми враженнями, формувати спільноти за інтересами та отримувати персоналізовані рекомендації. Уся логіка побудована довкола користувацького досвіду: від реєстрації та налаштування профілю до створення унікального контенту та взаємодії з іншими учасниками платформи.

Платформа має забезпечити можливість створення та ведення блогових записів (рецензій), залишення оцінок і коментарів, вподобань, а також підтримку соціальної взаємодії через систему друзів та персоналізовану стрічку.

2.1.1 Функціональні вимоги до програмної системи для звичайних користувачів

Звичайні користувачі платформи — це особи, які реєструються, створюють контент, взаємодіють з іншими користувачами та отримують рекомендації.

Функціональні вимоги до таких користувачів включають можливість реєстрації нового облікового запису, авторизацію за допомогою JWT токенів, зміну пароля та відновлення доступу до облікового запису. Користувачі можуть

редагувати особисту інформацію у своєму профілі, переглядати власні та чужі профілі, а також додавати інших користувачів до списку друзів або підписок.

Також вони можуть створювати блогові пости — зокрема рецензії, аналітичні матеріали та особисті думки щодо фільмів — з можливістю їх редагування або видалення.

Крім того, передбачена функціональність для створення, редагування та видалення інформації про фільми.

Платформа дозволяє користувачам коментувати публікації, оцінювати фільми за 5-бальною шкалою, а також ставити вподобання під постами й коментарями.

Для зручної навігації реалізовано пошук фільмів, користувачів та постів, з можливістю фільтрації публікацій за жанрами та рейтингом.

Система рекомендацій формує персоналізовані стрічки на основі колаборативної та контентної фільтрації.

Особлива увага приділяється підтримці українського контенту — акцент зроблено на просування українських фільмів і серіалів.

2.1.1 Функціональні вимоги до програмної системи для адміністраторів

Функціональні вимоги до програмної системи для адміністраторів передбачають модерацію контенту. Адміністратори мають змогу створювати фільми, оновлювати інформацію про них або видаляти їх із платформи.

2.2Нефункціональні вимоги до програмної системи

а) продуктивність:
   1) система має підтримувати одночасну роботу не менше 1000 активних користувачів без помітного погіршення швидкодії.

б) безпека:
   1) використання JWT токенів для авторизації;

2) захист від XSS, CSRF, SQL Injection [9].

в) масштабованість:

1) можливість горизонтального масштабування при зростанні кількості користувачів;
2) підтримка кешування та шардингу в майбутньому.

г) доступність та надійність:

1) безвідмовна робота 99.5% часу на місяць;
2) регулярне резервне копіювання даних.

д) розгортання та хостинг:

1) хостинг бекенду на Azure App Service;
2) зберігання та обробка даних у PostgreSQL на Supabase;
3) CI/CD через GitHub Actions або Azure Pipelines.

3 АРХІТЕКТУРА ТА ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML проектування програмного забезпечення

3.1.1 Розробка UML діаграми прецедентів

Першим етапом проєктування програмної системи Movie Captain стало створення UML-діаграми прецедентів (рис. Г.1 – Г.3). Діаграма була побудована з урахуванням функціональних потреб ключових груп користувачів та логіки взаємодії з системою.

Основним актором виступає користувач — зареєстрований учасник платформи, який має доступ до всіх базових можливостей системи після проходження авторизації. Авторизація є обов’язковою умовою для виконання більшості дій у системі, включаючи створення та взаємодію з контентом. Також система включає адміністративну функціональність, що передбачається, але не деталізується окремим актором на цій діаграмі.

Користувач має змогу переглядати бібліотеку фільмів, створювати рецензії, редагувати та видаляти їх, а також переглядати та додавати коментарі до контенту. Реалізована можливість залишати реакції на рецензії та коментарі, з подальшим переглядом власної активності.

Окрему групу функцій становить робота з обраними фільмами: користувач може додавати фільми в обране, переглядати їх, а також видаляти з цього списку. За допомогою функціоналу додавання та видалення друзів формується базова соціально-рекомендаційна складова системи.

Для персоналізації вмісту користувачу доступна можливість отримання рекомендацій, які генеруються на основі його списку вподобань. Також реалізована можливість керування профілем, зокрема редагування персональних даних.

Усі дії користувача тісно пов’язані з авторизацією — більшість прецедентів мають залежність <<include>> від базового процесу входу в систему. Допоміжні дії, такі як редагування, перегляд або видалення контенту, розширюють основні прецеденти через зв’язки типу <<extend>>.

Окремим актором у системі виступає адміністратор, який володіє розширеними правами доступу для управління контентом. Адміністратор має можливість створювати, оновлювати та видаляти записи фільмів. Таке розділення ролей забезпечує підтримання якості контенту та ефективне модераторське управління платформою.

3.1.2 Розробка UML діаграми станів

Для моделювання поведінки системи у процесі формування персоналізованих рекомендацій було розроблено UML-діаграму станів (рис. 3.1). Ця діаграма описує логіку переходів між ключовими етапами аналізу користувацької активності, що лежить в основі системи рекомендацій у Movie Captain.

Процес розпочинається з перевірки наявності доданих користувачем фільмів до обраного списку. Якщо такі фільми відсутні, система повертається до початкового стану або пропонує базові списки фільмів. Якщо ж фільми в обраному є – запускається перша хвиля генерації рекомендаційного рейтингу на основі контенту.

Наступним етапом є перевірка наявності друзів користувача. У випадку, якщо друзі відсутні, система формує список нових фільмів та список популярних фільмів – це дозволяє запропонувати щось загальнодоступне, навіть за мінімальної активності користувача. Якщо ж друзі додані, виконується додаткова генерація рекомендаційного рейтингу на основі взаємодій цих користувачів – що реалізує принцип колаборативної фільтрації.

Фінальним кроком є формування списку рекомендацій, які згодом відображаються користувачу на головній сторінці платформи.

Таким чином, діаграма станів чітко ілюструє послідовність логічних перевірок і дій у системі, відображаючи гнучкість алгоритму рекомендацій у залежності від рівня користувацької взаємодії. Це дозволяє адаптувати результати

під індивідуальні сценарії використання, забезпечуючи динамічний, персоналізований досвід.

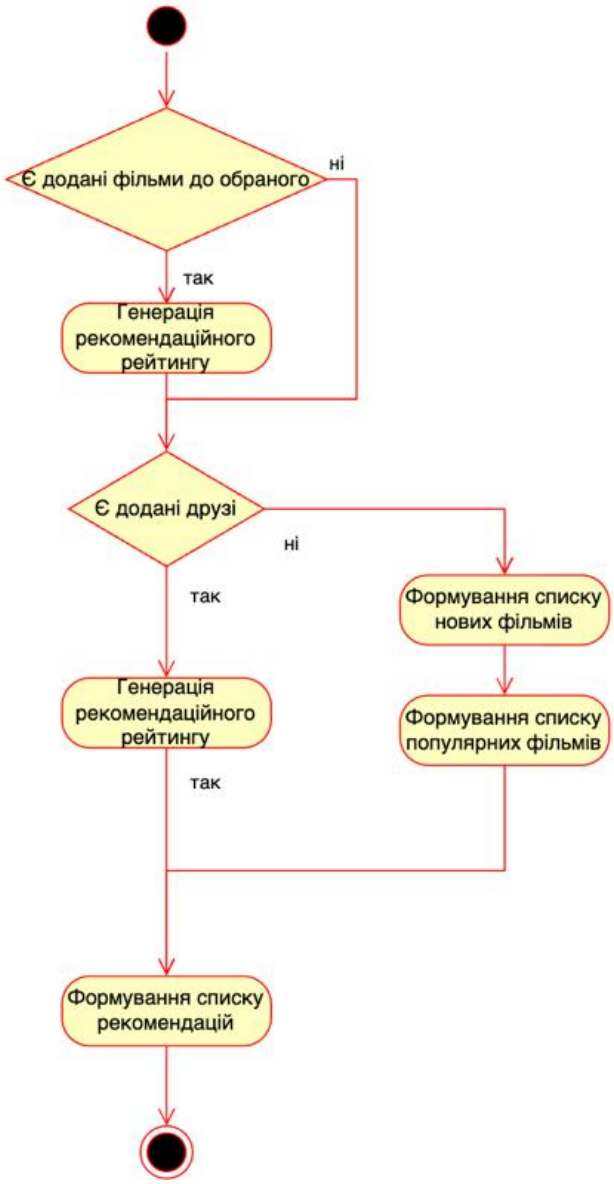

Рисунок 3.1 – UML діаграма станів (виконано самостійно)

3.2 Проектування архітектури програмного забезпечення

Архітектура програмної системи Movie Captain розроблена відповідно до принципів модульного моноліту із використанням сучасних хмарних рішень та

структурної ізоляції логіки за функціональними доменами. У центрі архітектурної моделі — принципи розділення відповідальностей, масштабованості та безпечної взаємодії між компонентами. Структура проєкту подана у вигляді діаграми розгортання на рис. 3.2.

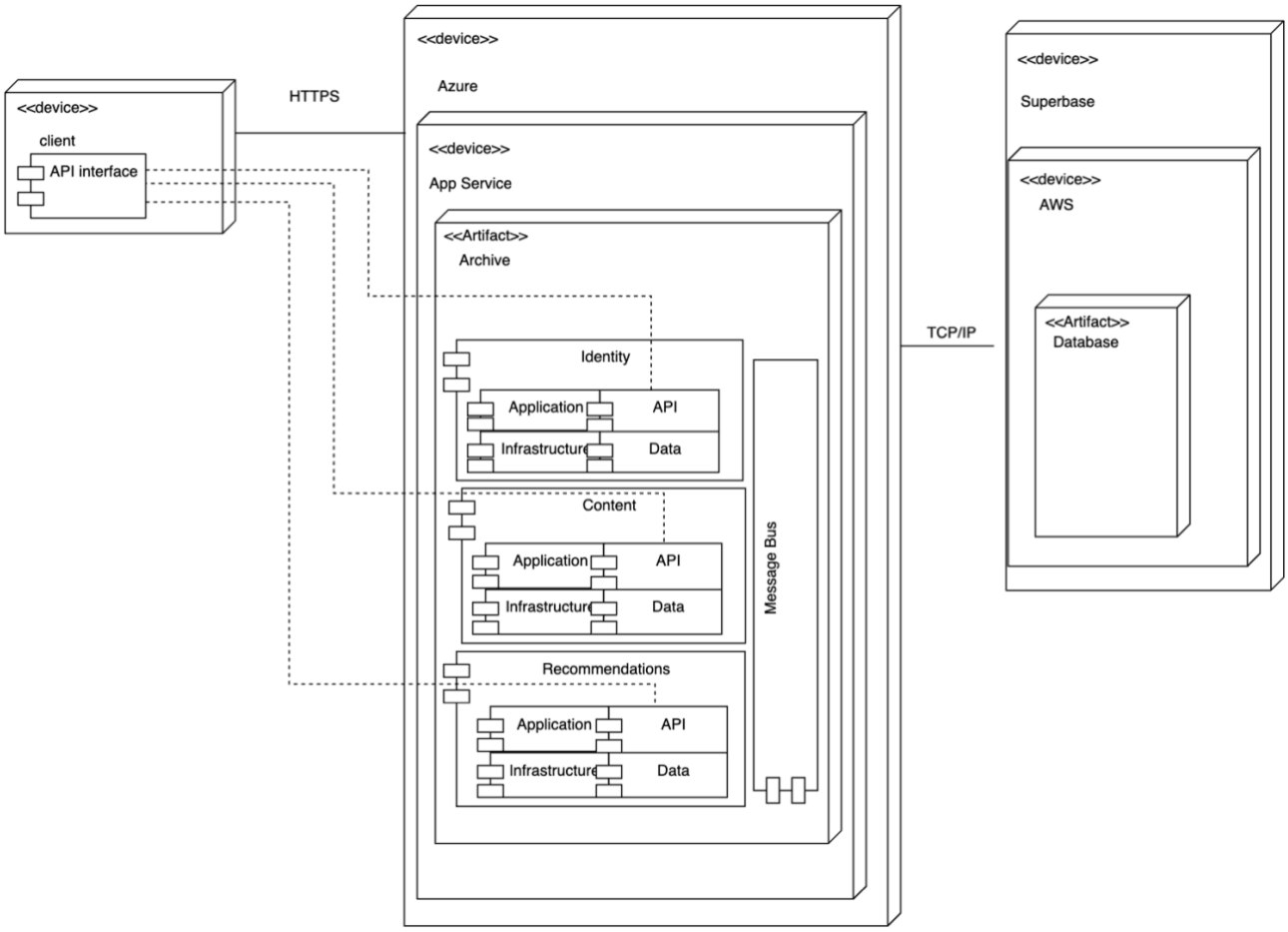

Рисунок 3.2 – Діаграма розгортання (виконано самостійно)

Система розгортається на хмарну платформу Azure як App Service [10], яка виступає хостом для всієї бекенд-частини. Вона включає в себе логічні модулі, кожен з яких структурований за принципом Clean Architecture [11]:

- Identity – відповідає за автентифікацію, авторизацію, управління обліковими записами користувачів;
- Content – обробляє публікації рецензій, коментарів, реакцій;
- Recommendations – відповідає за генерацію рекомендаційного рейтингу та формування персоналізованих списків фільмів.

Кожен модуль побудований згідно з розділенням на шари:

- Application – містить бізнес-логіку, CQRS-команди та обробники;
- Infrastructure – реалізує зовнішні інтеграції, доступ до даних;
- API – відповідає за прийом HTTP-запитів;
- Data – визначає доменні об'єкти.

Між модулями реалізована взаємодія через Message Bus (шину повідомлень), яка забезпечує передачу подій у відповідності до патерну event-driven communication та підтримує eventual consistency без необхідності прямої залежності між модулями.

Клієнтська частина взаємодіє із системою через відкритий API Interface по протоколу HTTPS.

Дані зберігаються в реляційній базі даних PostgreSQL, хостинг якої здійснюється через Supabase, що розміщений на інфраструктурі AWS. З'єднання між Azure App Service та Supabase здійснюється через захищений канал TCP/IP.

Запропонована архітектура дозволяє масштабувати окремі функціональні модулі, підтримує логічну ізоляцію бізнес-контекстів і забезпечує стабільну основу для розширення проєкту в майбутньому.

3.3 Проектування структури зберігання даних

Проєктування структури зберігання даних системи Movie Captain базується на використанні реляційної бази даних PostgreSQL. Основною метою структурування даних було забезпечення логічної відповідності між бізнес-об’єктами платформи, оптимізація запитів, а також підтримка гнучкої та масштабованої схеми для зростаючого обсягу контенту та взаємодії користувачів.

На рисунку 3.3 зображено ER-діаграму, що ілюструє зв’язки між основними сутностями.

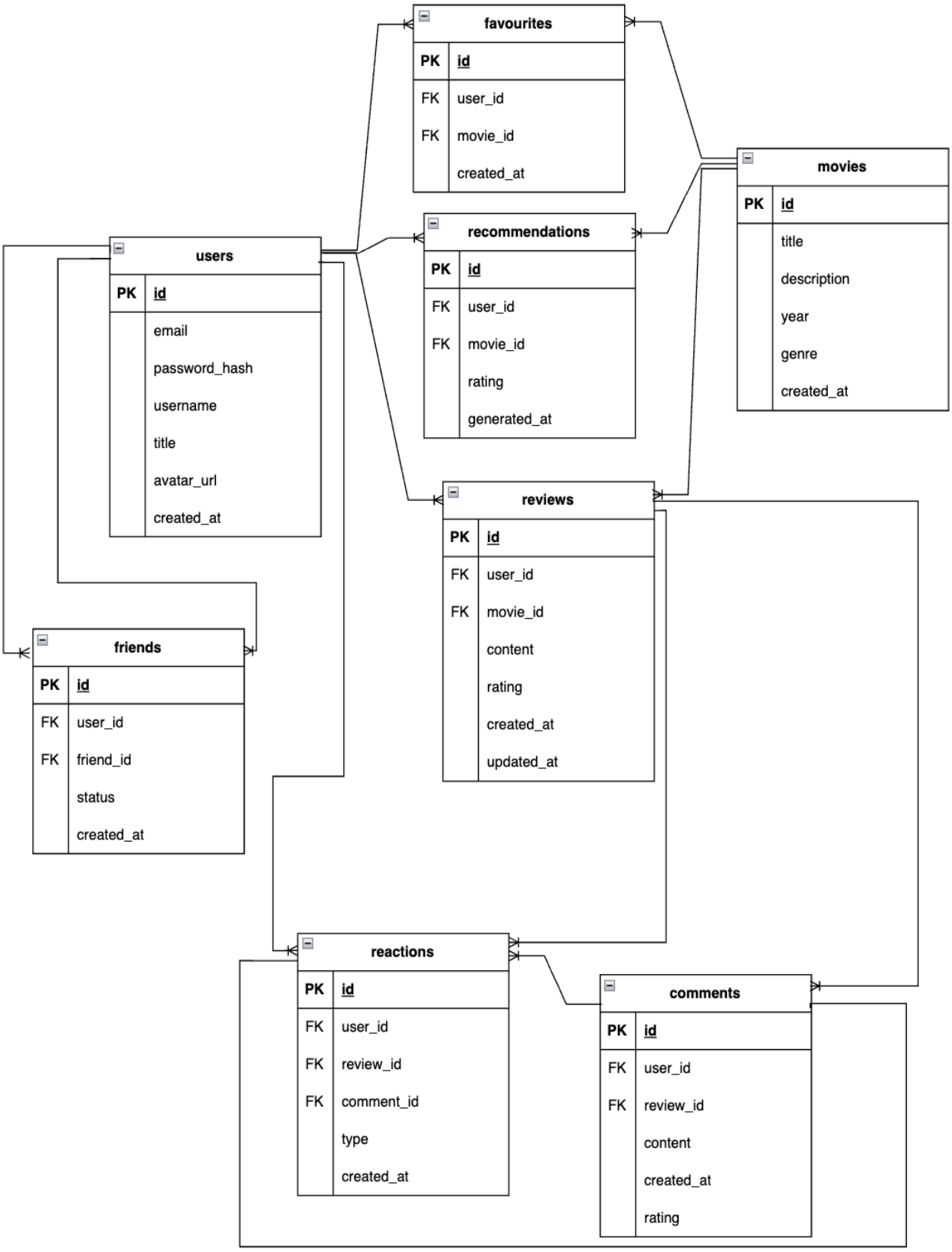

Рисунок 3.3 – ER-діаграма (виконано самостійно)

У центрі структури знаходиться таблиця users, яка пов’язана з іншими сутностями через зовнішні ключі. Кожен користувач може створювати reviews (рецензії) на фільми з таблиці movies, залишати comments (коментарі) до рецензій, а також реагувати (reactions) на рецензії або коментарі інших користувачів.

Функціонал збереження улюблених фільмів реалізовано через таблицю favourites, яка забезпечує зв’язок "багато до багатьох" між users та movies. Подібним чином реалізовано систему friends, яка дозволяє користувачам додавати одне одного до списку друзів. Таблиця recommendations містить персоналізовані рекомендації для кожного користувача на основі його вподобань, взаємодій та активності друзів.

Кожна таблиця містить часові мітки (created_at, updated_at), що дозволяє відслідковувати зміни та забезпечує можливість аудиту. Всі зовнішні ключі проіндексовано для покращення продуктивності при виконанні запитів.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Загальні відомості

Для розробки серверної частини програмноиї системи використовувалась середа розробки JetBrains Rider для написання серверного коду, і Docker для контейнеризації брокера повідомлень RabbitMq.

Система MovieCaptain побудована з використанням сучасного технологічного стеку на платформі .NET 8.0 з мовою програмування C# 12.0. В основі архітектури використовується ASP.NET Core для реалізації API та бізнес-логіки, а також MediatR для забезпечення патерну CQRS і обробки запитів та команд. Для взаємодії компонентів системи і підтримки асинхронного обміну повідомленнями використовується MassTransit у поєднанні з RabbitMQ як брокером повідомлень, що забезпечує надійне і масштабоване керування чергами повідомлень.

Збереження даних реалізовано через реляційні бази даних із використанням ORM Entity Framework Core, що дозволяє ефективно працювати з таблицями й автоматизувати міграції, а також використовується сучасна технологія для логування. СУБД була обрана реляційна, PostgreSql, оскільки вона має добру підтримку адаптерів EF Core, легко розгортається, підтримує усі необхідні функції для реляційних СУБД. Для адміністрування та моніторинку СУБД було обрано програму PgAdmin 4 оскільки вона досі активно розвивається і є однією з найпопулярніших в розробці.

4.2 Виклик та завантаження

ASP.NET Core є кросплатформеною платформою, тому запускати проєкт можна з різних середовищ, які підтримують .NET та мову програмування C#. Запускається проєкт за допомогою SDK .NET SDK 8.0, оскільки саме ця версія була обрана для розробки продукту. Проєкт має систему автоміграції даних, тому при чистому запуску всі необхідні схеми даних і таблиці як і початкові дані будуть автоматично занесені в базу даних при першому запуску.

4.3 Структура програмної системи

Система побудована за архітектурним стилем Modular Monolith — це проміжний підхід між класичним монолітом і мікросервісною архітектурою. Основні проекти реалізовано за принципами чистої шарової архітектури (clean layered architecture) (рис. 4.1). Вони не залежать ні від яких інших проектів і мають свої ізольовані схеми бази даних. Під час розгортання вона представлена єдиним артефактом, проте її внутрішня структура організована за принципами мікросервісів. Такий підхід забезпечує високу модульність, ізольованість компонентів і спрощує потенційний перехід до повноцінної мікросервісної архітектури в майбутньому.

Кожен модуль реалізований як набір окремих проєктів:

- основний проєкт, що містить бізнес-логіку, API та доступ до бази даних;
- проєкт із публічними контрактами для міжмодульної взаємодії;
- проєкт із модульними тестами, які покривають функціональність модуля.

Окрім цього, у рішенні присутній спільний допоміжний проєкт із утилітами, які можуть використовуватись у різних модулях, а також стартовий проєкт, що відповідає за загальну конфігурацію сервера. Кожен модуль самостійно виконує власну конфігурацію, що підвищує гнучкість і масштабованість системи.

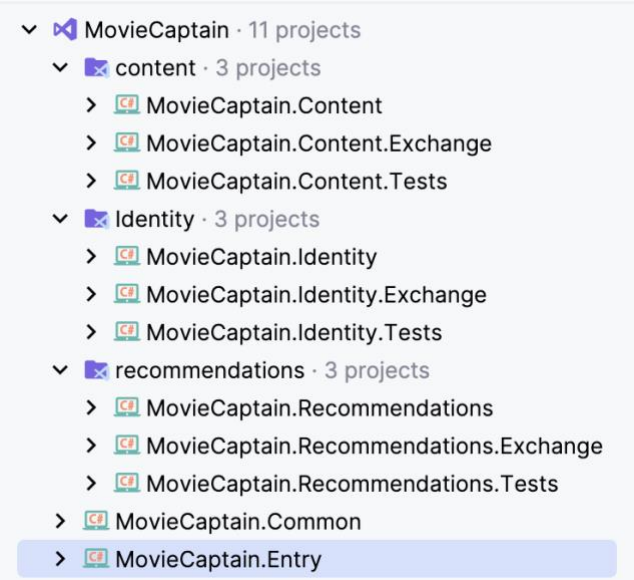

Рисунок 4.1 – Структура системи

У кожному модулі, що містить бізнес-логіку, реалізовано підхід Clean Architecture — це структурна архітектурна модель, яка дозволяє чітко розділити відповідальності між компонентами системи та забезпечує слабке зв’язування між ними. Основна ідея полягає в організації коду у вигляді шарів, де залежності завжди спрямовані від зовнішніх шарів до внутрішніх. Це забезпечує гнучкість, тестованість і легкість у зміні або заміні окремих частин системи (рис. 4.2).

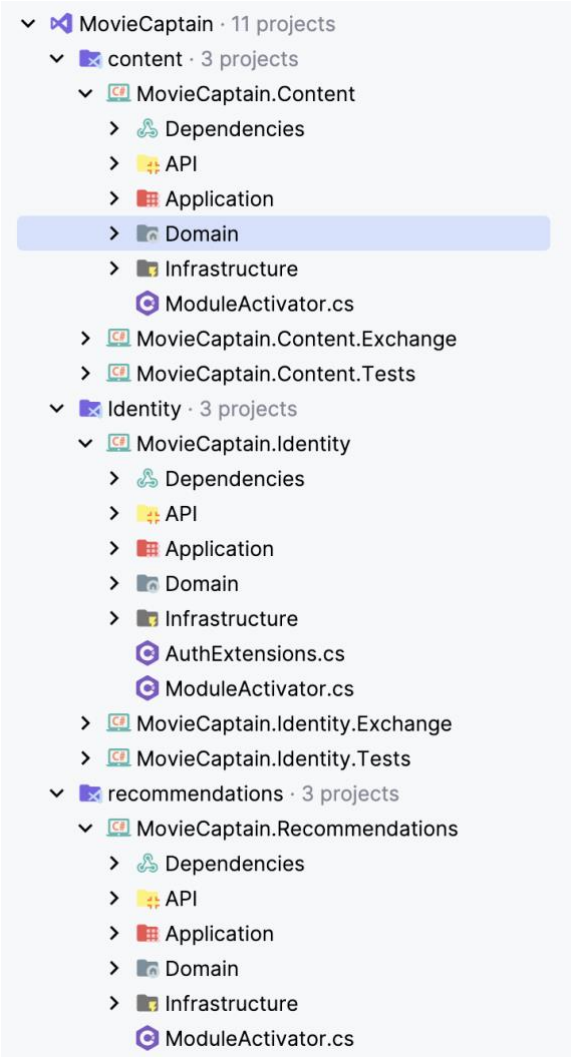

Рисунок 4.2 – Структура основного проекту модуля

Конкретна реалізація включає наступні шари:

- Domain (ядро): містить сутності, бізнес-правила та інтерфейси (контракти), що описують поведінку системи. Цей шар повністю

незалежний від інших і не має жодних залежностей від інфраструктури або фреймворків.

- Application (застосунок): реалізує бізнес-кейси, взаємодіє з доменними сутностями через інтерфейси. Усі сервіси та обробники запитів (наприклад, CQRS-хендлери) знаходяться тут. Він також не залежить від зовнішніх шарів.
- Infrastructure (інфраструктура): містить реалізації зовнішніх залежностей, зокрема доступ до бази даних, API-клієнти, системи логування тощо. Цей шар реалізує інтерфейси, визначені у внутрішніх шарах, і є залежним від них.
- API (веб-шар): відповідає за прийом і обробку HTTP-запитів, мапінг моделей запиту/відповіді та передачу запитів до шару застосунку. Це єдиний шар, що взаємодіє із зовнішнім середовищем (користувачем, мережею).

4.4 Організація взаємодії з зовнішніми сервісами

4.4.1 Організація роботи зі сховищем даних

У якості системи управління базами даних використовується PostgreSQL у поєднанні з Entity Framework Core як ORM-рішенням. Робота з базою даних організована з урахуванням високої модульності системи: кожен функціональний модуль має власну ізольовану схему у базі даних. Такий підхід забезпечує логічну та фізичну роздільність даних, підвищує безпеку, спрощує супровід, і найголовніше — дозволяє в перспективі переносити модулі в окремі фізичні бази даних або мікросервіси без істотних змін у їхній внутрішній структурі.

Міграції виконуються окремо для кожного модуля, що дає змогу розвивати кожну схему незалежно. Жорсткі зовнішні залежності між модулями на рівні БД відсутні — натомість використовується асинхронна міжмодульна комунікація через повідомлення. Кожен модуль самостійно обробляє події, що надходять з інших

модулів, і проєціює необхідні зміни у свою схему (наприклад, зберігає копії необхідних даних у власному форматі).

Такий підхід базується на принципах data ownership та event-driven projections і дозволяє зберігати модулі незалежними як на рівні коду, так і на рівні даних. Він також відкриває шлях до поступового переходу від модульного моноліту до повноцінної мікросервісної архітектури без значних переробок логіки роботи з даними.

4.4.2 Організація роботи з електронною поштою

У системі реалізовано функціональність надсилання електронних листів при виникненні певних подій, що потребують взаємодії з користувачем. Зокрема, одним із таких сценаріїв є запит на зміну пароля.

Для реалізації цієї функціональності використовується бібліотека MailKit, яка є популярним і потужним клієнтом для роботи з електронною поштою в .NET. Вона забезпечує підтримку сучасних протоколів (SMTP, POP3, IMAP), а також дозволяє легко реалізовувати як синхронну, так і асинхронну взаємодію з поштовим сервером.

В рамках реалізації було створено сервіс для формування та відправлення листів, який:

- генерує повідомлення з відповідним шаблоном (наприклад, посиланням на сторінку зміни пароля);
- встановлює SMTP-параметри (хост, порт, автентифікація);
- ініціалізує підключення та виконує надсилання листа через SmtpClient з MailKit.

Це дозволяє забезпечити надійну та безпечну доставку повідомлень до користувачів без залучення сторонніх сервісів.

Надсилання листів відбувається асинхронно, що не блокує виконання основної логіки застосунку і підвищує загальну продуктивність системи.

```
     public async Task
Consume(ConsumeContext<PasswordRecoveryEmailRequestedMessage> context)
{
    PasswordRecoveryEmailRequestedMessage message = context.Message;

    MimeMessage emailMessage = new();
    emailMessage.From.Add(new MailboxAddress("MovieCaptain",
_emailSettings.SenderEmail));
    emailMessage.To.Add(new MailboxAddress("", message.Email));
    emailMessage.Subject = "Password Recovery";
    emailMessage.Body = new TextPart("plain")
    {
        Text = $"Hello,\n\nWe have received a request to reset your
password. You can reset it using the following
token:\n\n{message.ResetToken}\n\nIf you did not request a password reset,
please ignore this message.\n\nThank you."
    };

    using SmtpClient smtpClient = new();
    try
    {
        await smtpClient.ConnectAsync(_emailSettings.SmtpServer,
_emailSettings.Port, SecureSocketOptions.StartTls);
        await smtpClient.AuthenticateAsync(_emailSettings.SenderEmail,
_emailSettings.AppPassword);
        await smtpClient.SendAsync(emailMessage);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Failed to send email: {ex.Message}");
        throw;
    }
    finally
    {
        await smtpClient.DisconnectAsync(true);
        smtpClient.Dispose();
    }
}
```

4.5 Опис програмних рішень серверної частини системи

4.5.1 Автентифікація та авторизація користувачів

Система автентифікації та авторизації реалізована на базі ASP.NET Core Identity з використанням access токенів та refresh токенів, що забезпечує надійне управління сесіями та доступом до ресурсів. Уся логіка ізольована в модулі Identity, який діє як самодостатній компонент системи без прямих залежностей від інших модулів.

Кінцева точка POST /api/identity/auth/register обробляє реєстрацію користувача, делегуючи логіку через CQRS-команду RegisterParticipantCommand. Після успішного створення облікового запису, модуль публікує повідомлення у брокер повідомлень (RabbitMQ), щоб інші модулі могли асинхронно зреагувати на створення нового користувача. Цей підхід дає змогу реалізувати проєкції користувача в інших модулях — тобто кожен модуль, який потребує інформації про користувача, створює власну локальну копію (репліку) даних на основі події. Таким чином забезпечується data ownership, модульність і незалежність між компонентами системи.

а) авторизація та управління токенами:
   1) авторизація реалізується через UserManager — при успішному логіні генеруються access токен і refresh токен;
   2) access токени створюються через сервіс ITokenService, мають обмежений термін дії та використовуються для доступу до захищених ресурсів;
   3) refresh токени зберігаються у базі даних і мають термін придатності. Після кожного оновлення токена (через POST /api/identity/auth/refresh), стара пара токенів замінюється на нову.

б) безпека та масштабованість:
   1) токени захищаються стандартними механізмами ASP.NET Core, включаючи middleware та політики авторизації;

2) розділення на access/refresh токени забезпечує баланс між безпекою та зручністю користувача;
3) архітектура підтримує горизонтальне масштабування і готова до винесення модуля автентифікації в окремий сервіс без істотних змін у клієнтських модулях.

4.5.2 Система дружби

У системі MovieCaptain реалізовано повноцінну підсистему дружби, яка дозволяє користувачам взаємодіяти між собою у соціальний спосіб. Основна мета — забезпечити можливість формування зв'язків між користувачами з подальшою інтеграцією цих зв'язків у додаткові сервіси, а саме рекомендації. Процес починається із надсилання запиту на дружбу. Для цього автентифікований користувач може через відповідний API (/api/identity/friends/invite) ініціювати запит, вказавши ідентифікатор іншого користувача та повідомлення. У відповідь система обробляє команду SendFriendRequestCommand, яка виконує перевірку на існування одержувача та формує об'єкт FriendRequest, що зберігається в базі даних. Після цього надсилається подія FriendRequestIncomingMessage через брокер повідомлень MassTransit, що дозволяє іншим сервісам реагувати на запит — формувати нотифікації або оновлювати локальні проекції даних.

Прийняття дружби здійснюється окремим запитом (/api/identity/friends/accept/{id}), де {id} — це ідентифікатор запиту на дружбу. Ця операція виконується через команду AcceptFriendRequestCommand, яка перевіряє права доступу (чи є поточний користувач одержувачем запиту), додає відправника у список друзів і видаляє запис запиту. Таким чином формується двосторонній зв'язок між користувачами. У разі помилки (наприклад, запит вже був видалений або користувач не є адресатом) викидається відповідний виняток, і операція скасовується.

Користувач може у будь-який момент переглянути свій список друзів. Для цього існує запит GET /api/identity/friends, який повертає список користувачів, що

перебувають у дружніх відносинах із поточним користувачем. Дані подаються у форматі DTO AccountInfo, який містить основну інформацію про користувача: ім’я, прізвище, email, нікнейм, біографію тощо. Якщо список друзів порожній, система повертає 404 Not Found.

У системі також реалізовано можливість видалення друга. Запит DELETE /api/identity/friends/{friendId} дозволяє поточному користувачеві видалити обраного друга за його ідентифікатором. Після перевірки зв’язку у списку друзів виконується його видалення, і якщо операція збереження проходить успішно, публікується подія UserFriendDeletedMessage. Така подія дозволяє рекомендаційному модулю актуалізувати внутрішні зв’язки та припинити доступ до відповідного функціоналу між цими користувачами.

Допоміжно реалізована функція пошуку нових знайомств. Через запит GET /api/identity/people/discover користувач може знайти іншу людину за ім’ям, прізвищем або email.

4.5.3 Керування фільмами

Керування контентом у системі Movie Captain реалізовано відповідно до архітектурного підходу CQRS (Command Query Responsibility Segregation), що забезпечує розділення запитів і команд, а також застосування подієвої моделі для інформування підсистем про зміни у даних. Основними операціями в межах керування фільмами є створення, оновлення та видалення записів про фільми.

Операція створення фільму виконується через команду CreateMovieCommand. Перед створенням перевіряється, чи не існує вже запису з такою ж назвою — це дозволяє уникнути дублювання. Новий фільм включає пов’язану інформацію: жанри, теги, акторський склад, опис, країну походження, дату релізу та ознаку українського походження. Після успішного збереження фільму у базі даних до брокера повідомлень надсилається подія MovieContentCreatedMessage, яка містить усі метадані, необхідні для системи

рекомендацій. Це забезпечує швидку доступність новоствореного фільму в інших частинах платформи.

Для оновлення даних про фільм використовується команда UpdateMovieCommand. Зміни можуть стосуватись будь-яких параметрів, включаючи жанри, опис, країну або дату виходу. Усі оновлення також фіксуються системою подій і надсилають відповідні повідомлення до пов’язаних підсистем, зокрема системи персоналізованих рекомендацій.

Видалення фільмів реалізоване через команду DeleteMovieCommand. Важливою частиною логіки цієї операції є обмеження доступу до повного видалення: якщо на фільм уже існують рецензії або коментарі, звичайний користувач не має права видалити його запис. У такому випадку операція переривається, і користувачу повідомляється про наявність залежностей.

Повне видалення фільму, включно з усіма пов’язаними рецензіями та коментарями, дозволено лише адміністратору. Це забезпечує контроль за цілісністю даних та дозволяє втручатись у випадках, коли контент порушує політику платформи.

Усі команди пов’язані з керуванням фільмами генерують відповідні події (MovieContentCreatedMessage, MovieContentUpdatedMessage, MovieContentDeletedMessage), які відслідковуються брокером повідомлень та використовуються іншими підсистемами для синхронізації змін.

4.5.4 Отримання рекомендацій на основі контенту

Система рекомендацій використовує гібридний підхід, поєднуючи контентно-орієнтовану та колаборативну фільтрацію. Контентна фільтрація базується на аналізі жанрових векторів фільмів, доданих користувачем у «улюблені». Профіль користувача P формується як середнє векторів жанрів вподобаних фільмів:

$$P = \frac{1}{N}\sum_{i=1}^{N} v_i \tag{4.1}$$

де P – профіль користувача;

N — кількість уподобаних фільмів;

v_i — one-hot вектор жанру і-го фільму.

Для рекомендацій обчислюється косинусна подібність між профілем і векторами інших фільмів:

$$sim(p, v) = \frac{p * v}{\|p\| * \|v\|} \tag{4.2}$$

де $sim(p, v)$ – косинусна схожість;

$p * v$ – скалярний добуток векторів p і v;

$\|p\|$ – евклідова норма (довжина) вектора p;

$\|v\|$ – евклідова норма (довжина) вектора v.

Використовуючи формулу 4.1, 4.2, можна побудувати наступну рекомендаційну систему, код якої наведено нижче:

```
public class CosineRecommendationSystem : IContentBasedRecommendationSystem
{
    private double[] BuildUserVector(Viewer viewer, int vectorSize)
    {
        double[] userVector = new double[vectorSize];
        if (viewer.Preferences is null || viewer.Preferences.Count == 0)
            return Enumerable.Repeat(0.0, vectorSize).ToArray();

        foreach (Movie movie in viewer.Preferences)
        {
            for (int i = 0; i < vectorSize; i++)
            {
                userVector[i] = movie.Vector[i];
            }
```

```
        }

        if (viewer.Preferences.Any() is false)
        {
            userVector[^2] = 1.0;
            userVector[^1] = 1.0;
        }

        userVector[^2] *= 1.5;
        userVector[^1] *= 2.0;

        for (int i = 0; i < vectorSize; i++)
        {
            userVector[i] /= viewer.Preferences.Count;
        }

        return userVector;
    }

    public Dictionary<Guid, MovieRecommendationMetadata> GetRelevant(Viewer
viewer, List<Movie> library)
    {
        int vectorSize = library.FirstOrDefault()?.Vector.Length ?? 0;
        double[] userVector = BuildUserVector(viewer, vectorSize);

        HashSet<Guid> seen = viewer.Preferences?.Select(m =>
m.Id).ToHashSet() ?? [];
        List<Movie> candidates = library
            .Where(m => !seen.Contains(m.Id))
            .ToList();

        Dictionary<Guid, MovieRecommendationMetadata> scores =
candidates.ToDictionary(
            movie => movie.Id,
            movie => new MovieRecommendationMetadata(
                Score: VectorUtils.CosineSimilarity(userVector, movie.Vector),
                IsUkrainian: movie.IsUkrainian));
        IEnumerable<MovieRecommendationMetadata> nonZeroScores =
scores.Values.Where(v => v.Score > 0).ToList();
```

```
        double medium = nonZeroScores.Any() ? nonZeroScores.Average(m =>
m.Score) : 1;

        return scores
            .OrderByDescending(kv => kv.Value.Score)
            .ThenBy(kv => kv.Value.IsUkrainian)
            .Where(kv => kv.Value.Score >= medium)
            .ToDictionary(kv => kv.Key, kv => kv.Value);
    }
}
```

Для демонстрації роботи системи спочатку проведемо експеримент, з бази даних було обрано фільм (рис. Г.4), і побудовано його вектор ґрунтуючись на його жанрових і тегових характеристиках, які можна частково побачити (оскільки вектор більше за 120 характеристик). Далі було побудовано вектор користувача грунтуючись на його списку уподобань (рис. 4.3). Після цього за формулою обраховуємо міру подібності. Спочатку, щоб знайти скалярний добуток, перемножаємо елементи з однаковим індексом і сумуємо 0.0 + 0.0 +0.0 + 0.0 + 0.0 + 0.0 + 0.0 + 0.0 + 0.2 + 0.2 +0.0+ 0.0 = 0.4. Далі обчислюємо норму вектора користувача $\sqrt{(0.2)^2 * 8} \approx 0.565$. Таким самим чином знаходимо норму для другого вектора $\sqrt{1^2 * 6} \approx 2.449$. Тепер за формулою 4.2 треба поділити скалярний добуток на добуток норм обох векторів, отримуємо наступне $\frac{0.4}{0.565 * 2.449} = 0.289$, отже два фільми схожі приблизно на 30%. Результат системи збігається з невеликою похибкою, тому можна вважати що система функціонує коректно (рис. 4.4). Чим цей відсоток більший, тим більш релевантний фільм для користувача і тим вище в рейтингу він буде знаходитись. (рис. 4.5). Остаточний результат подібності фільму буде повернено в наступному вигляді (рис. 4.6).

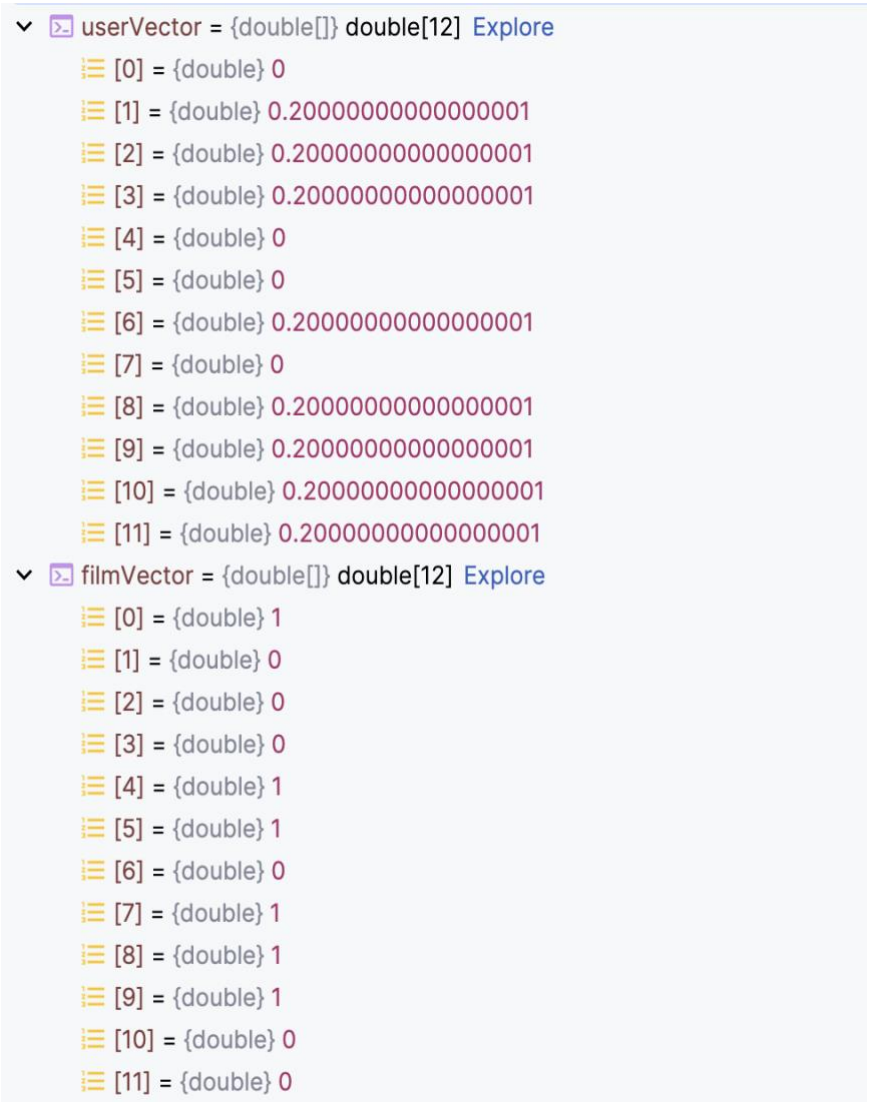

Рисунок 4.3 – Вектори фільму і користувача

```
public static double OptimizedCosineSimilarity(double[] userVector, double[] movieVector)  movieVector: double[122]
{
    List<double> u = [];  u: Count = 12
    List<double> m = [];  m: Count = 12

    for (int i = 0; i < userVector.Length; i++)  userVector: double[122]
    {
        if (userVector[i] == 0.0 && movieVector[i] == 0.0)  movieVector: double[122]    userVector: double[122]
            continue;

        u.Add(userVector[i]);  u: Count = 12    userVector: double[122]
        m.Add(movieVector[i]);  movieVector: double[122]    m: Count = 12
    }

    double similarity = VectorUtils.CosineSimilarity(u.ToArray(), m.ToArray());  similarity: 0.28867513459481287

    return similarity;  similarity: 0.28867513459481287
}
```

Рисунок 4.4 – Приклад розрахунку системи

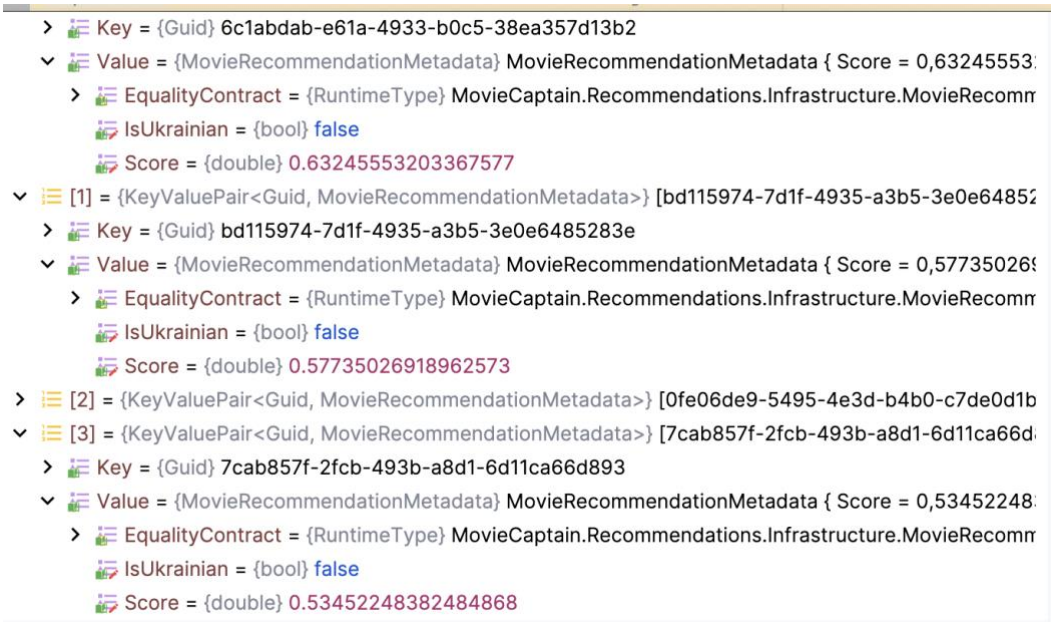

Рисунок 4.5 – Приклад формування списку рекомендацій

Рисунок 4.6 – Результат подібності

4.5.5 Отримання рекомендацій на основі дружніх зв’язків

У системі реалізовано два підходи до колаборативної фільтрації між друзями: на основі уподобань (тобто бінарних дій "додано до улюбленого") та на основі числових рейтингів (наприклад, від 1 до 5 зірок). Обидва підходи враховують лише тих друзів, із якими користувач має щонайменше дві спільні дії (вподобання або оцінки), а також вводять коефіцієнти довіри для зменшення впливу слабких зв’язків.

У фільтрації на основі уподобань кожного користувача представляють множиною фільмів, які він позначив як улюблені. Схожість між користувачем uuu та його другом vvv обчислюється за модифікованим бінарним коефіцієнтом Пірсона, або, якщо такий підхід не дає результату, — за коефіцієнтом Жаккара:

$$Jaccard\ (u, v) = \frac{|F_u \cap F_v|}{|F_u \cup F_v|} \tag{4.3}$$

де F_u, F_v — множини вподобаних фільмів користувачів.

Застосовується також фактор довіри:

$$TrustWeight = \frac{F_u \cap F_v}{F_u} \tag{4.4}$$

У фільтрації на основі рейтингів аналізуються числові оцінки, які користувачі виставляють фільмам. Якщо є щонайменше дві спільні оцінки, то схожість між користувачем і другом обчислюється за коефіцієнтом кореляції Пірсона:

$$sim(u, v) = \frac{\sum_i \left(r_{u,i} - \overline{r_u}\right)\left(r_{v,i} - \overline{r_v}\right)}{\sqrt{\sum_i \left(r_{u,i} - \overline{r_u}\right)^2} * \sqrt{\sum_i \left(r_{v,i} - \overline{r_v}\right)^2}} \tag{4.5}$$

де $sim(u, v)$ – міра схожості між користувачами u та v;

$r_{u,i}, r_{v,i}$ – рейтинги, які користувачі u,v поставили фільму i;

$\overline{r_u}, \overline{r_v}$ – середні рейтинги усіх фільмів, оцінених користувачами u,v.

У випадку відсутності статистично значущої кореляції (формула 4.5) застосовуються коефіцієнти Спірмена або Жаккара. Далі, для фільмів, які друг оцінив, а користувач — ні, формується оцінка релевантності:

$$R_{u,i} = \sum_{v \epsilon Friends(u)} sim(u, v) * r_{v,i} \tag{4.6}$$

де $R_{u,i}$ – оцінка релевантності фільму i для користувача u.

Підсумкова оцінка фільму базується на об’єднанні обох підходів. Компонент CollaborativeRecommendationSystem агрегує оцінки за допомогою вагових коефіцієнтів, наприклад, $w_r = 0.7$ для рейтингового підходу і $w_p = 0.3$ для

уподобань, тому застосовуючи формулу 4.6 і вагові коефіцієнти отримуємо формулу для розрахунку загального ненормалізованого рейтингу:

$$R_{u,i}^{aggregate} = \frac{\sum_{r \epsilon R} w_r \cdot R_{u,i}^r}{\sum_{r \in R} w_r} \quad (4.7)$$

де $R_{u,i}^{aggregate}$ = загальний рейтинг;

i – фільм;

r – алгоритм рекомендації;

w_r – вага алгоритму рекомендації;

$R_{u,i}^r$ – вага схожості фільму і передбачувана алгоритмом для користувача u.

Далі загальний рейтинг фільмів (формула 4.7) передаються аргументом в функцію нормалізації яка нормалізує до діапазону [0;1] за допомогою функції логістичного сігмоїда:

$$R_{u,i}^{final} = \frac{1}{1 + e^{(-\frac{R_{u,i}^{aggregate} - \mu}{\sigma})}} \quad (4.8)$$

де $R_{u,i}^{final} -$ нормований рейтинг $R_{u,i}^{final} \epsilon [0; 1]$

$\mu -$ середнє значення

$\sigma -$ стандартне відхилення

Такий підхід дозволяє враховувати смаки друзів і формувати персоналізовані рекомендації навіть за браком даних у самого користувача.

Проведемо експеримент, візьмемо користувача Anastasya Lysenko та Mykhailo Tkachenko та порахуємо коефіцієнт кореляції спільних інтересів. Для цього робимо запит в базу даних (рис. 4.8) для того, щоб подивитись які спільні фільми було спільно оцінено. Бачимо фільми Dune, Atlantis та Rhino. Порахуємо коефіцієнт кореляції для цього випадку застосовуючи формулу 4.5. Спочатку рахуємо середні значення для Михайла: $\bar{x} = \frac{5+5+4}{3} = \frac{14}{3} \approx 4.667$. Наступним етапом рахуємо

середні значення для Anastasya: $\bar{y} = \frac{4+4+5}{3} = \frac{13}{3} \approx 4.333$. Далі складемо таблицю складових, яка наведена в додатку Г (табл. Г.5).

Query Query History

```
1  SELECT * FROM recommendations."Viewers"
2  ORDER BY "Id" ASC
```

Data Output Messages Notifications

	Id [PK] uuid	Name text	Surname text
1	0197442d-64e3-73c8-b1b7-d6fe4b326a...	Mykhailo	Tkachenko
2	01974acb-3bd1-7fcf-9048-906e3d1f4da9	Olena	Shevchenko
3	01974acb-3d02-7d77-b30e-dbde70bc33...	Ivan	Bondar
4	01974acb-3d40-7074-86b2-82a42bbb17...	Mariya	Koval
5	01974acb-3d73-7092-bc93-0c1849c1c9...	Dmytro	Tkachenko
6	01974acb-3dae-7af6-82ee-5fb8142e587e	Anastasiya	Lysenko
7	01974acb-3dde-7ee8-bd38-92ad05730c...	Yuriy	Melnyk
8	01974acb-3e10-7c04-bc41-4df74131a5...	Kateryna	Horbach
9	01974acb-3e3d-7247-a707-fd6085d899...	Oleh	Polishchuk
10	01974acb-3e6c-77f3-aa83-8b4200474bf5	Nadiya	Tymoshenko
11	01974acb-3e9e-7be7-8e4f-2de5fa3aae36	Andriy	Verbytskyi
12	01974acb-b8ec-7fa8-ba69-faaf7eec0d66	Sofiia	Danylko
13	01974acb-b936-7ee6-8a91-59741295d5...	Taras	Zadorozhnyi
14	01974acb-b967-7639-8297-b1b719d1af...	Alina	Krut
15	01974acb-b99b-7b69-8456-96689c37fb...	Petro	Skrypnyk
16	01974acb-b9cd-792f-bcc9-bd60c83b972e	Yelyzaveta	Kramarenko
17	01974acb-ba0c-72ae-ac8b-2dbe6f147d3c	Mykola	Hnatyuk
18	01974acb-ba39-7e3b-844d-1a1b797cb6...	Viktoriya	Semenova

Total rows: 21 of 21 Query complete 00:00:00.088

Рисунок 4.7 Тестові користувачі

Тепер порахуємо суми: $\sum(x_i - \bar{x}) * (y_i - \bar{y}) = 0.222 - 0.111 + 0.222 = 0.333$; $\sum(x_i - \bar{x})^2 = 0.111 + 0.111 + 0.444 = 0.666$; $\sum(y_i - \bar{y})^2 = 0.444 + 0.111 + 0.111 = 0.666$. За формулою 4.5: $sim(u, v) = \frac{0.333}{\sqrt{0.666} * \sqrt{0.666}} = \frac{0.333}{0.666} = 0.5$. Приклади програмного обчислення для доведення правильності розрахунків наведено в додатку Г (рис. Г.6 – Г.7). Тепер спробуємо отримати рекомендації на основі дружніх стосунків, отримуємо рекомендовані фільми (рис. 4.9).

Query Query History

```
SELECT
    v."Name" || ' ' || v."Surname" AS "Viewer",
    mr."Stars" AS "ViewerRating",
    mov."Title" AS "Movie",
    f."Name" || ' ' || f."Surname" AS "Friend",
    mr2."Stars" AS "FriendRating"
FROM "recommendations"."MovieRating" mr
JOIN "recommendations"."Viewers" AS v ON mr."ViewerId" = v."Id"
JOIN "recommendations"."Movies" AS mov ON mr."MovieId" = mov."Id"
JOIN "recommendations"."MovieRating" AS mr2
    ON mr."MovieId" = mr2."MovieId" AND mr2."ViewerId" = '01974acb-3dae-7af6-82ee-5fb8142e587e'
JOIN "recommendations"."Viewers" AS f ON mr2."ViewerId" = f."Id"
WHERE mr."ViewerId" = '0197442d-64e3-73c8-b1b7-d6fe4b326a40';
```

Data Output Messages Notifications

	Viewer text	ViewerRating integer	Movie text	Friend text	FriendRating integer
1	Mykhailo Tkachenko	5	Dune: Part Two	Anastasiya Lysenko	5
2	Mykhailo Tkachenko	5	Atlantis	Anastasiya Lysenko	4
3	Mykhailo Tkachenko	4	Rhino	Anastasiya Lysenko	4

Рисунок 4.8 – Аналіз даних в базі даних

```
public class CollaborativeRecommendationSystem :
ICollaborativeRecommendationSystem
{
    private readonly IEnumerable<ICollaborativeRecommender> _recommenders;

    public
CollaborativeRecommendationSystem(IEnumerable<ICollaborativeRecommender>
recommenders)
    {
        _recommenders = recommenders;
    }

    public Dictionary<Guid, double> GetRelevant(Viewer viewer)
    {
        Dictionary<Guid, double> combinedScores = new();
        Dictionary<Guid, double> combinedWeights = new();
```

```
        foreach (ICollaborativeRecommender recommender in _recommenders)
        {
            Dictionary<Guid, double> scores =
recommender.GetCollaborativeScore(viewer);
            double weight = recommender.Weight;

            foreach (KeyValuePair<Guid, double> kv in scores)
            {
                if (combinedScores.TryAdd(kv.Key, 0))
                    combinedWeights[kv.Key] = 0;

                combinedScores[kv.Key] += kv.Value * weight;
                combinedWeights[kv.Key] += weight;
            }
        }

        Dictionary<Guid, double> weightedAverages = combinedScores
            .Where(kv => combinedWeights[kv.Key] > 0)
            .ToDictionary(kv => kv.Key, kv => kv.Value /
combinedWeights[kv.Key]);

        return Normalize(weightedAverages)
            .OrderByDescending(kv => kv.Value)
            .ToDictionary(kv => kv.Key, kv => kv.Value);
    }

    private static Dictionary<Guid, double> Normalize(Dictionary<Guid,
double> scores)
    {
        if (scores.Count == 0) return scores;

        double mean = scores.Values.Average();
        double std = Math.Sqrt(scores.Values.Average(v => Math.Pow(v - mean,
2)));
        if (std < 1e-6) return scores.ToDictionary(kv => kv.Key, _ => 1.0);

        return scores.ToDictionary(
            kv => kv.Key,
            kv =>
```

```
            {
                double z = (kv.Value - mean) / std;
                return 1 / (1 + Math.Exp(-z));
            }
        );
    }
}
```

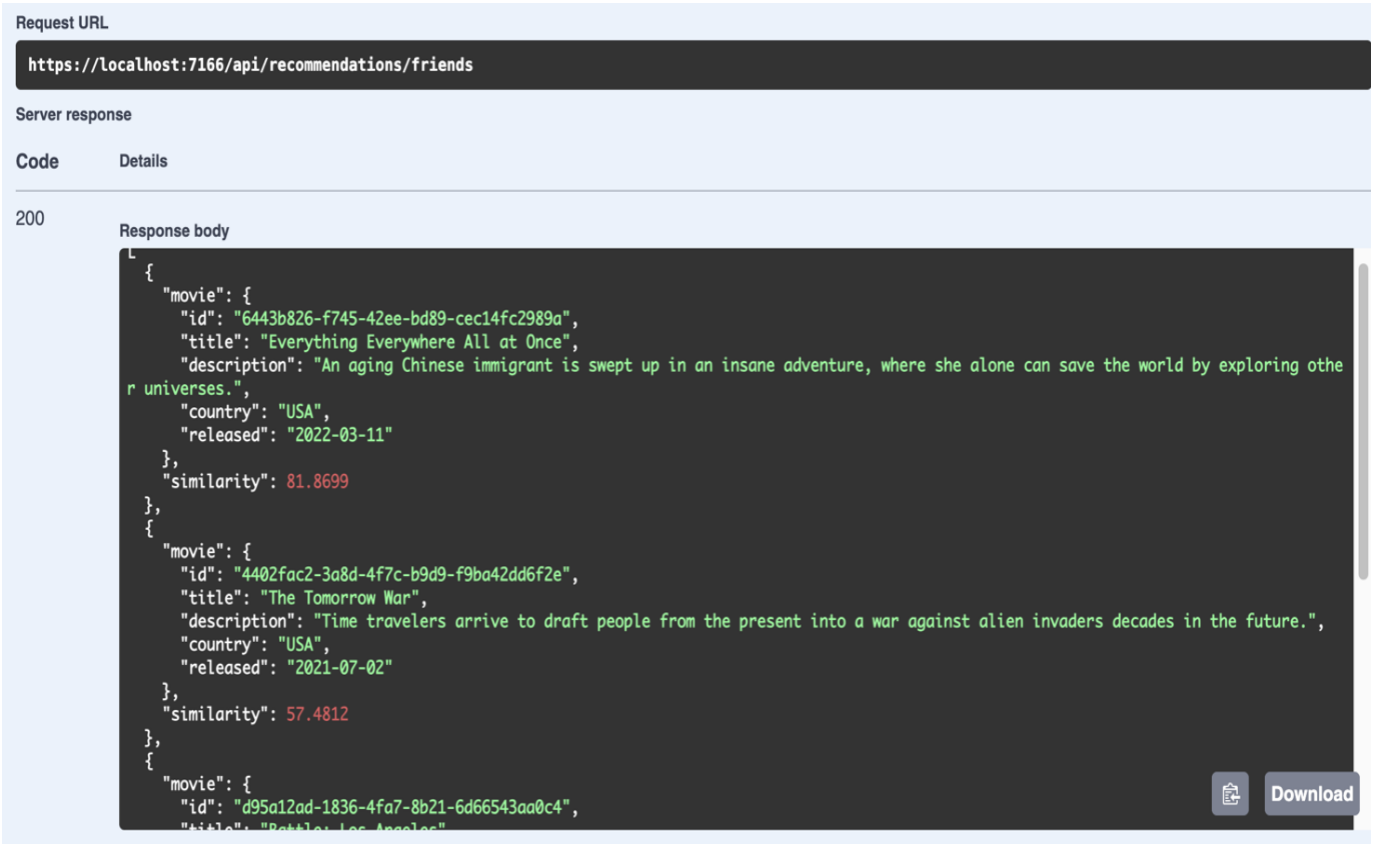

Рисунок 4.9 Результат роботи колаборативної рекомендації

4.6 Інтеграція та розгортання програмної системи

4.6.1 Розгортання програмної системи

У рамках побудови інфраструктури для програмної системи було використано хмарну платформу Microsoft Azure (рис. 4.9), що надала зручний інструментарій для управління ресурсами та розгортання застосунків. Всі компоненти системи були об’єднані в окрему групу ресурсів MovieCaptain, що спрощує адміністрування, моніторинг та масштабування.

Було налаштовано два App Service Plans — ASP-MovieCaptain-9816 та MovieCaptain-Backend, які надають обчислювальні потужності для хостингу вебзастосунків. Основний застосунок, реалізований у вигляді REST API на базі ASP.NET Core, розгорнуто як App Service під назвою moviecaptainapi.

Для контейнеризованих сервісів, зокрема RabbitMQ, було створено власний Azure Container Registry — MovieCaptainContainerRegistry, у якому зберігаються Docker-образи, що використовуються під час автоматизованого розгортання. Це дозволяє ефективно управляти залежностями та версіями образів.

Зберігання та обробку даних у системі забезпечує Supabase (рис. 4.10) — сучасна хмарна платформа, яка виступає як Backend-as-a-Service. В її основі лежить PostgreSQL, що гарантує надійність і масштабованість. Supabase також надає RESTful API, вбудовану авторизацію, функції безпеки та можливість інтеграції з зовнішніми сервісами, що значно прискорює розробку та спрощує супровід бази даних. Supabase обрано як альтернативу традиційним інстансам баз даних у Azure задля зниження вартості та покращення гнучкості у розгортанні. Розгорнута система доступна публічно по спеціальному службовому домену який надається Azure безкоштовно при створенні веб вебсервісу. Піднята система має відображати Swagger, що є свідком того що систему було успішно розгорнуто (рис. 4.11).

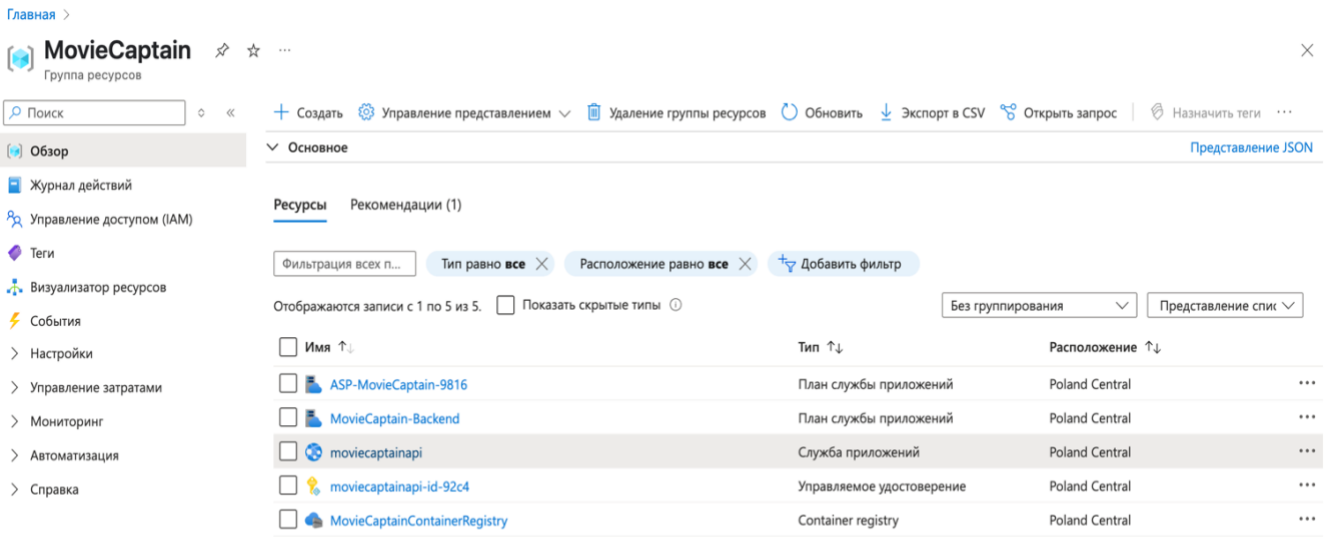

Рисунок 4.10 – Хмарна інфраструктура

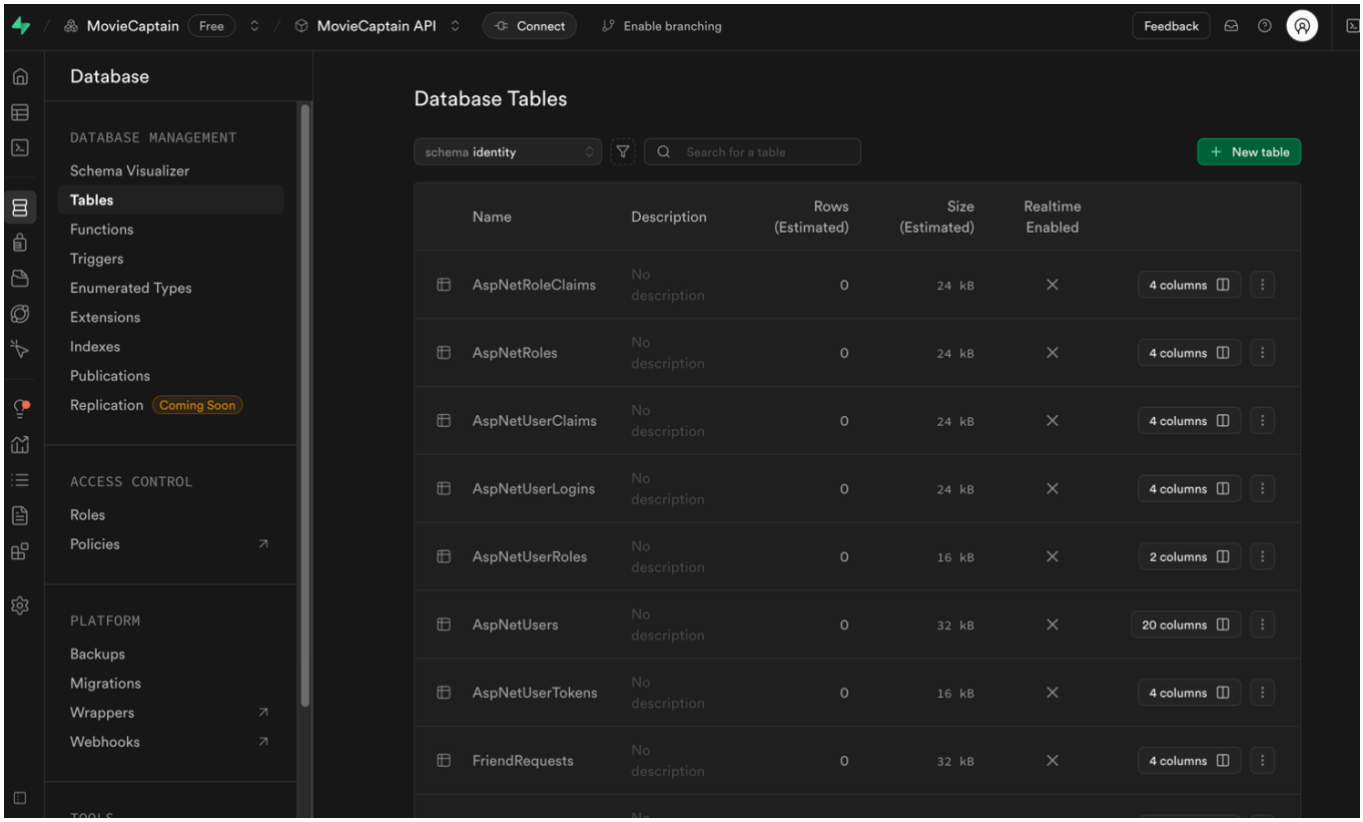

Рисунок 4.11 – Хмарне сховище Superbase

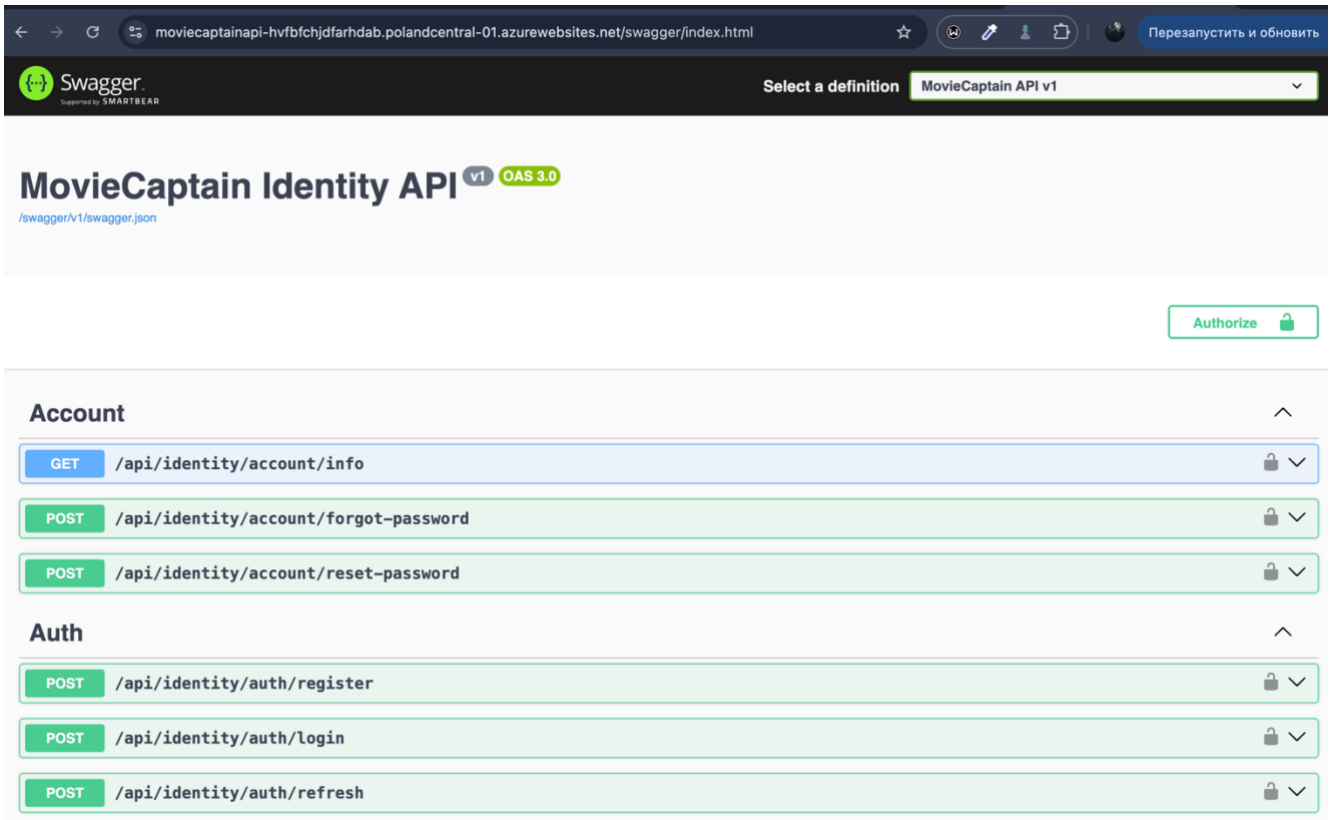

Рисунок 4.12 – Розгорнута система на публічному домені

4.6.2 Автоматизація публікації хмарної системи

З метою забезпечення надійного та контрольованого процесу розгортання, у проекті MovieCaptain було реалізовано автоматичну публікацію серверної частини застосунку з використанням CI/CD-пайплайнів (рис. 4.12), налаштованих за допомогою GitHub Actions.

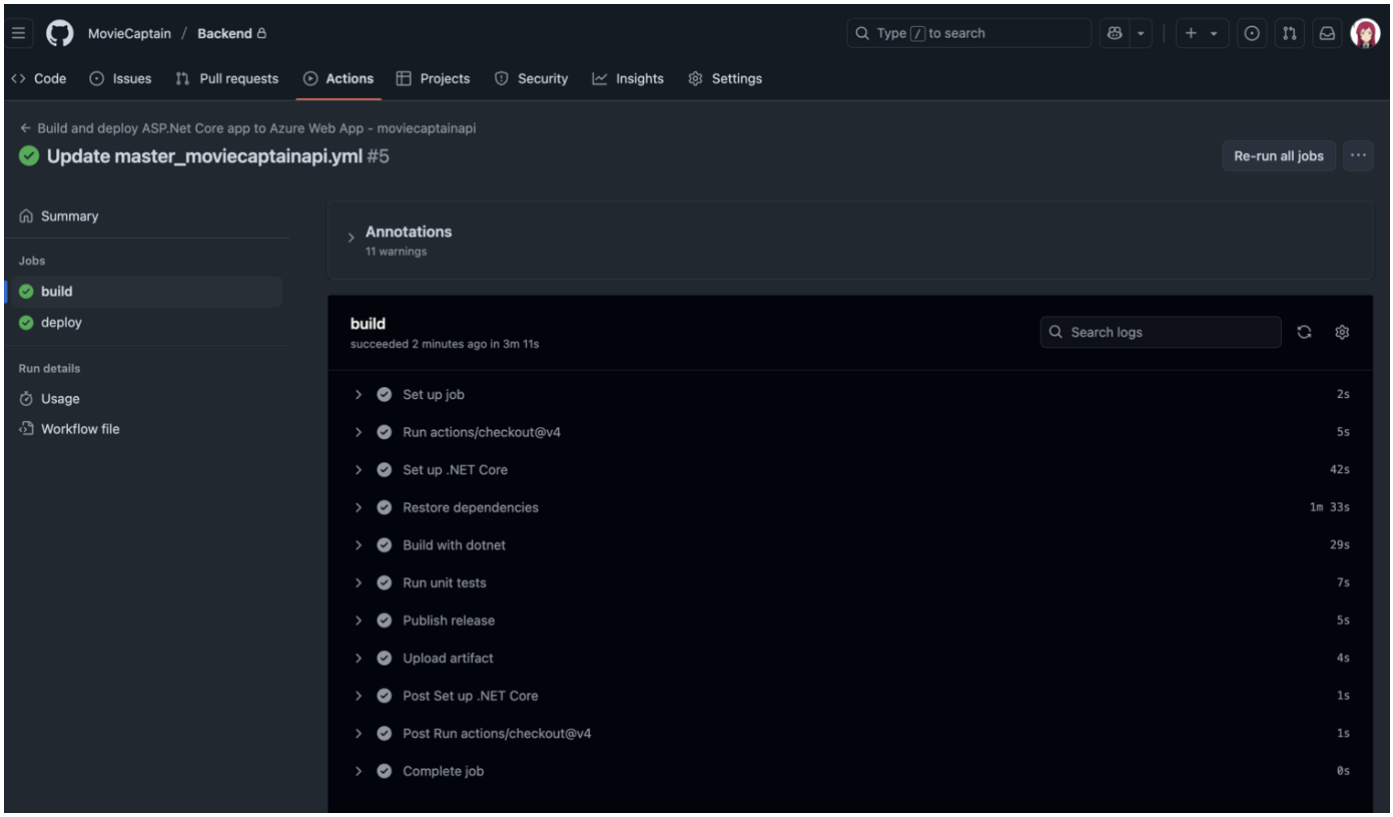

Рисунок 4.13 – Результат роботи пайплайну

Після кожного пушу до головної гілки репозиторію активується автоматичне виконання сценарію публікації.

У рамках пайплайна здійснюється кілька ключових етапів:

- перевірка коду — встановлюється середовище виконання .NET і виконується компіляція проєкту з конфігурацією Release;
- юніт-тестування – автоматично виявляються проєкти з іменем *.Tests.csproj, після чого для кожного з них виконується команда dotnet test без повторної побудови, з виводом результатів;

- публікація артефактів – після успішного тестування API застосунок публікується з використанням dotnet publish, і результат зберігається у вигляді артефактів;
- розгортання – готові артефакти автоматично передаються до Azure App Service (moviecaptainapi) за допомогою GitHub Action azure/webapps-deploy@v2. Ідентифікація відбувається через спеціальний секрет AZURE_WEBAPP_PUBLISH_PROFILE, який містить облікові дані публікації.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Вибір підходу тестування

У межах розробки системи було обрано модульне тестування як основний підхід до перевірки коректності реалізації бізнес-логіки. Цей підхід дозволяє ізолювати окремі компоненти системи та перевіряти їхню поведінку незалежно від зовнішнього середовища. Оскільки архітектура побудована на принципах CQRS, модульне тестування стало найбільш доцільним варіантом: кожна команда або запит має чітко визначену поведінку, що зручно перевіряти окремо. Такий підхід сприяє швидкому виявленню помилок, спрощує супровід коду та забезпечує стабільність змін у системі без порушення суміжної функціональності.

5.2 Тестування програмної системи

Для реалізації тестування створено окремі проєкти для кожного модуля системи (рис. 5.1), що дозволяє чітко відокремити тестовий код від основного. У кожному такому проєкті використовується підхід «один тестовий клас на одну CQRS-команду або запит», що забезпечує структурованість, зручність навігації та легкість у підтримці.

Тести реалізовані за допомогою фреймворку xUnit, а для заміни зовнішніх залежностей (контексти бази даних, логери, брокери повідомлень, користувацький контекст тощо) використовується бібліотека Moq. Така ізоляція дозволяє досягти повної контрольованості середовища тестування та отримати детерміновані, повторювані результати.

Усі тести дотримуються шаблону Arrange–Act–Assert, що підвищує їхню читабельність та дозволяє чітко простежити логіку перевірки. Увага приділена як позитивним, так і негативним сценаріям: тестується правильна обробка винятків, помилки авторизації, перевірка наявності сутностей у базі, граничні умови тощо. Це гарантує високу якість покриття та надійність усіх основних логік системи.

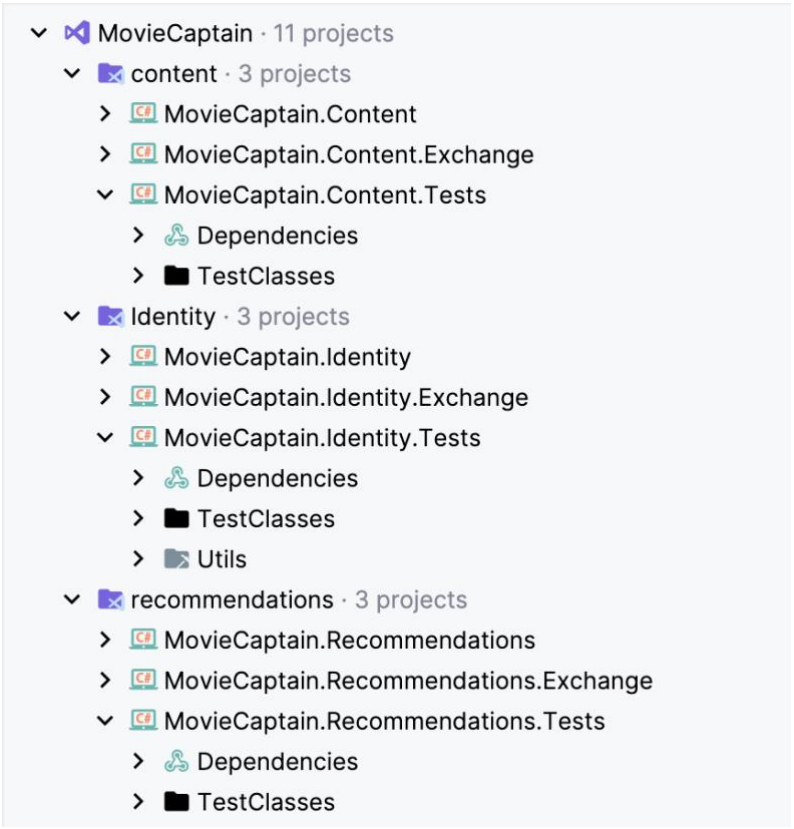

Рисунок 5.1 – Структура проектів тестування кожного модулю

Для створення фіктивних об'єктів використовується бібліотека Moq, розглянемо її використання на прикладі реєстрації в конструкторі одного з тестових класів:

```
public AcceptFriendRequestCommandHandlerTests()
{
    // Mock DbContext
    _dbContextMock = new Mock<ParticipantsContext>();

    // Mock the ExecutionContextStore
    Mock<ExecutionContextStore> executionContextStoreMock = new();
    _testIdentityId = Guid.NewGuid();

    // Setup the ExecutionContext store to return a valid ExecutionContext
    executionContextStoreMock
        .Setup(store => store.Get())
        .Returns(new ExecutionContext(
            _testIdentityId,
            new ClaimsPrincipal(
```

```
                new ClaimsIdentity([
                    new Claim(ClaimTypes.NameIdentifier,
_testIdentityId.ToString()),
                    new Claim(ClaimTypes.Email, "testuser@test.com"),
                    new Claim(ClaimTypes.Name, "testuser@test.com"),
                    new Claim(ClaimTypes.SerialNumber,
Guid.NewGuid().ToString())
                ]))
        );

    // Create an actual ExecutionContextAccessor using the mocked store
    ExecutionContextAccessor executionContextAccessor =
new(executionContextStoreMock.Object);

    // Create command handler with injected mocks
    _handler = new AcceptFriendRequestCommandHandler(
        _dbContextMock.Object,
        executionContextAccessor);
}
```

Тепер розглянемо використання моків для проведення валідації і ізольованого тестування:

```
[Fact]
public async Task Handle_ShouldAddFriend_WhenRequestIsValid()
{
    // Arrange
    Guid requesterId = Guid.NewGuid();

    Participant identity = new() { Id = _testIdentityId, Friends = new
List<Participant>() }; // Use same ID
    Participant requesterParticipant = new() { Id = requesterId };
    FriendRequest friendRequest = new() { Id = Guid.NewGuid(), ToId =
_testIdentityId, From = requesterParticipant };

    Mock<DbSet<Participant>> participantsDbSetMock =
MockHelper.CreateDbSetMock(new List<Participant> { identity });
    Mock<DbSet<FriendRequest>> friendRequestsDbSetMock =
```

```
MockHelper.CreateDbSetMock(new List<FriendRequest> { friendRequest });

    _dbContextMock.Setup(db =>
db.Participants).Returns(participantsDbSetMock.Object);
    _dbContextMock.Setup(db =>
db.FriendRequests).Returns(friendRequestsDbSetMock.Object);
    _dbContextMock.Setup(db =>
db.SaveChangesAsync(It.IsAny<CancellationToken>())).ReturnsAsync(1);

    AcceptFriendRequestCommand command = new
AcceptFriendRequestCommand(friendRequest.Id);

    // Act
    await _handler.Handle(command, CancellationToken.None);

    // Assert
    Assert.Contains(requesterParticipant, identity.Friends);
    _dbContextMock.Verify(db => db.Update(identity), Times.Once);
    _dbContextMock.Verify(db => db.FriendRequests.Remove(friendRequest),
Times.Once);
    _dbContextMock.Verify(db =>
db.SaveChangesAsync(It.IsAny<CancellationToken>()), Times.Once);
}
```

У цьому тесті перевіряється базовий позитивний сценарій — коректне додавання користувача до списку друзів після обробки дійсного запиту. Створюються два учасники: той, хто надсилає запит (requesterParticipant), і той, хто приймає його (identity). Останній ініціалізується з порожнім списком друзів, що дозволяє чітко перевірити зміну стану після виконання логіки.

Моки DbSet<Participant> і DbSet<FriendRequest> створюються за допомогою утиліти MockHelper, що дозволяє імітувати поведінку таблиць Entity Framework у пам’яті. Об'єкти моків повертаються при зверненні до db.Participants та db.FriendRequests, що імітує базовий доступ до бази даних без її реального використання.

Особливу роль у тесті відіграє метод Verify з бібліотеки Moq. Він дозволяє перевірити, що під час виконання тестованого методу були викликані конкретні методи залежностей з очікуваними параметрами.

Зокрема, перевіряється:

- що об’єкт identity був оновлений у контексті;
- що запит дружби був видалений;
- що всі зміни були збережені.

Метод It.IsAny<T>() використовується для вказівки, що перевірка має пройти незалежно від конкретного значення параметра.

ВИСНОВКИ

У результаті виконання кваліфікаціної роботи була розроблена програмна система створення соціальної мережі для синефілів, спроектована повноцінна архітектура веб-застосунку системи Movie Captain – україномовної соціальної платформи для синефілів.

У ході роботи було проаналізовано предметну галузь, виявлено ключові проблеми сучасних сервісів, а також сформульовано вимоги до майбутньої системи з урахуванням потреб користувачів.

В основі розробки закладено архітектурний стиль Modular Monolith, що дозволив логічно структурувати функціональність на незалежні доменні модулі та забезпечити можливість масштабування без втрати цілісності. Особливу увагу приділено реалізації CQRS-патерну, подійної взаємодії між модулями та підтримці принципу eventual consistency, що є ключовими для побудови продуктивної, асинхронної та розширюваної системи.

Ретельно спроєктовано структуру бази даних на основі PostgreSQL: передбачено зберігання профілів користувачів, рецензій, коментарів, реакцій, соціальних зв’язків, обраного контенту та системи рекомендацій. Кожен модуль має власну ізольовану схему, що спрощує розгортання, масштабування та супровід окремих частин системи.

У межах документації проєкту створено UML-діаграми, що чітко відображають логіку взаємодії користувачів із системою, поведінку підсистем, а також архітектуру розгортання та компонування внутрішніх модулів. Особливу увагу було приділено візуалізації процесу формування рекомендацій – одного з основних функціональних блоків системи.

Таким чином, запропонована модель проєкту Movie Captain є надійною основою для подальшої реалізації, тестування та вдосконалення. Вона відповідає сучасним вимогам до вебзастосунків, підтримує персоналізацію а також забезпечує зручний і масштабований фундамент для розвитку україномовної кіноплатформи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. IMDb. База даних фільмів. URL: https://www.imdb.com/ (Дата звернення: 10.05.2025)
2. Letterboxd. Соціальна мережа. URL: https://letterboxd.com/ (Дата звернення: 10.05.2025)
3. Microservices Killer: Modular Monolithic Architecture. URL: https://medium.com/design-microservices-architecture-with-patterns/microservices-killer-modular-monolithic-architecture-ac83814f6862 (Дата звернення: 25.04.2025)
4. Advanced Design Patterns in C#: Command Query Responsibility Segregation (CQRS). URL: https://krishan-samarawickrama.medium.com/advanced-design-patterns-in-c-command-query-responsibility-segregation-cqrs-d9ff598ccdf6 (Дата звернення: 20.04.2025)
5. **Building Event-Driven Microservices: Leveraging Organizational Data at Scale** / Adam Bellemare. O'Reilly Media, 2020. – 275 с.
6. Eventual Consistency vs. Strong Eventual Consistency vs. Strong Consistency. URL: https://www.baeldung.com/cs/eventual-consistency-vs-strong-eventual-consistency-vs-strong-consistency. (Дата звернення: 20.04.2025)
7. Learn Entity Framework Core. URL: https://www.learnentityframeworkcore.com/ (Дата звернення: 10.04.2025)
8. PostgreSql. Documentation. URL: https://www.postgresql.org/docs/17/index.html (Дата звернення: 12.05.2025)
9. The 10 Most Common Website Security Attacks. URL: https://www.tripwire.com/state-of-security/most-common-website-security-attacks-and-how-to-protect-yourself (Дата звернення: 08.05.2025)
10. Getting started with Azure App Service. URL: https://learn.microsoft.com/uk-ua/azure/app-service/getting-started?pivots=stack-net (Дата звернення: 10.05.2025)
11. Patterns of Enterprise Application Architecture / Martin Fowler та ін. Addison-Wesley, 2003. – 560 с.

12. Архів роботи на GitHub. URL: https://github.com/NureTkachenkoMykhailo/2025_B_PI_PZPI-21-2_Tkachenko_M_Yu