МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

«До захисту допущено»

В.о. завідувача кафедри

Оксана ШОВКОПЛЯС

(підпис)

04 грудня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 «Комп'ютерні науки»

освітньо-професійної програми «Інформатика»

на тему: «Інформаційна технологія проєктування веборієнтованої пошуково-рекомендаційної системи рецензування фільмів»

здобувача групи ІН.м-32 Гончаренка Дмитра Миколайовича

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Дмитро ГОНЧАРЕНКО

(підпис)

Керівник,

асистент кафедри комп'ютерних наук,

кандидат фізико-математичних наук Ольга ШУТИЛЄВА

(підпис)

Суми – 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Оксана ШОВКОПЛЯС
(підпис)

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 «Комп'ютерні науки», освітньо-професійної програми «Інформатика»
здобувача групи ІН.м-32 Гончаренка Дмитра Миколайовича

1. Тема роботи: «Інформаційна технологія проєктування веборієнтованої пошуково-рекомендаційної системи рецензування фільмів» затверджена наказом по СумДУ від *«03» грудня 2024 року № 1257-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 06 грудня 2024 року*
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналіз проблеми предметної області, постановка й формування завдань дослідження. 2) Огляд сучасних технологій проєктування веборієнтованих інформаційних систем. 3) Розробка пошуково-рекомендаційної системи. 4) Аналіз результатів.
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____
6. Консультанти до проєкту (роботи), із значенням розділів проєкту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «18» серпня 2024 р.

Завдання прийняв до виконання _____ (підпис) Керівник _____ (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналіз проблеми предметної області, постановка й формування завдань дослідження*		
2	*Огляд сучасних технологій проєктування веборієнтованих інформаційних систем*		
3	*Розробка пошуково-рекомендаційної системи рецензування фільмів*		
4	*Аналіз отриманих результатів*		
5	*Оформлення пояснювальної записки до кваліфікаційної роботи*		

Здобувач вищої освіти _____ (підпис) Керівник _____ (підпис)

АНОТАЦІЯ

Записка: 65 стор., 23 рис., 14 табл., 1 додаток, 24 використаних джерел.

Обґрунтування актуальності теми роботи – Тема кваліфікаційної роботи є актуальною, оскільки в сучасному інформаційному суспільстві постійно зростає обсяг цифрового контенту, і для користувачів стає дедалі важливішим мати зручний інструмент, що дозволяє швидко знаходити та отримувати рекомендації, які відповідають їхнім інтересам.

Об’єкт дослідження – процес створення рекомендацій для користувачів на основі даних про них.

Мета роботи – веборієнтованої пошуково-рекомендаційної інформаційної технології рецензування фільмів.

Методи дослідження – системи управління базами даних, технології для створення веборієнтованих інформаційних систем, рекомендаційні алгоритми.

Результати – на основі проведеного дослідження інформаційних джерел та аналізу різних технологій проєктування було розроблено веборієнтовану пошуково-рекомендаційну систему для рецензування фільмів. Система забезпечує користувачів можливістю знаходити та оцінювати кінострічки, враховуючи їхні індивідуальні вподобання.

RECOMMENDATION SYSTEM, CLEAN ARCHITECTURE, MS SQL SERVER, C#, ENTITY FRAMEWORK, ASP.NET, WEB API, JWT, COOKIE, TYPESCRIPT, REACT, MUI

ЗМІСТ

ВСТУП

Обґрунтування вибору теми роботи. В умовах перенасиченості інформаційного середовища важливим завданням стає надання користувачам персоналізованих рекомендацій, зокрема й під час вибору фільмів. Це завдання вирішується шляхом розробки веборієнтованих пошуково-рекомендаційних систем, які забезпечують індивідуальний підхід до користувача, враховуючи його інтереси.

Актуальність. Пошуково-рекомендаційні системи стали невід'ємною частиною цифрового світу, забезпечуючи персоналізовану взаємодію користувача з контентом. У кінематографічній індустрії, де кількість нових релізів зростає, важливість ефективного та індивідуалізованого підходу до вибору фільмів тільки посилюється. Користувачі потребують зручних і точних рекомендацій, які допомагають швидко знаходити цікаві фільми серед тисяч варіантів. Розвиток вебтехнологій та пошукових алгоритмів дозволяє створювати рішення, які можуть автоматично аналізувати вподобання користувачів та підбирати найкращі варіанти для кожного. Ця робота є актуальною, оскільки вона спрямована на створення технології, що задовольняє потреби користувачів у персоналізованому пошуку, підвищуючи зручність та швидкість доступу до інформації.

Об’єкт дослідження. Процес створення рекомендацій для користувачів на основі даних про них.

Предмет дослідження. Системи управління базами даних, технології для створення веборієнтованих інформаційних систем, рекомендаційні алгоритми.

Новизна. Відмінною рисою створеної системи є інтеграція механізмів аналізу оцінок фільмів користувачами, рецензій та реакцій на рецензії, що дозволяє враховувати як особисті вподобання користувача, так і загальні. Такий підхід забезпечує більш точні й релевантні рекомендації, оскільки

система здатна адаптуватися до змін у вподобаннях користувача та оперативно оновлювати рекомендації на основі нових даних.

Структура. Дана робота складається зі вступу, інформаційного огляду, постановки задачі, вибору методів для розв’язання поставленої задачі, програмної реалізації, висновків, списку використаних джерел та додатків.

1. ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1. Підходи до створення рекомендаційних систем

Рекомендаційна система – це система, яка за допомогою спеціального алгоритму або набору алгоритмів автоматично аналізує дані про користувачів, їхні вподобання та поведінку, щоб запропонувати найбільш релевантний контент, продукти чи послуги. Мета таких систем – полегшити вибір користувачам і підвищити їхню задоволеність від взаємодії з платформою [1].

Рекомендаційні системи застосовуються в багатьох сферах, таких як онлайн-магазини (Amazon, Rozetka), платформи для перегляду фільмів (Netflix), музичні сервіси (Spotify), соціальні мережі (Facebook, TikTok) та навіть в освітніх і фінансових додатках.

Принципи роботи рекомендаційних систем [2]:

1. Система збирає інформацію про поведінку користувача – його перегляди, покупки, оцінки, час, проведений на різних сторінках тощо.
2. Зібрані дані аналізуються за допомогою відповідних алгоритмів для визначення певних закономірностей.
3. На основі проведеного аналізу система створює список рекомендацій, який вона постійно коригує, враховуючи нові дані про користувача.

Рекомендаційні системи можна поділити на три основні типи:

1. Контентно-орієнтовані (Content-based) – аналізують характеристики об'єктів (фільмів, книг, музики) та користувацькі вподобання, щоб пропонувати подібні варіанти на основі історії вибору. Наприклад, якщо користувач часто дивиться комедії, йому пропонуватимуть більше фільмів цього жанру.
2. Системи на основі колаборативної фільтрації (Collaborative filtering) – роблять рекомендації на основі взаємодії користувача з іншими

користувачами. Наприклад, якщо інший користувач зі схожими вподобаннями позитивно оцінив певний фільм, система може порадити його і вам.

3. Гібридні рекомендаційні системи (Hybrid) – поєднують кілька підходів, щоб підвищити точність рекомендацій. Наприклад, такі платформи, як Netflix або Amazon використовують одночасно і аналіз контенту, і колаборативну фільтрацію, щоб надати більш точні рекомендації.

1.2. Підходи до побудови вебзастосунків

Вебзастосунки – це програми, які працюють у браузері і доступні через Інтернет. Вони дозволяють користувачам взаємодіяти з різними функціями та послугами без необхідності встановлення програмного забезпечення на своїх пристроях [3].

Загалом вебзастосунки складається з таких ключових елементів:

1. Клієнтська частина (frontend) – це частина вебзастосунку, яку бачить і з якою взаємодіє користувач. Він включає в себе всі елементи інтерфейсу, такі як кнопки, форми, меню та візуальні компоненти, що реалізуються за допомогою різних мов програмування та фреймворків. Фронтенд відповідає за забезпечення зручності користування та візуальної привабливості застосунку.

2. Серверна частина (backend) – це частина вебзастосунку, яка обробляє запити від фронтенду і виконує всю бізнес логіку. Бекенд забезпечує аутентифікацію, авторизацію, обробку даних, що надходять з фронтенду та взаємодію з базою даних [4].

3. База даних – це організована структура для зберігання, управління та отримання даних, необхідних для функціонування вебзастосунку. Вона може бути реляційною, де дані зберігаються у вигляді таблиць з чіткими зв'язками між ними, або нереляційною, де дані зберігаються у форматі документів. Бази даних дозволяють зберігати інформацію та забезпечують ефективний доступ до неї через різні запити.

Також для повноцінного функціонування вебзастосунку потрібно реалізувати такі елементи [5]:

1. Хостинг та розгортання – це процеси, які забезпечують доступність вебзастосунку в Інтернеті. Хостинг передбачає використання сервера або хмарних платформ, для розміщення вебзастосунку. Розгортання включає в себе встановлення необхідного програмного забезпечення, налаштування серверного середовища та бази даних, завантаження файлів вебзастосунку на хостинг. Ці етапи гарантують, що застосунок буде доступний для користувачів і працюватиме стабільно.

2. Інтеграція сторонніх сервісів – це процес, за допомогою якого вебзастосунок взаємодіє з зовнішніми API та сервісами для розширення своїх функцій. Це може включати платежі, аутентифікацію, аналітику та інші функції, які не реалізуються всередині застосунку. Інтеграція дозволяє зекономити час і ресурси, адже розробники можуть використовувати вже готові рішення, щоб покращити користувацький досвід та забезпечити додаткові можливості без необхідності створювати все з нуля.

1.2.1. Підходи до побудови серверної частини

1. Безсерверна архітектура.

Безсерверна архітектура дозволяє розробникам створювати бекенд-додатки без необхідності управляти серверною інфраструктурою, оскільки обчислення виконуються на стороні хмарних провайдерів. Це забезпечує автоматичне масштабування, зменшує витрати на обслуговування та дає змогу зосередитися на написанні коду замість управління серверами. Однак ця архітектура має свої недоліки, зокрема залежність від постачальників хмарних послуг і обмеження на час виконання функцій, що може ускладнити реалізацію деяких сценаріїв.

2. Монолітна архітектура.

Монолітна архітектура передбачає, що всі компоненти вебдодатку, такі як інтерфейс, бізнес-логіка та доступ до даних, об'єднані в одному проєкті. Це

спрощує розробку і деплоймент, оскільки все управляється як єдине ціле. Однак, з ростом проєкту можуть виникнути труднощі з масштабуванням та підтримкою, оскільки зміна в одній частині системи може вплинути на інші частини, ускладнюючи обслуговування та тестування [6].

3. Мікросервісна архітектура.

У мікросервісній архітектурі вебдодаток складається з незалежних сервісів, кожен з яких виконує певну функцію і може розгортатися окремо. Цей підхід дозволяє розробникам обирати найкращі технології для кожного сервісу, спрощує масштабування та тестування, адже зміни в одному сервісі не впливають на інші. Проте управління великою кількістю сервісів може бути складним, вимагати значних зусиль для інтеграції та моніторингу [7;8].

1.2.2. Підходи до побудови клієнтської частини

1. Односторінкові додатки (SPA)

Односторінкові додатки завантажують весь необхідний HTML, CSS і JavaScript один раз, а потім динамічно оновлюють вміст без перезавантаження сторінки. Це забезпечує швидку та плавну взаємодію користувачів, оскільки зміни в інтерфейсі реалізуються за допомогою JavaScript. Популярні фреймворки для створення SPA включають React, Angular та Vue.js. Однак такий підхід може ускладнити SEO-оптимізацію, оскільки контент завантажується асинхронно.

2. Багатосторінкові додатки (MPA)

Багатосторінкові додатки складаються з декількох HTML-сторінок, які завантажуються з сервера при кожному переході. Цей підхід є традиційним для вебдодатків і добре підходить для SEO, оскільки кожна сторінка може бути проіндексована пошуковими системами. Створення MPA може бути простішим для малих або середніх проєктів, але воно може бути менш ефективним у плані швидкості завантаження та користувацького досвіду порівняно з SPA.

3. Прогресивні вебдодатки (PWA)

Прогресивні вебдодатки поєднують у собі переваги вебдодатків та мобільних додатків. Вони забезпечують можливість роботи в офлайн-режимі, швидке завантаження та можливість установки на пристрої користувачів. PWA використовують технології, такі як Service Workers і Web App Manifest, для забезпечення високої продуктивності та зручності. Це дозволяє створювати адаптивні інтерфейси, які можуть працювати на різних платформах і пристроях.

1.3. Аналіз існуючих рішень

Аналіз існуючих рішень допомагає зрозуміти, які функції та інструменти вже використовуються в популярних платформах для оцінки та рецензування фільмів, а також виявити їхні сильні та слабкі сторони. Це дозволяє визначити, що саме очікує цільова аудиторія, яких аспектів бракує поточним платформам, і як можна вдосконалити користувацький досвід або додати унікальні функції. Такий огляд забезпечує ґрунтовну базу для формування концепції майбутнього проєкту, допомагаючи уникнути дублювання функціоналу та ефективно реалізувати нові ідеї.

Розглянемо такі популярні сервіси:

- IMDb
- Rotten Tomatoes
- Metacritic
- Letterboxd
- TMDb (The Movie Database)

Детальне порівняння платформ за контентом і функціями наведене в таблиці 1.1, а за взаємодією з користувачем та принципом створення рекомендацій в таблиці 1.2.

Таблиця 1.1 – Порівняння за контентом і функціями.

Платформа	**Тип контенту**	**Основні функції для користувачів**	**Система оцінювання**
IMDb	Огляди фільмів, телесеріалів, акторів і знімальних груп	Оцінки, огляди, можливість створення списків, участь у дискусіях	10-бальна система рейтингу від користувачів
Rotten Tomatoes	Рецензії від критиків і глядачів, агреговані рейтинги	"Томатометр", обговорення, рейтинг користувачів і критиків	Відсоток позитивних рецензій, рейтинг "свіжий" або "гнилий"
Metacritic	Рецензії від критиків з різних джерел, рейтинги	Агреговані огляди, можливість перегляду оцінок користувачів та критиків	100-бальна система балів, "метаскор" для критиків та оцінки користувачів
Letterboxd	Персональні списки фільмів, огляди від користувачів	Оцінки, рецензії, створення списків, слідкування за іншими користувачами	5-зіркова система рейтингу
TMDb	Інформація про фільми, серіали, акторів, коментарі користувачів	Оцінки, коментарі, створення списків, інтеграція з іншими додатками	10-бальна система рейтингу

Таблиця 1.2 – Порівняння за взаємодією з користувачем та рекомендаціями.

Платформа	**Вид рекомендацій**	**Особливості спільноти**
IMDb	Персоналізовані рекомендації	Велика аудиторія, активні обговорення
Rotten Tomatoes	Рекомендації на основі оцінок критиків і користувачів	Вплив на кінокритику, сильна спільнота
Metacritic	Рекомендації за агрегованими оцінками	Активна спільнота, аналітичний підхід
Letterboxd	Соціальна рекомендація	Соціальна взаємодія, акцент на відгуках
TMDb	Персоналізовані рекомендації	Активна спільнота ентузіастів, відкриті дані

Аналізуючи існуючі платформи оцінювання та рецензування фільмів, можна побачити як сильні, так і слабкі сторони кожної з них. IMDb пропонує детальну базу даних і персоналізовані рекомендації, але не забезпечує достатньої інтерактивності для користувачів, які прагнуть більшого соціального залучення. Rotten Tomatoes і Metacritic надають агреговані критичні рецензії, що робить їх цінними для об’єктивного аналізу, однак вони обмежені в соціальній взаємодії між користувачами. Letterboxd і TMDb дозволяють активну взаємодію з контентом та іншими користувачами, але не мають системи агрегованих оцінок.

З цього слідує, що є потреба створення платформи, яка б поєднувала кращі аспекти кожного рішення: надійну інформаційну базу, агреговані оцінки, гнучку та персоналізовану систему рекомендацій. Такий підхід допоможе задовольнити потреби різних типів користувачів і створити конкурентоспроможний продукт.

1.4. Постановка задачі

Метою роботи є розробка веборієнтованої пошуково-рекомендаційної інформаційної технології рецензування фільмів.

Функціональні вимоги до системи:

- Користувач може переглядати список фільмів.
- Користувач може застосовувати фільтри до списку фільмів (назва, жанр).
- Користувач може авторизуватися в системі.
- Авторизований користувач може оцінювати фільми.
- Авторизований користувач може отримати персоналізовані рекомендації, які базуються на основі даних про нього.
- Авторизований користувач може поставити реакцію на рецензію критика.
- Критик може створювати, редагувати та видаляти рецензії на фільми.

- Адміністратор може вносити, редагувати та видаляти дані про фільми.
- Адміністратор може створювати нові жанри фільмів.
- Адміністратор може керувати ролями користувачів.

Для досягнення даної мети потрібно виконати такі кроки:

1. Створити надійну та адаптивну систему для зберігання даних.

1.1. Розробити структуру бази даних враховуючи всі необхідні сутності системи.

2. Розробити серверну частину вебдодатку.

2.1. Реалізувати систему обробки запитів користувачів в залежності від їх ролі (отримання інформації про фільми, редагування, оцінювання тощо).

2.2. Реалізувати рекомендаційний алгоритм.

2.3. Реалізувати систему авторизації/автентифікації.

3. Розробити клієнтську частину вебдодатку.

3.1. Реалізувати зручний та надійний інтерфейс для взаємодії з користувачем.

3.2. Реалізувати обмежену доступність до функціоналу системи в залежності від ролі користувача.

4. Налаштувати взаємодію всіх частин системи.

5. Провести тестування системи.

2. ВИБІР МЕТОДІВ РОЗВ’ЯЗАННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

Вибір методів розв’язання поставленої задачі є надзвичайно важливим, оскільки саме від нього значною мірою залежить якість кінцевого результату. При створенні програмних продуктів правильний вибір відповідних підходів та інструментів дозволяє не лише оптимізувати використання ресурсів, але й знизити ризик виникнення помилок, скоротити час розробки та забезпечити зручність подальшого обслуговування додатка. Обґрунтований підхід до вибору методів також сприяє легкості масштабування рішення та його адаптації до майбутніх потреб.

2.1. Вибір підходу до створення рекомендаційних систем

Гібридні (Hybrid) рекомендаційні системи поєднують методи контентно-орієнтованого фільтрування (Content-based) і колаборативного фільтрування (Collaborative filtering). Контентний підхід використовує характеристики об’єктів (наприклад, жанр фільму, актори, режисери), тоді як колаборативне фільтрування базується на схожості між користувачами та їх уподобаннях [1].

Гібридний підхід може компенсувати недоліки кожного з методів поодинці. Наприклад, якщо новий користувач має мало взаємодій (так званий cold-start), контентне фільтрування може бути корисним, оскільки не потребує великої кількості історії. У свою чергу, колаборативне фільтрування дозволяє отримати рекомендації, базуючись на поведінці подібних користувачів.

Гібридна система має потенціал усунути обмеження традиційних методів, таких як проблема надмірної спеціалізації (over-specialization) або упередження популярності (popularity bias), забезпечуючи більш персоналізовані та якісніші рекомендації.

2.2. Вибір методів для побудови серверної частини

Вибір методів для побудови серверної частини є ключовим етапом у розробці будь-якої програмної вебсистеми. Одним із перших рішень, яке потрібно прийняти, є вибір архітектури бекенду, яка визначатиме загальну структуру та взаємодію між компонентами серверної частини, що в свою чергу впливатиме на масштабованість, надійність і ефективність системи. Після визначення архітектури, необхідно обрати відповідний бекенд фреймворк, який надасть необхідні інструменти для реалізації серверної логіки, взаємодії з базою даних та забезпечення безпеки. Правильний вибір архітектури та фреймворку дозволяє досягти оптимальної продуктивності, зручності підтримки та подальшого розвитку програмного забезпечення.

2.2.1. Вибір архітектури бекенду

Clean Architecture (чиста архітектура) є одним із найпоширеніших підходів до проєктування бекенд-систем, що дозволяє досягти високої гнучкості, ізоляції бізнес-логіки від інших компонентів системи та забезпечує легкість у тестуванні та підтримці [9].

Це архітектурний стиль, що спрямований на створення програмних систем з чітко визначеними рівнями абстракції та слабким зв’язком між компонентами. В основі цього підходу лежить принцип розділення системи на шари, кожен з яких виконує свою специфічну задачу. Архітектура повинна бути гнучкою, щоб змінювати окремі частини без впливу на інші.

Структура Clean Architecture розглядає систему як набір концентричних кіл, де кожне коло відповідає за окрему функціональність. Важливим принципом є ізоляція бізнес-логіки від інтерфейсів користувача та зовнішніх залежностей [10].

Основні принципи Clean Architecture:

Незалежність від UI – бізнес-логіка не повинна залежати від конкретної реалізації інтерфейсу користувача. Це дозволяє змінювати інтерфейс без впливу на ядро програми.

Незалежність від баз даних – бізнес-логіка не повинна бути прив'язана до конкретної бази даних чи технології зберігання даних.

Тестованість – кожен шар має бути легким для тестування. Всі залежності повинні бути впроваджуваними, а бізнес-логіка повинна бути відокремлена від зовнішніх сервісів, що дозволяє легко проводити юніт-тестування.

Простота – архітектура повинна бути зрозумілою і мінімалістичною, з чітким поділом на функціональні блоки.

Clean Architecture складається з кількох основних рівнів (кілець) [11], які організовані таким чином, щоб забезпечити чітке розділення відповідальностей:

1. Core (ядро). У самому центрі знаходиться бізнес-логіка – це шар, який не залежить від жодних зовнішніх компонентів. У цьому шарі реалізуються основні бізнес-процеси, які не залежать від технологій, інтерфейсів або баз даних. Зазвичай це доменні об'єкти або сервіси, які виконують основні функції програми.

2. Use Cases (використання). Цей шар містить бізнес-логіку, яка реалізує сценарії взаємодії з користувачем чи іншими системами. Use case визначає, як користувачі або зовнішні сервіси взаємодіють з доменом та яку інформацію отримують чи передають.

3. Interface Adapters (адаптери інтерфейсів). Цей рівень включає адаптери та конвертери даних, що забезпечують взаємодію між зовнішнім світом (наприклад, вебсервером або користувацьким інтерфейсом) та внутрішньою логікою. Це може бути, наприклад, контролери вебзапитів, які перетворюють дані в зручний для ядра формат.

4. External Interfaces (зовнішні інтерфейси). На зовнішньому рівні знаходяться всі компоненти, які безпосередньо взаємодіють з зовнішнім

світом, такі як бази даних, вебсервери, системи авторизації, API-запити та інші технології. Ці елементи взаємодіють з бізнес-логікою через адаптери інтерфейсів.тНа рисунку 2.1 наведено структуру Clean Architecture:

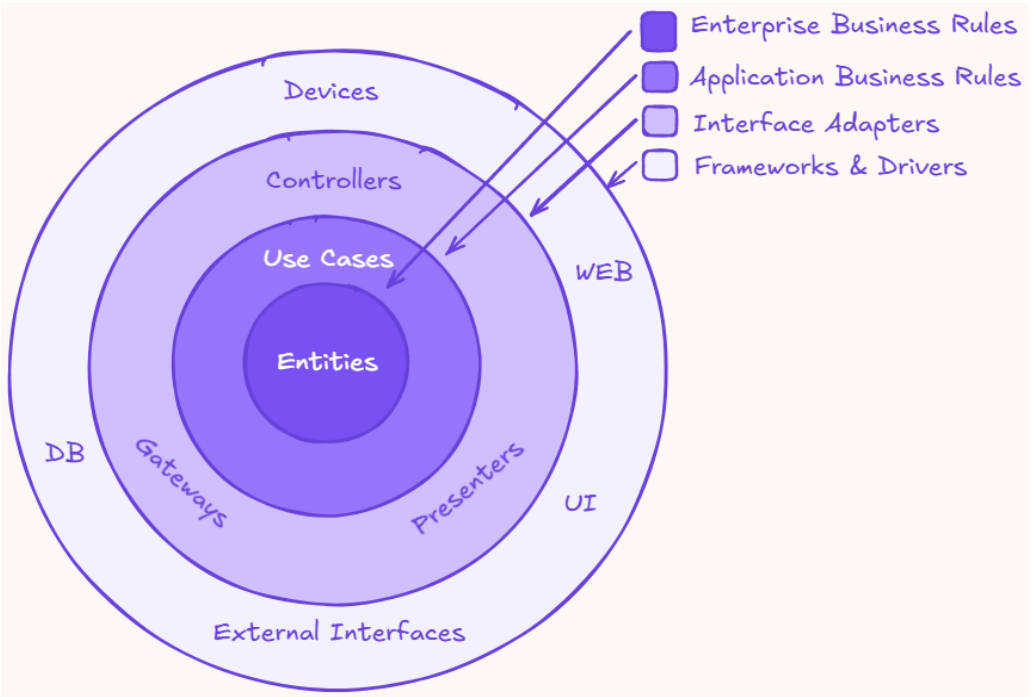

Рисунок 2.1 – Clean Architecture

Переваги Clean Architecture:

Завдяки структурованості коду новий функціонал впроваджується швидше, що позитивно впливає на продуктивність розробки та конкурентоспроможність рішення. Оскільки система краще підтримується і легше змінюється, витрати на її оновлення зменшуються. Структурований код і можливість тестування знижують кількість багів і підвищують якість кінцевого продукту. Завдяки модульності архітектури, можна швидко адаптувати систему до нових вимог або до зростання обсягу користувачів.

2.2.2. Вибір бекенд фреймворку

ASP.NET Core – це кросплатформовий, швидкий та сучасний фреймворк від Microsoft для створення вебдодатків і API. Він відомий своєю продуктивністю, безпекою та підтримкою масштабованості, що робить його

ідеальним для великих проєктів. Завдяки інтеграції з .NET Core, ASP.NET Core забезпечує високу продуктивність і підтримку різноманітних мов програмування, таких як C#. Фреймворк також активно підтримується та має сильну спільноту, що полегшує розробку складних рішень [12;13].

Express.js – простий у використанні фреймворк для Node.js, який дозволяє швидко створювати сервери та API. Він побудований на JavaScript, що робить його зручним для тих, хто працює з фронтендом, широко застосовується у проєктах, що вимагають високої продуктивності. Окрім того підтримує багато додаткових бібліотек, що полегшує та пришвидшує розробку.

Django – це популярний фреймворк для Python, створений для швидкої розробки вебдодатків та сайтів. Завдяки своїй архітектурі "batteries included", Django містить усе необхідне для реалізації бекенду, включаючи ORM, аутентифікацію та шаблонізатор. Це робить його зручним для стартапів і великих проєктів, які потребують швидкого розгортання. Крім того, Django підтримує високий рівень безпеки, що робить його популярним вибором для фінансових проєктів.

Spring Boot – це розширення популярного фреймворку Spring для Java, яке спрощує налаштування та конфігурацію додатків. Spring Boot дозволяє розробникам швидко розпочати проєкт без необхідності в складних налаштуваннях, надаючи зручне середовище для побудови вебдодатків. Він підтримує високий рівень масштабованості, що робить його популярним серед великих компаній, які працюють з мікросервісами та хмарними рішеннями.

Детальне порівняння фреймворків наведене в таблиці 2.1:

Таблиця 2.1 – Порівняння бекенд фреймворків.

Характеристика	**ASP.NET Core**	**Express.js**	**Django**	**Spring Boot**
Швидкість розгортання	Швидке розгортання завдяки Docker та підтримці контейнеризації	Дуже швидке, мінімум налаштувань	Швидке завдяки вбудованим інструментам	Швидке, але потребує більше конфігурацій

Підтримка мікросервісів	Добре підтримує завдяки модульності	Можливе, але не оптимальне	Обмежена підтримка	Відмінна підтримка
Підтримка REST API	Легка реалізація з автоматизованими інструментами	Легко реалізується, мінімалістичний API	Зручна підтримка через Django REST Framework	Висока підтримка з вбудованими модулями
Гнучкість у налаштуванні	Гнучкий, можливість налаштовувати будь-які аспекти	Дуже гнучкий та легкий у кастомізації	Досить жорсткий, багато рішень "з коробки"	Гнучкий, але потребує знання Spring
Можливості розширення	Висока через різноманітні бібліотеки та інтеграції	Широкі можливості завдяки екосистемі Node.js	Висока через численні пакети Python	Висока завдяки Spring екосистемі
Інтеграція з базами даних	Широка підтримка SQL та NoSQL (EF Core)	Легка інтеграція з різними базами даних	Вбудована підтримка баз даних через ORM	Різноманітна підтримка (через Hibernate)
Зручність у відладці	Висока, наявні потужні інструменти для відладки	Залежить від інструментів Node.js	Інтуїтивна відладка, але складніша для великих проєктів	Висока, з розширеними можливостями

ASP.NET Core виділяється серед інших фреймворків завдяки своїй високій продуктивності, підтримці масштабованості та сучасним інструментам безпеки [14]. На відміну від Express.js, ASP.NET Core підходить для розробки як малих, так і великих проєктів, де продуктивність має важливе значення. Порівняно з Django, ASP.NET Core є швидшим і краще підходить для корпоративних рішень. Spring Boot також є потужним інструментом, але ASP.NET Core краще відповідає сучасним вимогам безпеки.

2.3. Вибір технологій для реалізації системи збереження даних

Object-Relational Mapping (ORM) – це технологія, яка спрощує взаємодію з базами даних, забезпечуючи програмістам роботу з об'єктами та класами замість SQL-запитів. ORM-інструменти дозволяють автоматично

перетворювати об'єкти в таблиці бази даних і навпаки. Цей підхід робить розробку швидшою, спрощує обслуговування та знижує ризик помилок, пов'язаних із SQL-запитами.

Entity Framework Core (EF Core) – це сучасний ORM для .NET, що є кросплатформеною та легкою версією Entity Framework. EF Core дозволяє працювати з різними базами даних (наприклад, SQL Server, PostgreSQL, MySQL) та підтримує гнучкі підходи до розробки – як Code First, коли модель створюється з коду, так і Database First, коли моделі генеруються з існуючої бази даних [15]. Переваги Entity Framework Core:

- Автоматизацію запитів – дозволяє створювати запити на основі LINQ.
- Кросплатформеність – працює на Windows, Linux, macOS.
- Гнучкість та розширюваність – легко налаштовується під специфічні потреби проєкту.
- Зручна міграція – управління версіями бази даних завдяки механізму міграцій.

Таким чином, вибір EF Core як ORM може суттєво пришвидшити розробку, зробити взаємодію з базою даних простішою, більш контрольованою та забезпечити відповідність сучасним стандартам роботи з даними.

Далі потрібно визначитися з системою управління базами даних. Для цього проведемо аналіз найпопулярніших СУБД. Результати аналізу наведені в таблиці 2.2:

Таблиця 2.2 – Порівняння систем управління базами даних.

Параметр	**MS SQL Server**	**MySQL**	**PostgreSQL**
Ліцензія	Пропрієтарна (є безкоштовна версія Express)	Open Source (GNU GPL)	Open Source (PostgreSQL License)
Платформи	Windows, Linux, macOS	Windows, Linux, macOS	Windows, Linux, macOS

Масштабованість	Висока, особливо в корпоративних рішеннях	Висока, добре підходить для невеликих та середніх проєктів	Висока, особливо популярна для складних систем
Інтеграція з .NET	Глибока інтеграція, особливо з C# та ASP.NET	Можлива, але не така зручна	Підтримується, але без нативної інтеграції
Розширення функціональності	Через вбудовані функції та тригери	Можливість створення плагінів, тригерів	Широкий набір розширень, можливість створення функцій
Безпека	Потужні засоби захисту даних, вбудовані засоби шифрування та аудиту	Базові механізми безпеки	Висока безпека, шифрування та розширені ролі
Можливість резервного копіювання	Дуже розвинене та автоматизоване	Є функції, але потребують налаштування	Добре розвинене, але налаштування можуть бути складні
Підтримка транзакцій	Підтримка ACID, потужні механізми	Підтримка ACID	Підтримка ACID, багато додаткових можливостей
Підтримка складних запитів	Дуже ефективна, добре підходить для аналітики	Деколи потребує оптимізації	Чудова продуктивність, особливо з великими наборами даних
Підтримка індексів	Різноманітні типи індексів	Базові типи індексів	Підтримка складних типів індексів

MS SQL Server [16] має перевагу над MySQL та PostgreSQL у кількох аспектах, зокрема завдяки тісній інтеграції з технологіями Microsoft, зручності роботи з корпоративними даними, надійності безпеки та автоматизації резервного копіювання. Він особливо підходить для корпоративних систем, де важливі висока продуктивність та інтеграція з іншими продуктами Microsoft.

Хоча MySQL і PostgreSQL є потужними open-source рішеннями, MS SQL Server має перевагу в складних корпоративних і аналітичних середовищах.

2.4. Вибір фронтенд фреймворку

React – це бібліотека для створення користувацьких інтерфейсів, розроблена Facebook. Вона забезпечує розробникам можливість створювати компоненти, які оновлюються автоматично при зміні стану. React використовує віртуальний DOM, що підвищує продуктивність та забезпечує швидке оновлення змін на сторінці. Основні переваги включають високу гнучкість, однонаправлений потік даних, простоту інтеграції зі сторонніми бібліотеками, активну спільноту і багату екосистему [17].

Angular – це повноцінний фреймворк, розроблений Google, який використовує TypeScript. Він пропонує двонаправлене зв'язування даних, вбудовані сервіси для роботи з HTTP-запитами та інструменти для управління формами, маршрутизацією і залежностями. Angular забезпечує строгу структуру коду і глибоку функціональність для великих проєктів, однак може бути складнішим для освоєння порівняно з іншими фреймворками [18].

Vue – це прогресивний фреймворк для створення користувацьких інтерфейсів. Його легко інтегрувати в наявні проєкти, і він добре підходить для створення SPA (односторінкових застосунків). Vue має зрозумілу структуру, що дозволяє розробникам швидко його освоїти. Головними перевагами є висока гнучкість, простота і можливість індивідуального налаштування, хоча він поступається React за популярністю та обсягом доступних сторонніх бібліотек [19].

React, серед інших фреймворків, вирізняється гнучкістю, активною підтримкою спільноти, чудовою документацією та широкою екосистемою додаткових бібліотек. Він також забезпечує високу продуктивність завдяки віртуальному DOM та можливості легкого створення динамічних інтерфейсів. Це робить його оптимальним вибором для більшості сучасних вебдодатків.

MUI – це популярна бібліотека компонентів для React, що реалізує дизайн-систему Material Design, розроблену Google. Вона надає набір готових до використання компонентів, таких як кнопки, таблиці, діалогові вікна, форми та інші елементи інтерфейсу, що значно прискорює розробку сучасних вебдодатків. Використання MUI дозволяє забезпечити єдиний стиль інтерфейсу, що виглядає професійно і відповідає найкращим практикам вебдизайну. Завдяки гнучким можливостям налаштування, MUI дозволяє адаптувати елементи інтерфейсу до специфічних потреб проєкту, забезпечуючи при цьому високу продуктивність та кросбраузерність [20].

3. ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1. Моделювання системи

Для створення програмного забезпечення важливим кроком є моделювання системи, яке дозволяє краще зрозуміти її структуру, поведінку і взаємодії. Моделювання допомагає виявити вимоги до системи, ефективно організувати подальшу роботу над нею та уникнути багатьох помилок ще на ранніх етапах розробки.

Use Case Diagram – це діаграма, яка відображає функціональні можливості системи з точки зору користувачів (акторів). Вона показує взаємодію між акторами і системою через різні сценарії використання (use cases), що описують основні завдання або функції, які система виконує для досягнення певних цілей. Така діаграма допомагає зрозуміти, як користувачі взаємодіють із системою, і окреслює основні вимоги до її функціональності.

Виділимо всіх акторів та сценарії використання системи (таблиця 3.1).

Таблиця 3.1 – Актори та сценарії використання системи.

Актори	Сценарії використання
• Неавторизований користувач • Авторизований користувач (Оцінювач) • Критик • Адміністратор • База даних	• Реєстрація, Авторизація • Отримання фільму • Отримання списку фільмів • Отримання рекомендацій про фільми • Пошук фільму по певним критеріям • Створення, Редагування, Видалення фільму • Оцінювання фільму • Створення, Редагування, Видалення жанру фільму • Отримання списку жанрів фільмів • Отримання рецензії на фільм • Отримання списку рецензій на фільм • Оцінювання рецензії на фільм • Створення, Редагування, Видалення рецензії на фільм • Зміна ролі користувачів

Після цього створимо Use Case Diagram (рисунок 3.1).

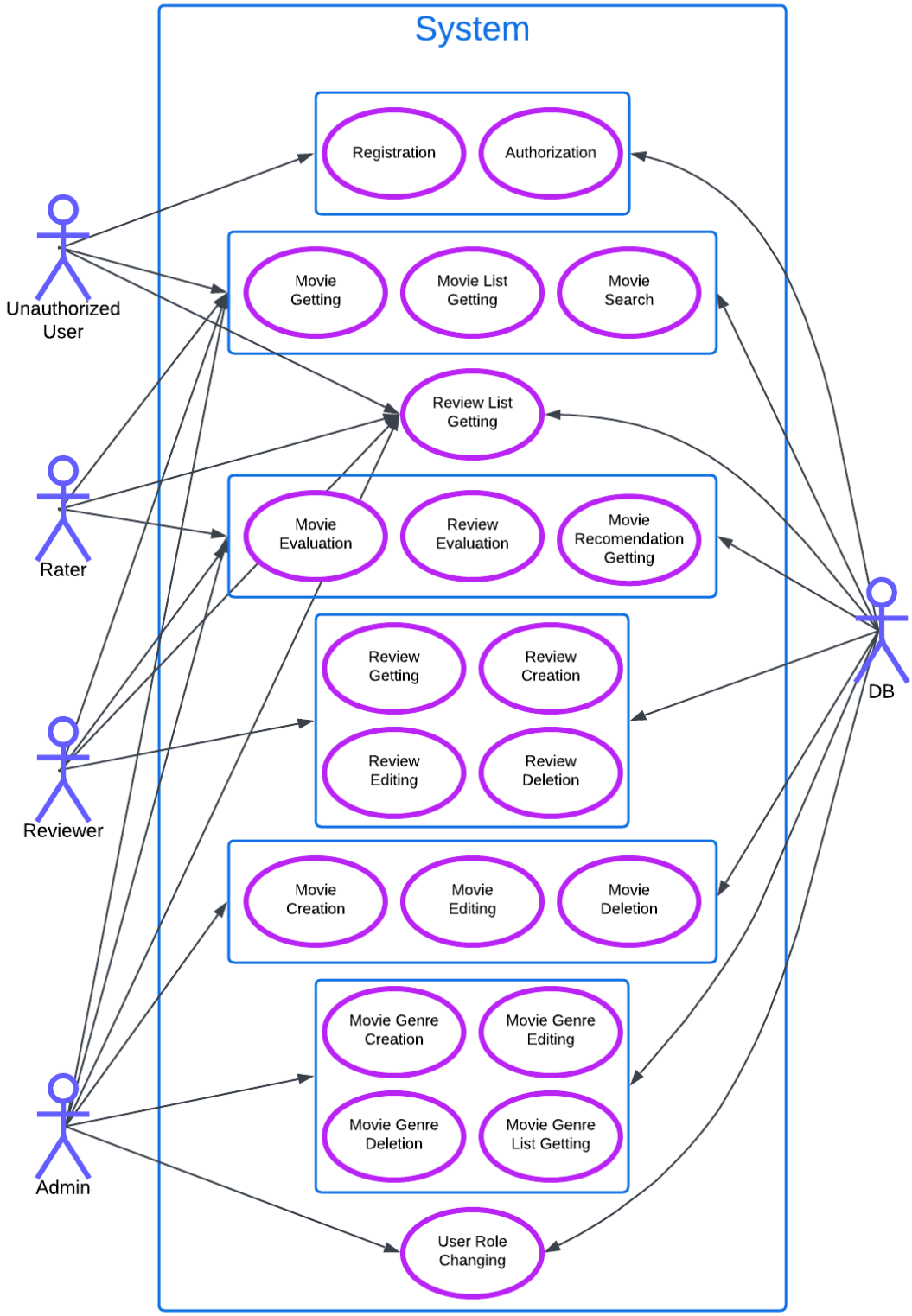

Рисунок 3.1 – Use Case Diagram

3.2. Проєктування системи

Відповідно до Client–Server архітектури весь вебзастосунок буде розділено на дві незалежні одна від одної частини: Client та Server.

Взаємодія між сервером на ASP.NET та клієнтом на React відбувається через HTTP-запити. Основні етапи взаємодії між ними:

- Клієнт відправляє запит на сервер
- Обробка запиту на сервері
- Обробка помилок і безпека
- Оновлення інтерфейсу

3.2.1. Проєктування бекенду

Відповідно до Clean Architecture все бекенд рішення буде складатися з п’яти проєктів:

- API – основний проєкт рішення, точка ініціалізації та запуску програми. У цьому проєкті будуть розміщені контролери, які відповідають за обробку HTTP-запитів та взаємодію з іншими шарами архітектури, забезпечуючи доступ до функціональності програми через API.
- Application – проєкт-бібліотека класів сервісів додатку та проміжного програмного забезпечення (middleware).
- Core – проєкт-бібліотека моделей, опцій, винятків, фільтрів, DTO, класів розширення та констант.
- Infrastructure – проєкт-бібліотека класів зовнішніх (які не відносяться до бізнес логіки системи) сервісів для роботи з JWT, зображеннями та хешами паролів.
- Persistence – проєкт-бібліотека класів які відповідають за взаємодію з базою даних.

Відповідність проєктів додатку до шарів Clean Architecture наведено на рисунку 3.2:

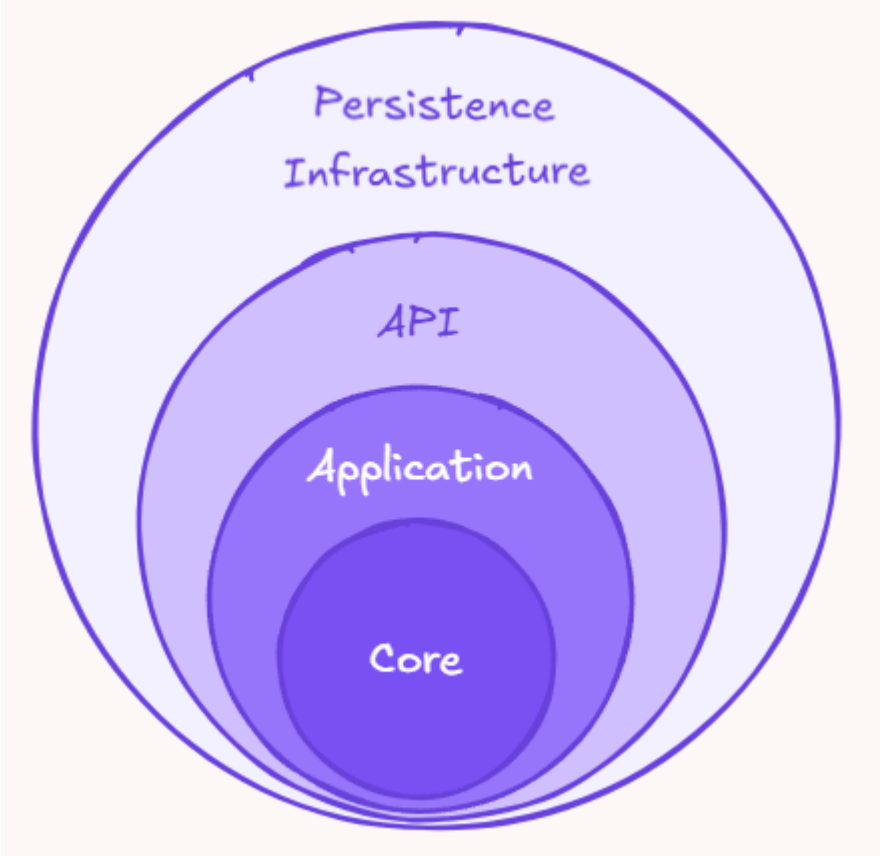

Рисунок 3.2 – Відповідність проєктів додатку до шарів CA.

3.2.2. Проєктування фронтенду

Фронтенд створюється відповідно до рекомендованої структури React проєктів [17]. Основними частинами типового проєкту React є:

Компоненти – це будівельні блоки інтерфейсу користувача в React-додатках. Вони можуть бути як функціональними, так і класовими та відповідають за відображення UI, обробку логіки, взаємодію з іншими компонентами. Компоненти можуть бути багаторазово використані, що сприяє модульності та зручності підтримки коду.

Стан і контекст – це механізми для управління даними в React-додатках. Стан визначає динамічні дані, що можуть змінюватися під час життєвого циклу компонента. Контекст дозволяє передавати дані між компонентами без необхідності вручну пропускати їх через кожен рівень вкладеності, забезпечуючи глобальний доступ до стану.

Маршрутизація – це механізм для створення багатосторінкових додатків на основі React без необхідності повного перезавантаження сторінки. За допомогою бібліотек, таких як React Router, можна визначати маршрути (URLs) і відображати відповідні компоненти на основі поточного шляху.

Стилізація – це процес додавання CSS або інших стилістичних рішень для оформлення зовнішнього вигляду компонентів. У React використовують різні підходи, як-от глобальні стилі, CSS Modules, бібліотеки типу Styled Components чи інші методи, що дозволяють створювати стилі безпосередньо у компонентах.

API-запити та сервіси – це логіка для взаємодії з зовнішніми API, що дозволяє отримувати дані, надсилати запити чи виконувати бізнес-логіку. API-запити організовуються в окремі функції або модулі (сервіси), щоб спростити управління даними та їх обробку в компонентах.

3.3. Реалізація бекенду

Відповідно було створено п’ять проєктів кожен з яких відповідає за певні завдання.

На рисунку 3.3 наведена структура проєкту Reviews.API:

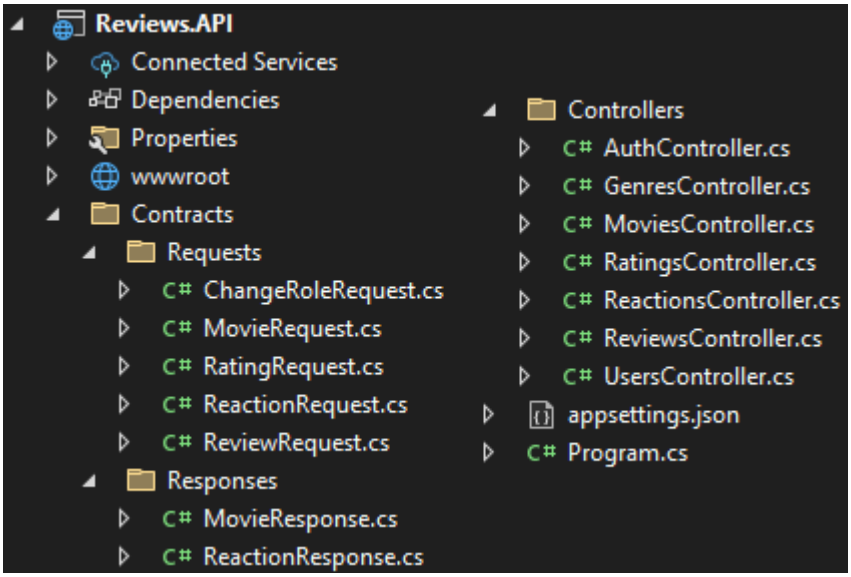

Рисунок 3.3 – Структура проєкту Reviews.API.

Детальна інформації про кожен клас проєкту Reviews.API наведена в таблиці 3.2:

Таблиця 3.2 – Класи та їх функціонал проєкту Reviews.API.

Клас	**Відповідальність / Функціональність**
ChangeRoleRequest.cs	Клас даних запиту для зміни ролі користувача
MovieRequest.cs	Клас даних запиту для створення та редагування фільму
RatingRequest.cs	Клас даних запиту для оцінювання фільму
ReactionRequest.cs	Клас даних запиту для оцінювання рецензії на фільм
ReviewRequest.cs	Клас даних запиту для створення та редагування рецензії на фільм
MovieResponse.cs	Клас даних відповіді на запит отримання інформації про фільм
ReactionResponse.cs	Клас даних відповіді на запит отримання інформації про реакцію на рецензію
AuthController.cs	Клас-контролер для обробки запитів авторизації та реєстрації користувачів
GenresController.cs	Клас-контролер для обробки запитів створення, редагування, видалення та отримання списку жанрів фільмів
MoviesController.cs	Клас-контролер для обробки запитів створення, редагування, видалення та отримання списку фільмів або рекомендацій про фільми
RatingsController.cs	Клас-контролер для обробки запитів створення, редагування, видалення та отримання оцінок користувачів на фільми, отримання кількості оцінок та середнього значення для певного фільму
ReactionsController.cs	Клас-контролер для обробки запитів створення, редагування, видалення та отримання реакцій на рецензії
ReviewsController.cs	Клас-контролер для обробки запитів створення, редагування, видалення та отримання списку рецензій на фільми
UsersController.cs	Клас-контролер для обробки запитів отримання списку користувачів та зміни їх ролі
Program.cs	Точка реєстрації сервісів, налаштування контролерів, авторизації, міграцій, статичних файлів, підключення до БД, ініціалізації та запуску програми.

Dependency Injection (DI) у ASP.NET Core — це техніка впровадження залежностей, яка дозволяє управляти ними між компонентами застосунку. Замість того, щоб класи створювали свої залежності самостійно, вони отримують їх (через конструктор) із зовнішнього контейнера. Реєстрація всіх сервісів від яких залежать інші класи відбувається у класі Program.cs.

Контролери в ASP.NET Core — це компоненти, що відповідають за обробку вхідних HTTP-запитів, виконання логіки бізнес-процесів і повернення відповідей клієнтам (наприклад, у вигляді HTML-сторінок або JSON-даних) [21]. Вони є основною частиною архітектури MVC [22] (Model-View-Controller) і взаємодіють з моделями та представленнями для керування потоком даних у вебдодатку.

На рисунку 3.4 наведена структура проєкту Reviews.Application:

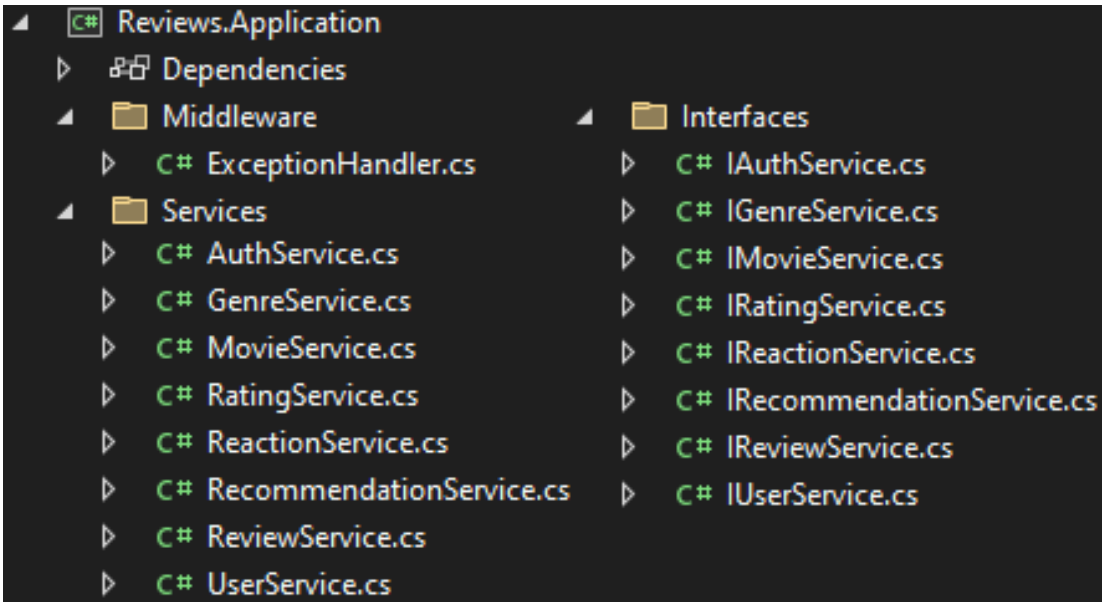

Рисунок 3.4 – Структура проєкту Reviews.Application.

Детальна інформації про кожен клас проєкту Reviews.Application наведена в таблиці 3.3:

Таблиця 3.3 – Класи та їх функціонал проєкту Reviews.Application.

Клас	**Відповідальність / Функціональність**
ExceptionHandler.cs	Клас проміжного програмного забезпечення для обробки винятків
AuthService.cs	Клас-сервіс для авторизації та реєстрації користувачів
GenreService.cs	Клас-сервіс для створення, редагування, видалення та отримання списку жанрів фільмів
MovieService.cs	Клас-сервіс для створення, редагування, видалення та отримання списку фільмів
RatingService.cs	Клас-сервіс для створення, редагування, видалення та отримання оцінок користувачів на фільми, отримання кількості оцінок та середнього значення для певного фільму
ReactionService.cs	Клас-сервіс для створення, редагування, видалення та отримання реакцій на рецензії
RecommendationService.cs	Клас-сервіс для отримання рекомендацій про фільми
ReviewService.cs	Клас-сервіс для створення, редагування, видалення та отримання списку рецензій на фільми
UserService.cs	Клас-сервіс для отримання списку користувачів та зміни їх ролі

Сервіси в ASP.NET Core — це класи, що реалізують бізнес-логіку або виконують допоміжні завдання (наприклад, обробка даних, логування, робота з файлами). Вони впроваджуються за допомогою Dependency Injection, що спрощує управління залежностями та тестування. Сервіси надають функціонал, який можна легко використовувати у контролерах або інших частинах додатка, забезпечуючи модульність і повторне використання коду.

На рисунку 3.5 наведена структура проєкту Reviews.Core:

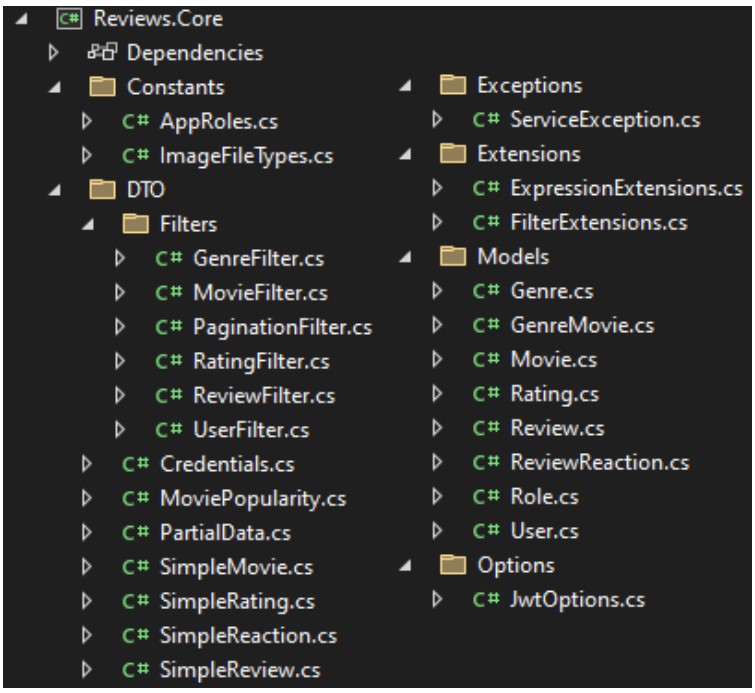

Рисунок 3.5 – Структура проєкту Reviews.Core.

Детальна інформації про кожен клас проєкту Reviews.Core наведена в таблиці 3.4:

Таблиця 3.4 – Класи та їх функціонал проєкту Reviews.Core.

Клас	**Відповідальність / Функціональність**
AppRoles.cs	Константи ролей системи
ImageFileTypes.cs	Константи типів файлів зображень
GenreFilter.cs	Фільтр для жанрів фільмів
MovieFilter.cs	Фільтр для фільмів
PaginationFilter.cs	Фільтр для реалізації пагінації
RatingFilter.cs	Фільтр для оцінок користувачів
ReviewFilter.cs	Фільтр для рецензій користувачів
UserFilter.cs	Фільтр для користувачів
Credentials.cs	Клас облікових даних користувача
MoviePopularity.cs	Клас для популярності фільмів
PartialData.cs	Клас для представлення частини даних
SimpleMovie.cs	Спрощена модель фільму
SimpleRating.cs	Спрощена модель рейтингу фільму
SimpleReaction.cs	Спрощена модель реакції на рецензію
SimpleReview.cs	Спрощена модель рецензії
ServiceException.cs	Клас виняток, який може виникнути при виконанні бізнес логіки

ExpressionExtensions.cs	Клас для розширення функціональності виразів
FilterExtensions.cs	Клас для розширення функціональності фільтрів
Genre.cs	Модель жанру
GenreMovie.cs	Модель відношення жанрів до фільмів
Movie.cs	Модель фільму
Rating.cs	Модель оцінки фільму
Review.cs	Модель рецензії
ReviewReaction.cs	Модель реакції на рецензію
Role.cs	Модель ролі користувача
User.cs	Модель користувача
JwtOptions.cs	Клас опцій для JWT

Моделі в ASP.NET Core — це класи, що представляють дані програми. Вони використовуються для опису структури даних, їхньої валідації та взаємодії з базою даних. У контексті архітектури MVC (Model-View-Controller) моделі відповідають за обробку даних: збереження, отримання, валідацію тощо, забезпечуючи взаємодію з контролерами та представленнями для відображення інформації користувачам.

На рисунку 3.6 наведена структура проєкту Reviews.Infrastructure:

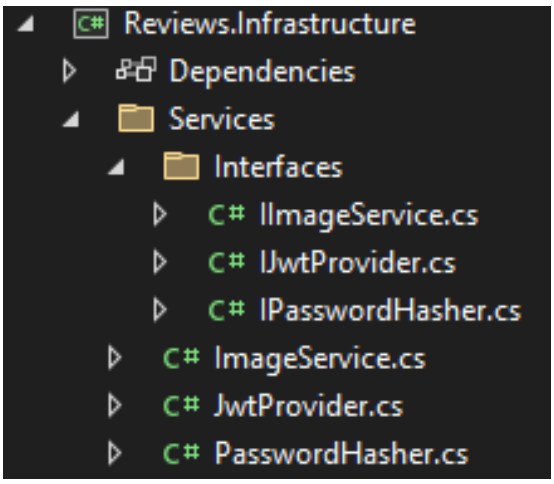

Рисунок 3.6 – Структура проєкту Reviews.Infrastructure.

Детальна інформації про кожен клас проєкту Reviews.Infrastructure наведена в таблиці 3.5:

Таблиця 3.5 – Класи та їх функціонал проєкту Reviews.Infrastructure.

Клас	**Відповідальність / Функціональність**
ImageService.cs	Клас-сервіс для обробки та зберігання зображень (постерів фільмів)
JwtProvider.cs	Клас для створення JWT
PasswordHasher.cs	Клас для створення та перевірки хешів паролів

На рисунку 3.7 наведена структура проєкту Reviews.Persistence:

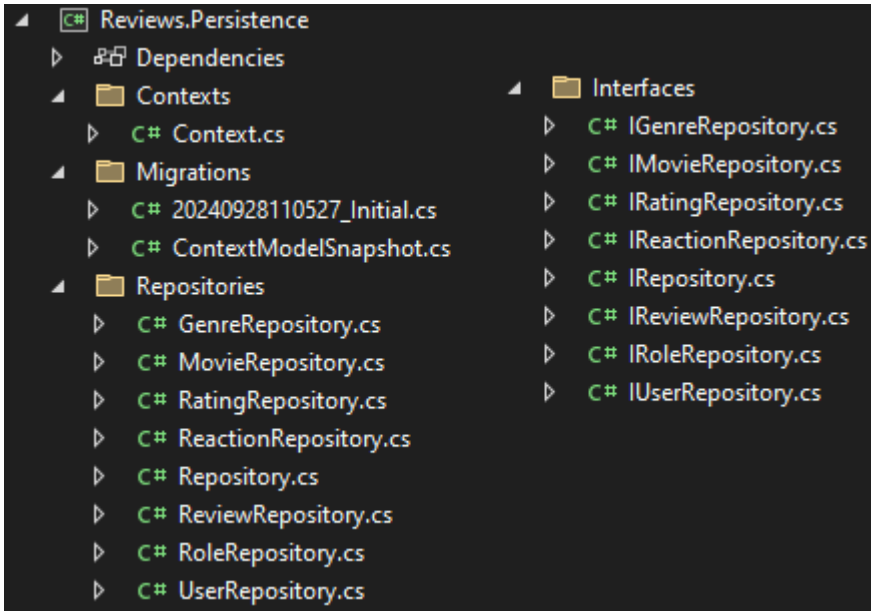

Рисунок 3.7 – Структура проєкту Reviews.Persistence.

Детальна інформації про кожен клас проєкту Reviews.Persistence наведена в таблиці 3.6:

Таблиця 3.6 – Класи та їх функціонал проєкту Reviews.Persistence.

Клас	**Відповідальність / Функціональність**
Context.cs	Клас для реалізації контексту бази даних (EF Core)
Repository.cs	Клас для реалізації базового функціоналу репозиторію
GenreRepository.cs	Клас-репозиторій жанрів фільмів
MovieRepository.cs	Клас-репозиторій фільмів
RatingRepository.cs	Клас-репозиторій рейтингу фільмів
ReactionRepository.cs	Клас-репозиторій реакцій на рецензії
ReviewRepository.cs	Клас-репозиторій рецензій фільмів
RoleRepository.cs	Клас-репозиторій ролей користувачів
UserRepository.cs	Клас-репозиторій користувачів

Контекст в Entity Framework Core – це клас, що успадковує DbContext і представляє сеанс взаємодії з базою даних. Він забезпечує доступ до таблиць бази даних за допомогою властивостей DbSet<T>, дозволяючи виконувати запити, додавати, змінювати та видаляти дані. Контекст також управляє відстеженням змін, кешуванням, транзакціями й обробкою запитів до бази даних, виступаючи як основний об’єкт для взаємодії між додатком та джерелом даних [15].

Репозиторій – це шаблон проєктування, що створює рівень абстракції між бізнес-логікою програми та шаром доступу до даних. Репозиторій надає стандартизований інтерфейс для роботи з даними, що дозволяє зменшити залежність від конкретної технології доступу до даних та спрощує тестування і підтримку коду. Репозиторії часто використовуються разом із DbContext для організації чіткої структури доступу до даних у програмі.

Так як було використано підхід Code First (спочатку пишеться код а потім EF Core створює БД) маємо наступну структуру бази даних (рисунок 3.8):

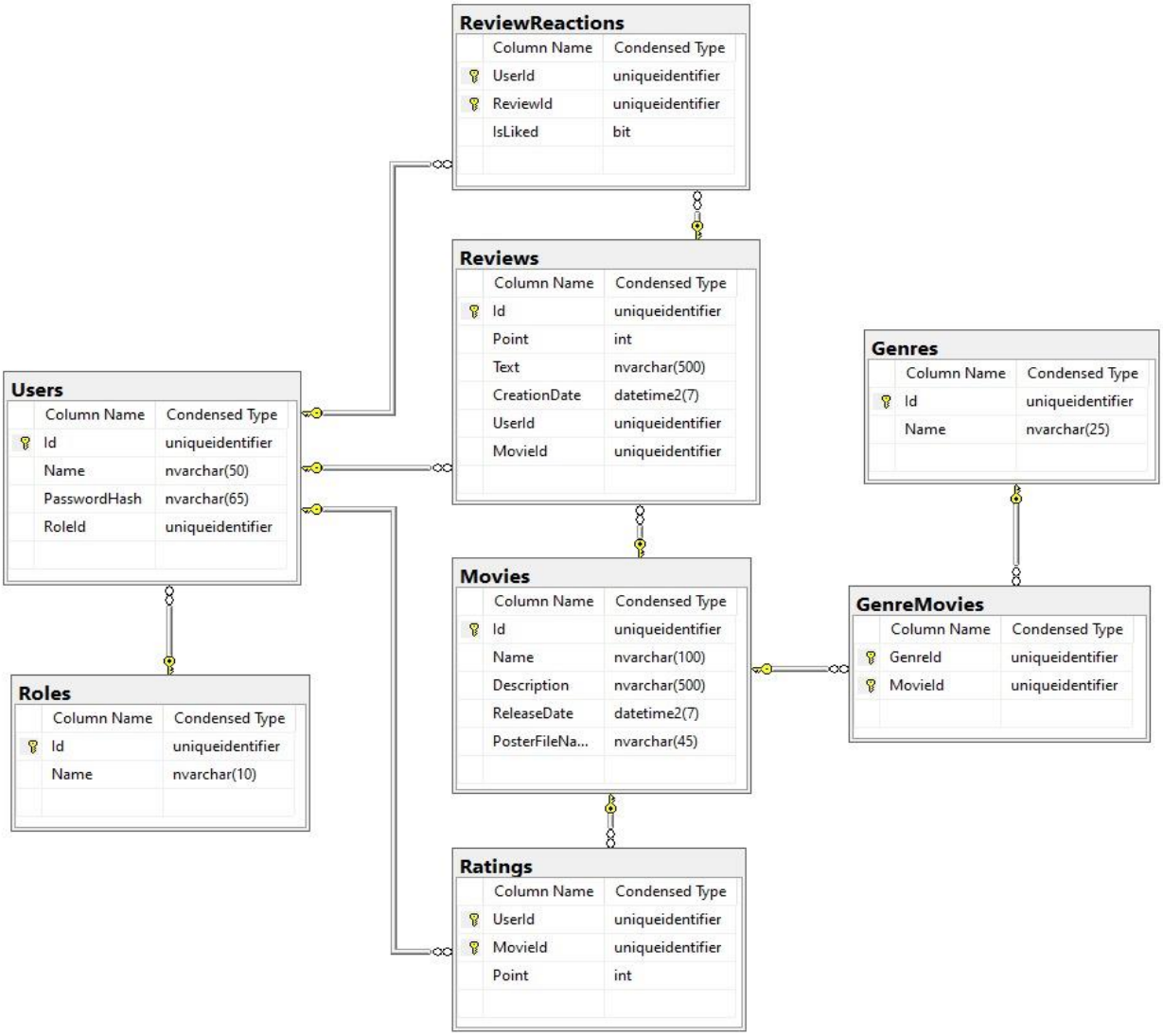

Рисунок 3.8 – ER Diagram бази даних системи.

3.4. Реалізація рекомендаційного алгоритму

Рекомендаційний алгоритм має три основні частини:

1. Формування вхідних даних.
2. Обробка даних.
3. Формування результату.

За формування вхідних даних відповідають репозиторії, а за обробку та формування результату клас-сервіс RecommendationService. Весь код рекомендаційного алгоритму наведено в Додатку А. Розглянемо детально кожний етап алгоритму.

3.4.1. Формування вхідних даних

Відповідно до гібридного підходу побудови рекомендаційних систем потрібно враховувати як подібність користувачів, так і відповідність самих об'єктів (фільми, жанри, рецензії), які юзери оцінюють або з якими взаємодіють. Це дозволяє створювати більш точні та персоналізовані рекомендації, комбінуючи переваги контент-орієнтованого підходу і колаборативної фільтрації.

Для цього сформуємо відповідні набори вхідних даних (таблиця 3.7).

Таблиця 3.7 – Набори вхідних даних.

Набір даних	**Програмна реалізації отримання даних**
Масив елементів (Id фільму, оцінка) для відповідного користувача.	RatingRepository .GetForUserAsync()
Словник елементів (Id подібного користувача, (Масив елементів (Id фільму, оцінка))) для відповідного користувача, де подібним користувачем вважається той, хто оцінив хоч один фільм оцінений заданим користувачем. Причому масив формується для фільмів, які оцінив відповідний користувач.	RatingRepository .GetRelatedRatingsAsync()
Масив елементів (Id рецензії, реакція) для відповідного користувача	ReactionRepository .GetForUserAsync()
Словник елементів (Id подібного користувача, (Масив елементів (Id рецензії, реакція))) для відповідного користувача, де подібним користувачем вважається той, хто оцінив хоч одну рецензію оцінену заданим користувачем. Причому масив формується для рецензій, які оцінив відповідний користувач.	ReactionRepository .GetRelatedReactionsAsync()
Словник елементів (Id подібного користувача, (Масив елементів (Id фільму, оцінка))) для відповідного користувача, де подібним користувачем вважається той, хто оцінив хоч один фільм оцінений заданим користувачем. Причому масив формується для фільмів, які відповідний користувач не оцінював.	RatingRepository .GetUnrelatedRatingsAsync()
Словник елементів (Id фільму, (Масив елементів рецензій (оцінка, кількість лайків на рецензію, кількість дизлайків на рецензію)))	ReviewRepository .GetSimpledStacksAsync()

Масив даних про фільми.	MovieRepository .GetSimpledListAsync()
Словник елементів (Id жанру, кількість фільмів цього жанру які сподобалися користувачу) для відповідного користувача.	GenreRepository .GetFavoriteGenreIdsAsync()

3.4.2. Обробка даних

Обробка даних відбувається в методі GetRecommendationsAsync() класу RecommendationService (рисунок 3.9). Спочатку формується предикат за фільтром фільмів (по назві, жанру тощо). Потім створюється порожній словник елементів (Id фільму, рекомендаційна оцінка), який в подальшому буде заповнено та оновлено відповідно до вхідних наборів даних.

```
public async Task<PartialData<(Movie, int)>> GetRecommendationsAsync(Guid targetUserId, MovieFilter filter)
{
    Expression<Func<Movie, bool>>? movieRestriction = filter.GetPredicate();
    ConcurrentDictionary<Guid, double> movieScores = [];

    await AdjustScoresByRatingsAsync(movieScores, targetUserId, movieRestriction);
    await AdjustScoresByReviewsAsync(movieScores, movieRestriction);
    await AdjustScoresByGenresAsync(movieScores, targetUserId, movieRestriction);

    return await GenerateResultAsync(movieScores, filter);
}
```

Рисунок 3.9 – Метод, який реалізовує рекомендаційний алгоритм.

AdjustScoresByRatingsAsync() – метод для врахування оцінок користувачів при формуванні остаточної рекомендаційної оцінки.

Метод AdjustScoresByRatingsAsync() в свою чергу спочатку викликає метод GetSimilarUsersAsync() для розрахунку подібності користувачів на основі їх оцінок на фільми та реакцій на рецензії. Коефіцієнт подібності розраховується на основі коефіцієнта кореляції Пірсона [23] та коефіцієнта подібності розмірностей вхідних даних:

$$p = \frac{l_{min}}{l_{max}}; \quad r = \frac{\Sigma(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\Sigma(x_i - \bar{x})^2 \Sigma(y_i - \bar{y})^2}}; \quad pp = p * r$$

Де:

r – коефіцієнт кореляції Пірсона.

x_i – значення змінної x у вибірці.

$\bar{x}$ – середнє значення змінної x.

y_i – значення змінної y у вибірці.

$\bar{y}$ – середнє значення змінної y.

p – коефіцієнта подібності розмірностей вибірок.

l_{min} – довжина меншої вибірки.

l_{max} – довжина більшої вибірки.

pp – коефіцієнт подібності.

Цю логіку реалізують два перевантажені методи PearsonCorrelation() (для подібності користувачів на основі їх оцінок на фільми та реакцій на рецензії).

Після розрахування подібності користувачів на основі отриманих даних та словника елементів (Id подібного користувача, (Масив елементів (Id фільму, оцінка))) для відповідного користувача, де подібним користувачем вважається той, хто оцінив хоч один фільм оцінений заданим користувачем, причому масив формується для фільмів, які відповідний користувач не оцінював, формується початкове значення рекомендаційної оцінки.

Переходимо до другої частини – методу AdjustScoresByReviewsAsync(). AdjustScoresByReviewsAsync() – метод для врахування загальної популярності фільмів на основі рецензій та реакцій на ці рецензії.

На основі вхідних даних, а саме словника елементів (Id фільму, (Масив елементів рецензій (оцінка, кількість лайків на рецензію, кількість дизлайків на рецензію))) у методі GetMoviesPopularityAsync() вираховується загальна популярність фільмів.

Відповідно до загальної популярності фільмів вносяться зміни до рекомендаційної оцінки.

Переходимо до третьої частини – методу AdjustScoresByGenresAsync(). AdjustScoresByGenresAsync() – метод для врахування улюблених жанрів відповідного користувача.

На основі вхідних даних, а саме словника елементів (Id жанру, кількість фільмів цього жанру які сподобалися користувачу), вносяться зміни до рекомендаційної оцінки.

Всі розрахунки залежать від певних констант та коефіцієнтів наведених на рисунку 3.10:

```
private const int PositivePoint = 4;
private const int AveragePoint = 3;

private const double PopularFilmMinScale = 1;
private const double PopularFilmGap = 0.2;

private const double PopularGenreMinScale = 1.5;
private const double PopularGenreGap = 0.5;

private const double RatingSimilarityImpact = 0.7;
private const double ReactionSimilarityImpact = 0.3;
```

Рисунок 3.10 – Константи та коефіцієнти.

3.4.3. Формування результату

Останнім етапом алгоритму є формування результату. За це відповідає метод GenerateResultAsync(). На основі створеного словника елементів (Id фільму, рекомендаційна оцінка) створюється масив елементів (Id фільму, відносна рекомендаційна оцінка) за допомогою функції перетворення наведеній на рисунку 3.11:

```
static int ScaleToRange(double oldValue, double oldMin, double oldMax, double newMin, double newMax)
{
    if (oldMin == oldMax)
    {
        return (int)newMax;
    }

    return (int)(newMin + (oldValue - oldMin) * (newMax - newMin) / (oldMax - oldMin));
}
```

Рисунок 3.11 – Функція перетворення.

Де:

oldValue – старе значення (рекомендаційна оцінка).

oldMin – найменше значення із створеного словника.

oldMax – найбільше значення із створеного словника.

newMin – нове найменше значення (1).

newMax – нове найбільше значення (100).

Після цього з БД по Id фільму знаходиться вся додаткова необхідна інформація. Звідси маємо як результат масив елементів (дані про фільм, відносна рекомендаційна оцінка).

3.5. Реалізація фронтенду

Відповідно до рекомендованої структури React проєктів клієнтська частина додатку складається з таких ключових елементів:

- Компоненти – будівельні блоки інтерфейсу додатку.
- Контекст – полегшення та покращення взаємодії між компонентами. Реалізації авторизації та аутентифікації.
- Моделі – сутності (класи для даних).
- Сервіси – реалізовують логіку взаємодії з зовнішніми API, що дозволяє отримувати дані, надсилати запити на сервер.
- React Router – реалізація маршрутизації по додатку.
- Допоміжні класи та функції.

Загальна структура клієнтської частини наведена на рисунку 3.12:

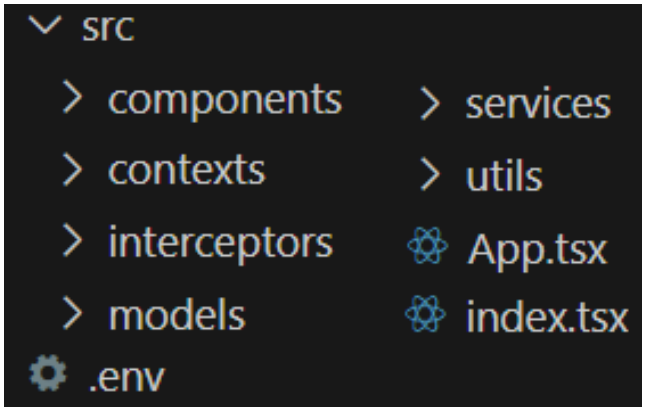

Рисунок 3.12 – Загальна структура фронтенду.

Кожний компонент відповідає за певні функції. Детальна інформації про них наведена в таблиці 3.8:

Таблиця 3.8 – Компоненти та їх призначення.

Компоненти	**Призначення**
AdminPage	Візуальне відображення сторінки адміністратора.
ReviewsAppBar	Верхня навігаційна панель.
LoginForm	Форма авторизації.
RegisterForm	Форма реєстрації.
CreateGenre	Форма для створення жанрів.
GenreContainer	Елемент-контейнер для інших компонентів пов’язаних із жанрами.
GenreSearch	Пошуковий елемент для жанрів.
GenreTable	Таблиця із списком жанрів.
UpdateGenre	Форма оновлення жанру.
CreateMovie	Форма створення фільму.
MovieCard	Картка фільму.
MovieContainer	Елемент-контейнер для інших компонентів пов’язаних із фільмами.
MovieGrid	Елемент для відображення набору карток фільми.
MovieSearch	Пошуковий елемент для фільмів.
MovieView	Сторінка з інформацією про фільм.
UpdateMovie	Форма оновлення сутності фільму.
CreateReview	Форма створення сутності фільму.
OwnReview	Картка рецензії на фільм для поточного користувача.
ReviewCard	Картка рецензії на фільми.
ReviewGrid	Елемент для відображення набору карток рецензій на фільм.
ReviewTab	Елемент для переходу між власною рецензією та всіма рецензіями на фільм.
UpdateReview	Форма оновлення рецензії на фільм.
RatingCard	Картка рейтингу фільму.
UserContainer	Елемент-контейнер для інших компонентів пов’язаних із користувачами.
UserSearch	Пошуковий елемент для користувачів.
UserTable	Таблиця із списком користувачів.
Loader	Елемент який показує стан завантаження даних необхідних іншим компонентам.
Router	Маршрутизація

Як було вказано вище для взаємодії з серверною частиною створено відповідні сервіси (таблиця 3.9).

Таблиця 3.9 – Сервіси та їх призначення.

Сервіси	**Призначення**
AuthService	Реєстрація та авторизація користувача.
GenreService	Отримання відфільтрованого списку жанрів. Створення, редагування, видалення жанрів.
LoginService	Реєстрація та авторизація користувача.
MovieService	Отримання відфільтрованого списку фільмів. Отримання даних про фільм необхідних для відображення сторінки фільму. Отримання рекомендацій про фільми. Створення, редагування, видалення фільмів.
RatingService	Отримання загальної кількості оцінок на фільм. Отримання середньої оцінки для фільму. Отримання оцінки на фільм для певного користувача. Створення, редагування, видалення оцінок на фільми.
ReactionService	Отримання реакції на рецензію для певного користувача. Створення, редагування, видалення реакцій на рецензії.
ReviewService	Отримання відфільтрованого списку рецензій. Отримання середньої оцінки для фільму від рецензентів. Отримання рецензії на фільм для поточного користувача. Створення, редагування, видалення рецензій.
UserService	Отримання відфільтрованого списку користувачів. Оновлення даних про користувача.

Для реалізації сервісів було використано бібліотеку Axios [24]. Axios – це бібліотека для роботи з HTTP-запитами, яка дозволяє легко інтегрувати клієнтську частину додатка з API. Вона забезпечує простий і гнучкий синтаксис для виконання запитів GET, POST, PUT, DELETE та інших. Axios автоматично обробляє JSON-дані, підтримує налаштування заголовків, таймаутів і авторизації, а також дозволяє використовувати перехоплювачі (interceptors) для обробки запитів і відповідей. Приклад перехоплювача авторизації наведено на рисунку 3.13:

```
authAxios.interceptors.request.use(
  (request) => {
    const token = Cookies.get("auth");

    if (token) {
      request.headers["Authorization"] = `Bearer ${token}`;
    }

    return request;
  },
  (error) => {
    return Promise.reject(error);
  }
);
```

```
authAxios.interceptors.response.use(
  (response) => {
    return response;
  },
  (error) => {
    if (error.response.status === 401) {
      LoginService.logout();
    }
    return Promise.reject(error);
  }
);
```

Рисунок 3.13 – Перехоплювач авторизації.

Розглянемо зовнішній вигляд інтерфейсу користувача додатку.

Кожний користувач має доступ до головної сторінки (рисунок 3.14). На ній знаходиться список із картками фільмів та пошукова панель:

- Пошук по назві.
- Пошук по жанрам.
- Кількість карток для відображення.

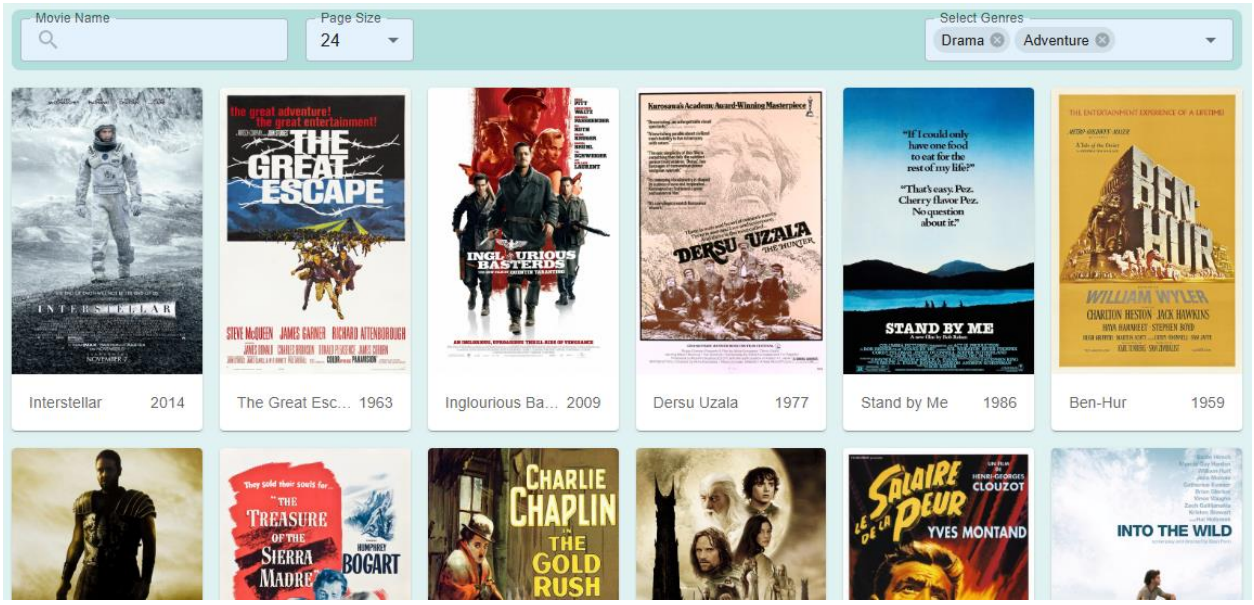

Рисунок 3.14 – Головна сторінка додатку.

Після знаходження необхідного фільму користувач може перейти на його сторінку (рис. 3.15). Для цього потрібно просто натиснути на відповідну картку.

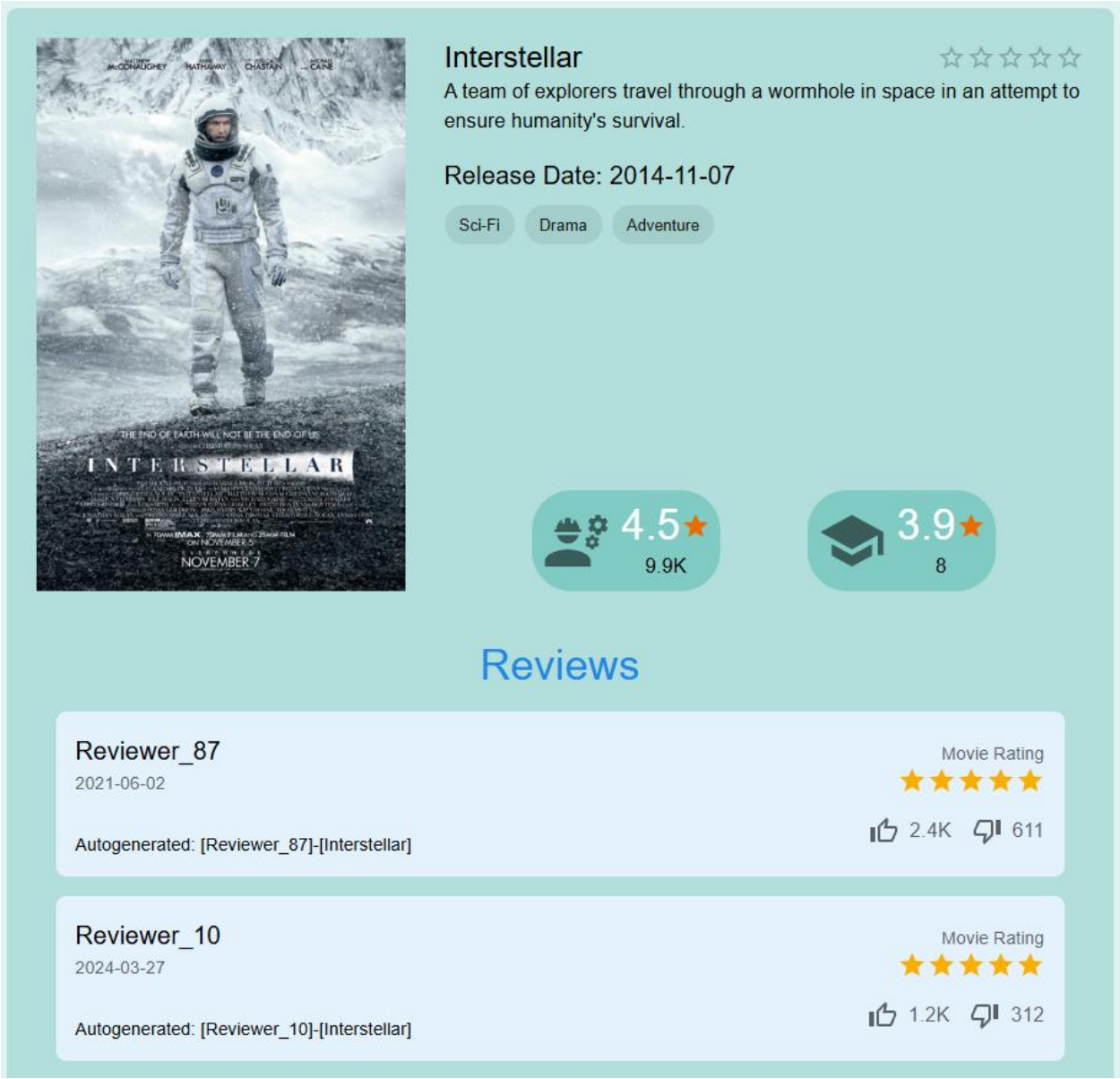

Рисунок 3.15 – Сторінка фільму.

На даній сторінці знаходиться детальна інформація про фільм, його усереднені оцінки та рецензії на нього. Оцінювати фільми та виставляти реакції на рецензії можуть тільки зареєстровані користувачі, а створювати самі рецензії тільки користувачі із роллю рецензента.

Для авторизації у системі потрібно перейти на сторінку авторизації/реєстрації вибравши відповідний пункт із навігаційного списку (рисунок 3.16).

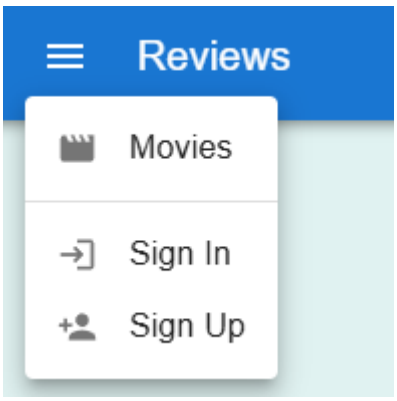

Рисунок 3.16 – Навігація по системі.

Для авторизації/реєстрації потрібно заповнити відповідну форму (рисунок 3.17).

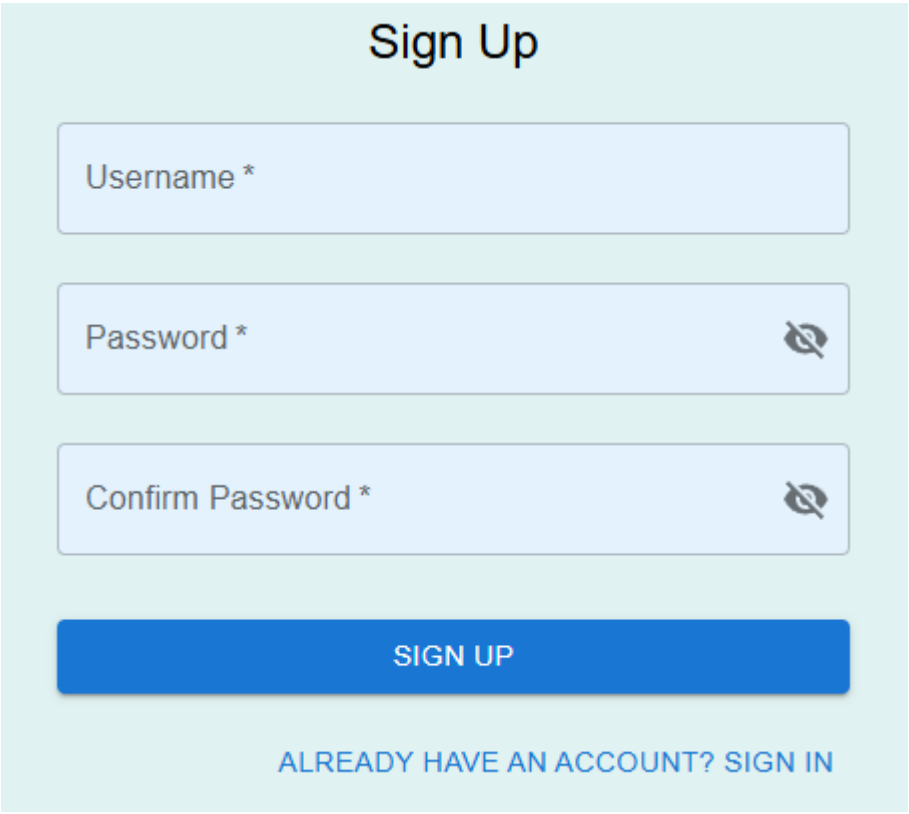

Рисунок 3.17 – Форма реєстрації користувача.

Кожний користувач із роллю рецензента може створити рецензію на фільм. Для цього потрібно на сторінці фільму заповнити форму рецензії (рис. 3.18).

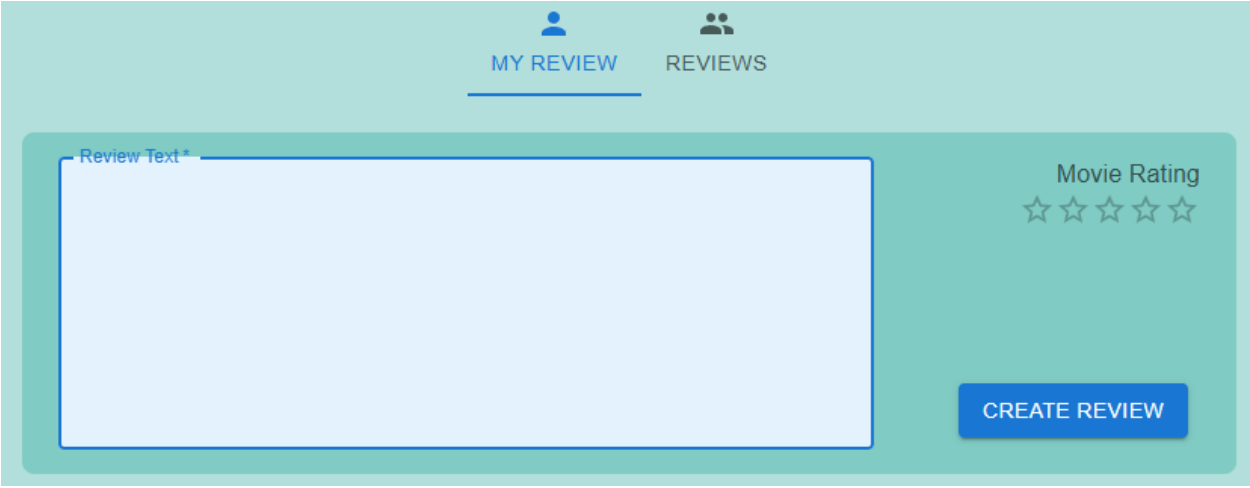

Рисунок 3.18 – Форма створення рецензії на фільм.

Також кожний зареєстрований користувач має змогу отримати персональні рекомендації про фільми (рис. 3.19). Для кожного фільму є відносний рекомендаційний рейтинг який змінюється від 100 до 1.

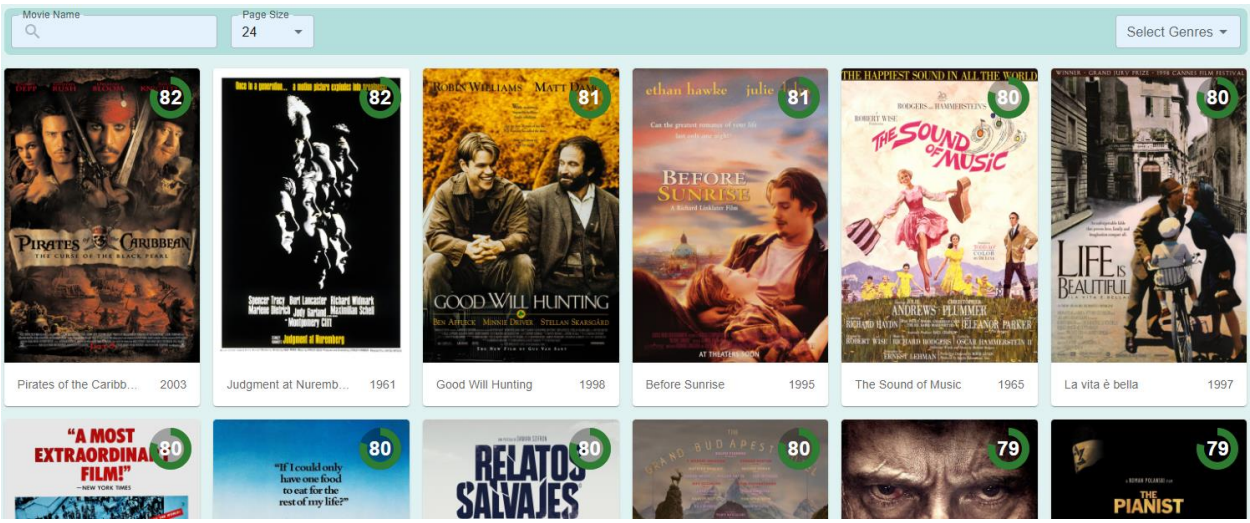

Рисунок 3.19 – Сторінка з рекомендаціями.

Для ефективного управління системою існує сторінка адміністратора, перейти на яку може тільки користувач із відповідною роллю (рис.3.20). На ній знаходиться список користувачів, список жанрів, форма створення нового жанру та кнопка переходу на сторінку із формою створення нового фільму.

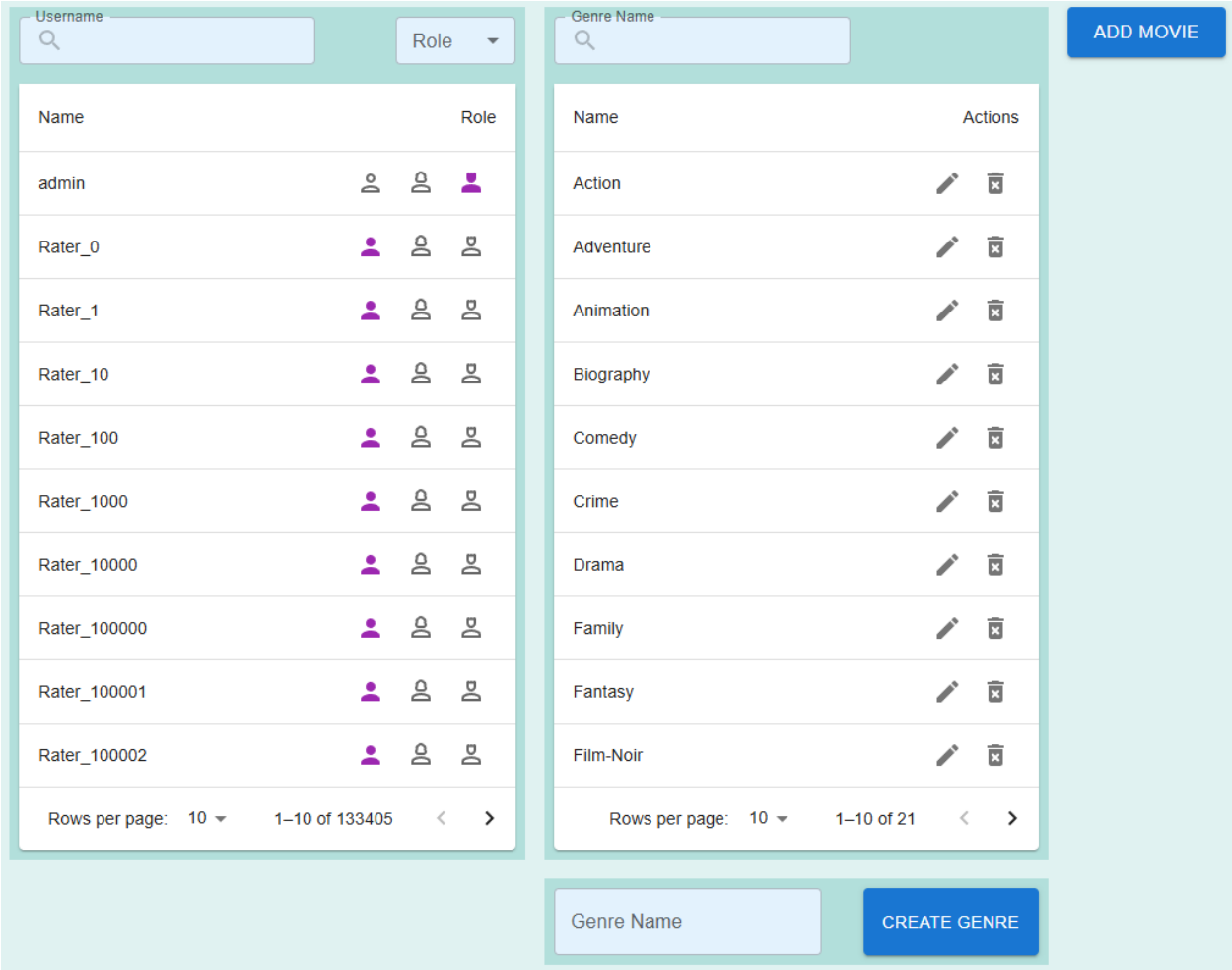

Рисунок 3.20 – Сторінка адміністратора.

Також адміністратор може змінювати роль користувачів, редагувати або видаляти жанри.

Для створення нового фільму потрібно перейти на відповідну сторінку та заповнити форму (рис. 3.21):

- Завантажити постер фільму.
- Вказати назву фільму.
- Вказати опис фільму.
- Вказати жанри фільму.
- Обрати дату випуску фільму.

Після чого натиснути відповідно кнопку. Система збереже вказані дані та перенаправить користувача на сторінку фільму.

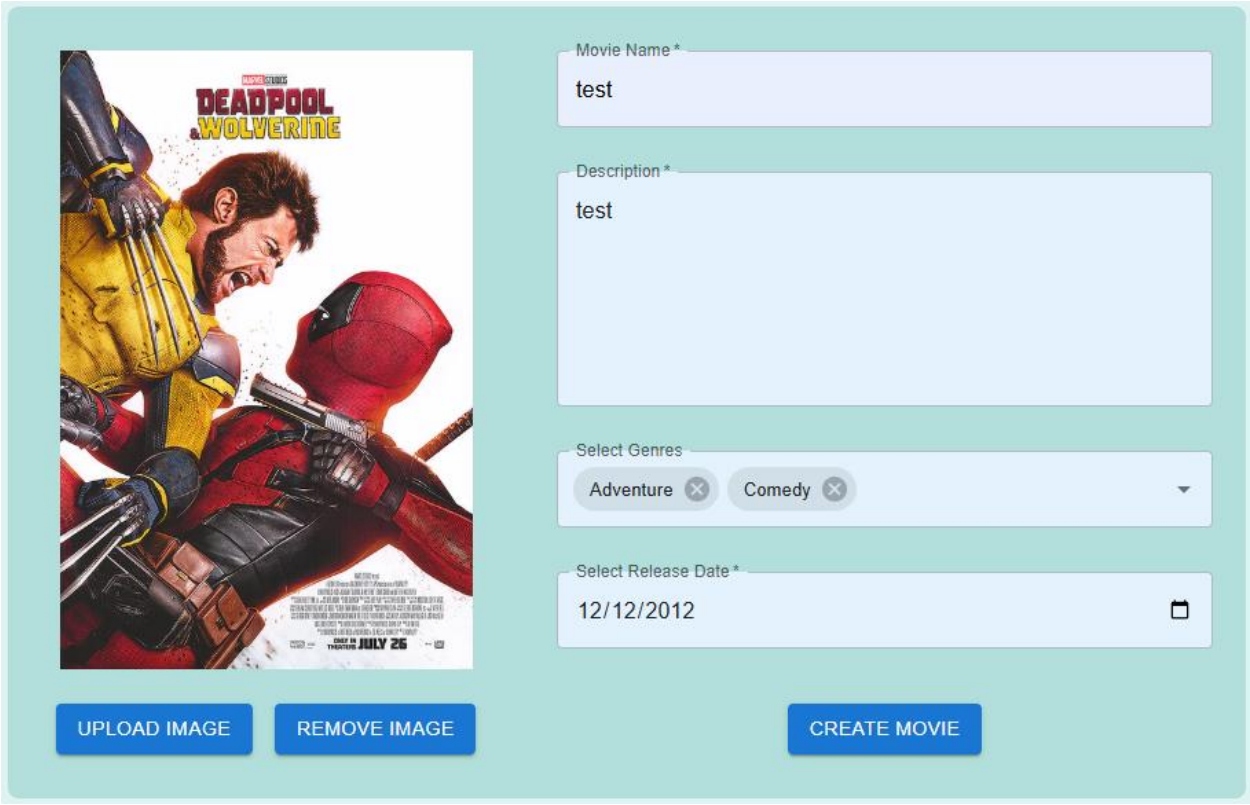

Рисунок 3.21 – Форма створення фільму.

Посилання на GitHub репозиторій: https://github.com/mmvch/reviews

3.6. Тестування системи

Тестування системи є важливим етапом у забезпеченні її якості, оскільки дозволяє виявити приховані помилки та недоліки, які могли залишитися непоміченими під час розробки. Воно спрямоване на перевірку роботи системи в умовах, максимально наближених до реальних, що дає змогу оцінити її стабільність і продуктивність. Завдяки тестуванню можна мінімізувати ризики збоїв, які потенційно можуть негативно вплинути на користувацький досвід.

Крім того, тестування забезпечує впевненість у готовності системи до запуску, її здатності масштабуватися відповідно до зростання навантаження та стабільності під час впровадження майбутніх оновлень. Це критично важливо

для підтримання високого рівня якості продукту. Для тестування системи було створено відповідні набори даних (табл. 3.10).

Таблиця 3.10 – Тестові набори даних.

Назва об'єктів	**Кількість**
Звичайні користувачі	133 265
Рецензенти	133
Жанри фільмів	21
Фільми	250
Фільми-Жанри	624
Рецензії	2005
Оцінки фільмів	2 250 458
Реакції на рецензії	4 563 189

Було протестовано всі сценарії використання, а саме:

- Реєстрація, Авторизація
- Отримання фільму
- Отримання списку фільмів
- Отримання рекомендацій про фільми
- Пошук фільму по певним критеріям
- Створення, Редагування, Видалення фільму
- Оцінювання фільму
- Створення, Редагування, Видалення жанру фільму
- Отримання списку жанрів фільмів
- Отримання рецензії на фільм
- Отримання списку рецензій на фільм
- Оцінювання рецензії на фільм
- Створення, Редагування, Видалення рецензії на фільм
- Зміна ролі користувачів.

Основним завданням тестування було вирахувати швидкодію рекомендаційного алгоритму (сценарій використання – отримання рекомендацій про фільми). Результати тестування наведені на рисунках 3.22 – 3.23.

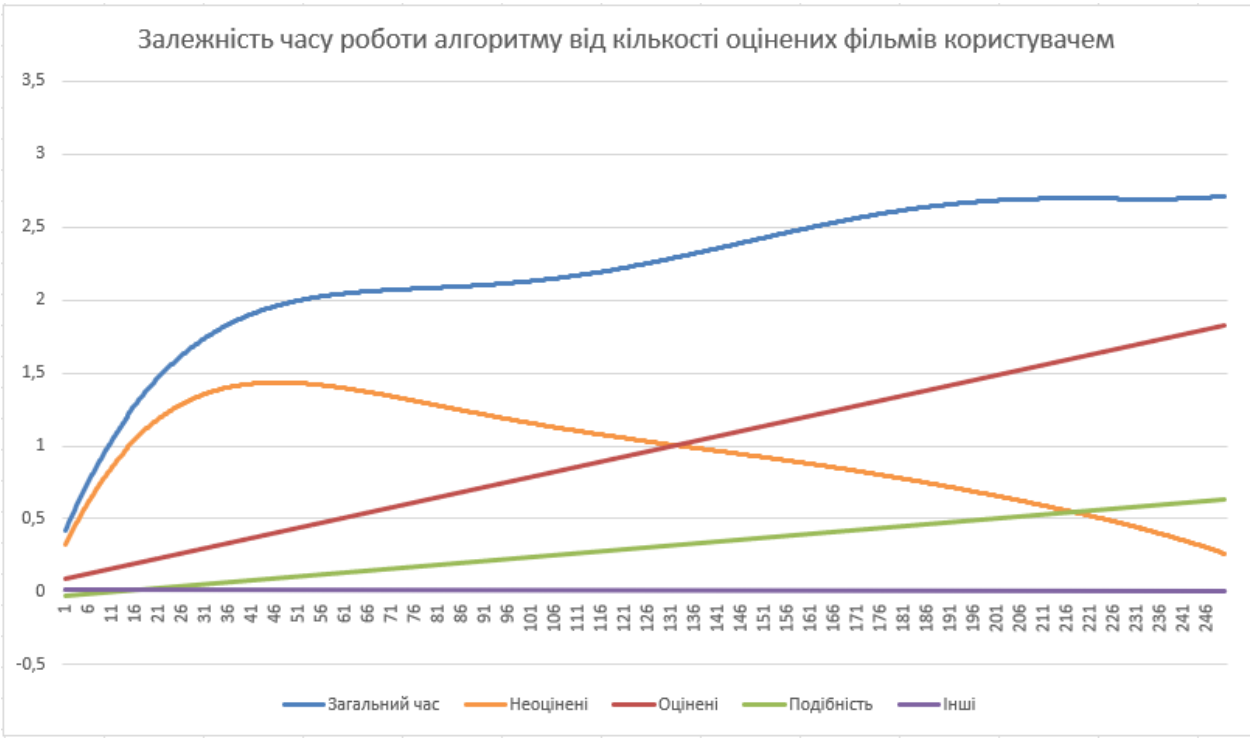

Рисунок 3.22 – Графік залежності часу роботи алгоритму від кількості оцінених фільмів користувачем.

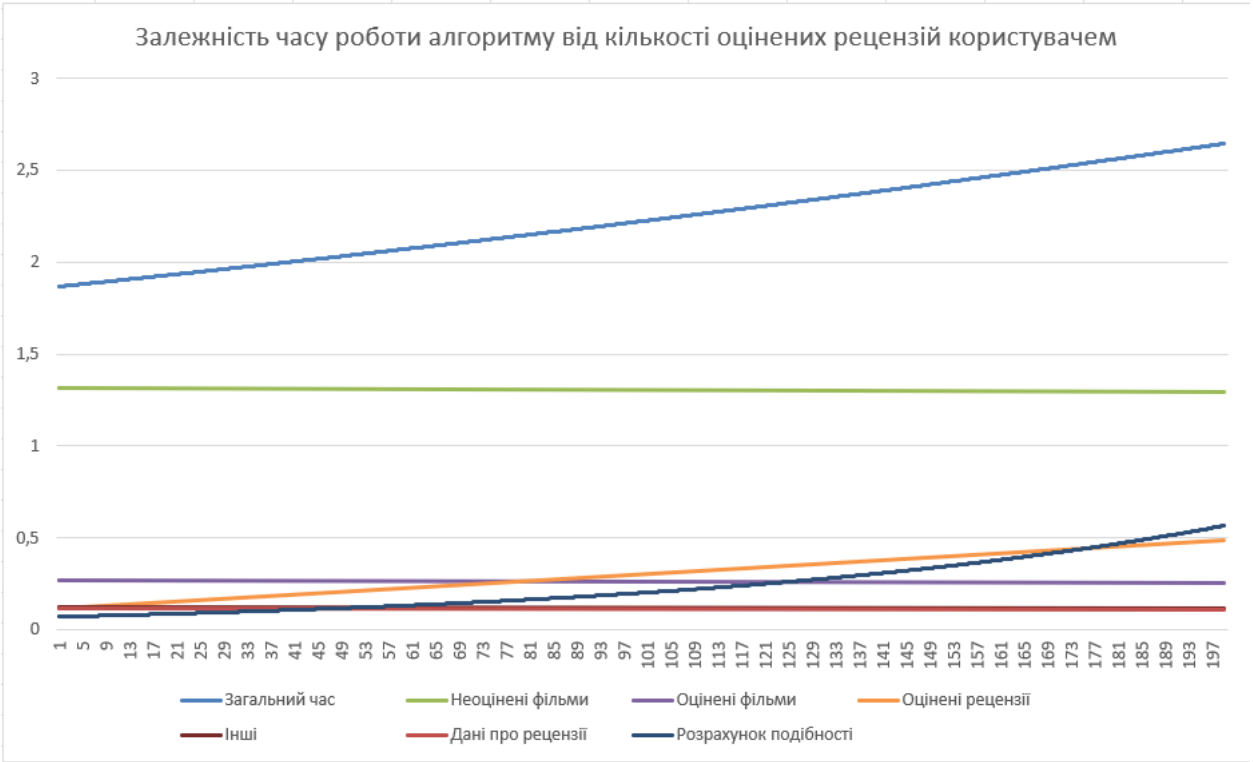

Рисунок 3.23 – Графік залежності часу роботи алгоритму від кількості оцінених рецензій користувачем.

ВИСНОВКИ

У результаті виконання роботи було розроблено веборієнтовану пошуково-рекомендаційну інформаційну технологію рецензування фільмів та інформаційну технологію створення рекомендацій на основі даних про користувачів та даних про об'єкти системи (фільми, жанри, рецензії), які юзери оцінюють або з якими взаємодіють.

Спочатку було проведено інформаційних огляд підходів до створення рекомендаційних систем та вебзастосунків в цілому. Проаналізовано існуючі рішення. Звідси маємо, що є необхідність створення платформи, яка об'єднуватиме найкращі риси кожного з існуючих рішень: надійну базу даних, агреговані рейтинги, а також гнучку і персоналізовану систему рекомендацій. Це дозволить задовольнити потреби різних категорій користувачів і створити конкурентоспроможний продукт.

Для реалізації поставленої задачі було проведено дослідження, на основі якого обрано такі технології та архітектурні рішення:

- Гібридний підхід – для побудови рекомендаційних систем.
- Clean Architecture – для побудови серверної частини.
- ASP.NET Core – для реалізації серверної частини.
- EF Core, MS SQL Server – для реалізації системи зберігання даних.
- React, MUI – для реалізації клієнтської частини.

Відповідно до обраних технологій та архітектурних рішень було:

- За допомогою Use Case Diagram змодельовано систему в цілому. Виділено всіх акторів та функціонал застосунку.
- Спроєктовано відповідно до Clean Architecture бекенд.
- Спроєктовано відповідно до найкращих підходів створення React застосунків фронтенд.
- Реалізовано за допомогою C# рекомендаційний алгоритм.
- Реалізовано за допомогою ASP.NET Core серверну частину.

- Реалізовано за допомогою EF Core, MS SQL Server систему зберігання даних.
- Реалізовано за допомогою React та MUI клієнтську частину.
- Проведено тестування системи, причому ключову роль було відведено тестуванню рекомендаційного алгоритму.

Розроблена пошуково-рекомендаційна система значно спрощує процес персоналізованого пошуку фільмів, забезпечуючи користувачів ефективними рекомендаціями. Дана технологія побудована на універсальних принципах, що дозволяють легко адаптувати її до змін потреб користувачів і в подальшому впроваджувати нові функціональні можливості.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Aggarwal C. C. Recommender Systems: The Textbook. Springer, 2021. 530 p.

2. Ricci F., Rokach L., Shapira B. Recommender Systems Handbook. Springer, 2022. 1050 p.

3. Giroux M. ASP.NET 8 MVC Fundamentals: Step-by-Step Instructions for Building Web Applications with ASP.NET 8, Integrating SQLite for Robust Data Handling, and Utilizing Entity Framework. Kindle Edition, 2024. 229 p.

4. Fernando C. Solution Architecture Patterns for Enterprise: A Guide to Building Enterprise Software Systems. Kindle Edition, 2022. 498 p.

5. Percival B., Gregory H. Architecture Patterns with Python: Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices. O'Reilly Media, 2020. 301 p.

6. Price J. C# 10 and .NET 6 – Modern Cross-Platform Development: Build Apps, Websites, and Services with ASP.NET Core 6, EF Core 6, Blazor, and More. Packt Publishing, 2021. 820 p.

7. Williams T. Microservices Design Patterns in .NET: Making sense of microservices design and architecture using .NET Core. Packt Publishing, 2023. 300 p.

8. Baptista G., Francesco A. Software Architecture with C# 12 and .NET 8 - Fourth Edition: Build enterprise applications using microservices, DevOps, EF Core, and design patterns for Azure. Packt Publishing, 2024. 756 p.

9. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2019. 533 p.

10. Esposito D. Clean Architecture with .NET (Developer Reference). Microsoft Press, 2024. 336 p.

11. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Addison-Wesley, 2021. 432 p.

12. Sanctis V. ASP.NET Core 8 and Angular. Packt Publishing, 2024. 804 p.

13. ASP.NET Core: Microsoft. ASP.NET Core Documentation. URL: https://docs.microsoft.com/aspnet/core (date of access: 12.09.2024).

14. Danylko R. J. ASP.NET 8 Best Practices. Packt Publishing, 2023. 408 p.

15. Entity Framework Core: Microsoft. Entity Framework Core Documentation. URL: https://docs.microsoft.com/ef/core (date of access: 21.09.2024).

16. MS SQL Server: Microsoft. MS SQL Server Documentation. URL: https://learn.microsoft.com/sql/sql-server (date of access: 16.09.2024).

17. React: React Documentation. URL: https://react.dev/learn (date of access: 09.10.2024).

18. Angular: Angular Documentation. URL: https://angular.dev/overview (date of access: 09.10.2024).

19. Vue: Vue Documentation. URL: https://vuejs.org/guide/introduction (date of access: 09.10.2024).

20. Material UI: Material UI Documentation. URL: https://mui.com/material-ui (date of access: 10.10.2024).

21. Troelsen A., Japikse P. Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming. Apress, 2021. 1400 p.

22. Freeman A. Pro ASP.NET Core 7: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages. Apress, 2023. 1032 p.

23. Boslaugh S. Statistics in a Nutshell, 2nd Edition. O'Reilly Media, 2012. 594 p.

24. Axios: Axios Documentation. URL: https://axios-http.com/docs/intro (date of access: 18.10.2024).

ДОДАТОК А

```
public class RecommendationService : IRecommendationService
{
	private const int PositivePoint = 4;
	private const int AveragePoint = 3;

	private const double PopularFilmMinScale = 1;
	private const double PopularFilmGap = 0.2;

	private const double PopularGenreMinScale = 1.5;
	private const double PopularGenreGap = 0.5;

	private const double RatingSimilarityImpact = 0.7;
	private const double ReactionSimilarityImpact = 0.3;

	private readonly IMovieRepository _movieRepository;
	private readonly IGenreRepository _genreRepository;
	private readonly IRatingRepository _ratingRepository;
	private readonly IReviewRepository _reviewRepository;
	private readonly IReactionRepository _reactionRepository;

	public RecommendationService(IMovieRepository movieRepository,
IGenreRepository genreRepository,
		IRatingRepository ratingRepository, IReviewRepository reviewRepository,
IReactionRepository reactionRepository)
	{
		_movieRepository = movieRepository;
		_genreRepository = genreRepository;
		_ratingRepository = ratingRepository;
		_reviewRepository = reviewRepository;
		_reactionRepository = reactionRepository;
	}

	public async Task<PartialData<(Movie, int)>> GetRecommendationsAsync(Guid
targetUserId, MovieFilter filter)
	{
		Expression<Func<Movie, bool>>? movieRestriction =
filter.GetPredicate();
		ConcurrentDictionary<Guid, double> movieScores = [];

		await AdjustScoresByRatingsAsync(movieScores, targetUserId,
movieRestriction);
		await AdjustScoresByReviewsAsync(movieScores, movieRestriction);
		await AdjustScoresByGenresAsync(movieScores, targetUserId,
movieRestriction);

		return await GenerateResultAsync(movieScores, filter);
	}

	private async Task<PartialData<(Movie Movie, int Score)>>
GenerateResultAsync(ConcurrentDictionary<Guid, double> movieScores, MovieFilter
filter)
	{
		static int ScaleToRange(double oldValue, double oldMin, double oldMax,
double newMin, double newMax)
		{
			if (oldMin == oldMax)
			{
				return (int)newMax;
			}
```

```
                  return (int)(newMin + (oldValue - oldMin) * (newMax - newMin) /
(oldMax - oldMin));
            }

            int totalAmount = movieScores.Count;

            if (totalAmount == 0)
            {
                  return new PartialData<(Movie, int)>
                  {
                        TotalAmount = 0
                  };
            }

            double maxValue = movieScores.Values.Max();
            double minValue = movieScores.Values.Min();

            var partialMovieScores = movieScores
                  .OrderByDescending(ms => ms.Value)
                  .Skip(filter.CurrentPage * filter.PageSize ?? 0)
                  .Take(filter.PageSize ?? totalAmount)
                  .ToDictionary(ms => ms.Key, ms => ScaleToRange(ms.Value,
minValue, maxValue, 1, 100));

            Guid[] movieIds = partialMovieScores
                  .Select(ms => ms.Key)
                  .ToArray();

            Movie[] movies = await _movieRepository.GetFilteredListAsync(movieIds);

            PartialData<(Movie, int)> result = new()
            {
                  Data = movieIds.Join(movies, id => id, movie => movie.Id, (id,
movie) => (movie, partialMovieScores[id])),
                  TotalAmount = totalAmount
            };

            return result;
      }

      private async Task AdjustScoresByGenresAsync(ConcurrentDictionary<Guid,
double> movieScores,
            Guid targetUserId, Expression<Func<Movie, bool>>? movieRestriction)
      {
            var movies = await
_movieRepository.GetSimpledListAsync(movieRestriction);
            var favoriteGenres = await
_genreRepository.GetFavoriteGenreIdsAsync(targetUserId, PositivePoint);

            int maxCount = favoriteGenres.Count > 0 ? favoriteGenres.MaxBy(fg =>
fg.Value).Value : 0;

            if (maxCount == 0)
            {
                  return;
            }

            var movieGenres = movies
                  .Select(movie => new SimpleMovie
                  {
                        Id = movie.Id,
                        GenreIds = favoriteGenres.Select(g =>
g.Key).Intersect(movie.GenreIds).ToArray()
                  })
                  .Where(result => result.GenreIds.Length > 0)
```

```
                .ToArray();

            Parallel.ForEach(movieGenres, movieGenre =>
            {
                if (movieScores.TryGetValue(movieGenre.Id, out double
currentValue))
                {
                    foreach (var GenreId in movieGenre.GenreIds)
                    {
                        double scale = PopularGenreMinScale +
favoriteGenres[GenreId] / (double)maxCount * PopularGenreGap;
                        double newValue = currentValue >= 0 ? currentValue
* scale : currentValue / scale;
                        movieScores.TryUpdate(movieGenre.Id, newValue,
currentValue);
                    }
                }
            });
        }

        private async Task AdjustScoresByReviewsAsync(ConcurrentDictionary<Guid,
double> movieScores,
            Expression<Func<Movie, bool>>? movieRestriction)
        {
            MoviePopularity[] movies = await
GetMoviesPopularityAsync(movieRestriction);

            int[] counts = movies.Select(m => Math.Abs(m.Count)).ToArray();
            int maxCount = counts.Length > 0 ? counts.Max() : 0;

            if (maxCount == 0)
            {
                return;
            }

            Parallel.ForEach(movies, movie =>
            {
                if (movieScores.TryGetValue(movie.MovieId, out double
currentValue))
                {
                    double scale = PopularFilmMinScale + movie.Count /
(double)maxCount * PopularFilmGap;
                    double newValue = currentValue >= 0 ? currentValue * scale
: currentValue / scale;
                    movieScores.TryUpdate(movie.MovieId, newValue,
currentValue);
                }
            });
        }

        private async Task<MoviePopularity[]>
GetMoviesPopularityAsync(Expression<Func<Movie, bool>>? movieRestriction)
        {
            var reviewStacks = await
_reviewRepository.GetSimpledStacksAsync(movieRestriction);

            var result = reviewStacks
                .Select(g => new MoviePopularity
                {
                    MovieId = g.Key,
                    Count = g.Value.Count(a => a.Point > AveragePoint &&
a.Likes > a.Dislikes ||
                                                                    a.Point <
AveragePoint && a.Likes < a.Dislikes) -
```

```
                                        g.Value.Count(a => a.Point > AveragePoint &&
a.Likes < a.Dislikes ||
                                                                        a.Point <
AveragePoint && a.Likes > a.Dislikes)
                    })
                    .Where(a => a.Count != 0)
                    .ToArray();

             return result;
      }

      private async Task AdjustScoresByRatingsAsync(ConcurrentDictionary<Guid,
double> movieScores,
             Guid targetUserId, Expression<Func<Movie, bool>>? movieRestriction)
      {
             Dictionary<Guid, double> similarities = await
GetSimilarUsersAsync(targetUserId);
             Dictionary<Guid, SimpleRating[]> ratings = await
_ratingRepository.GetUnrelatedRatingsAsync(targetUserId, movieRestriction);

             Parallel.ForEach(similarities, similarity =>
             {
                    if (ratings.TryGetValue(similarity.Key, out SimpleRating[]?
subRatings))
                    {
                           foreach (var rating in subRatings)
                           {
                                  movieScores.AddOrUpdate(rating.MovieId,
                                  rating.Point * similarity.Value,
                                  (key, value) => value + rating.Point *
similarity.Value);
                           }
                    }
             });
      }

      private async Task<Dictionary<Guid, double>> GetSimilarUsersAsync(Guid
targetUserId)
      {
             ConcurrentDictionary<Guid, double> similarityScores = [];

             var ratings = await
_ratingRepository.GetRelatedRatingsAsync(targetUserId);
             var targetSubRatings = await
_ratingRepository.GetForUserAsync(targetUserId);

             Parallel.ForEach(ratings, subRatings =>
             {
                    double similarity = PearsonCorrelation(targetSubRatings,
subRatings.Value);

                    if (similarity != 0)
                    {
                           similarityScores.TryAdd(subRatings.Key, similarity *
RatingSimilarityImpact);
                    }
             });

             var reactions = await
_reactionRepository.GetRelatedReactionsAsync(targetUserId);
             var targetSubReactions = await
_reactionRepository.GetForUserAsync(targetUserId);

             Parallel.ForEach(reactions, subReactions =>
             {
```

```
                    double similarity = PearsonCorrelation(targetSubReactions,
subReactions.Value);

                    if (similarity != 0)
                    {
                          similarityScores.AddOrUpdate(
                                subReactions.Key,
                                similarity * ReactionSimilarityImpact,
                                (key, value) => value + similarity *
ReactionSimilarityImpact
                          );
                    }
             });

             return similarityScores.ToDictionary();
      }

      private static double PearsonCorrelation(SimpleRating[] ratings1,
SimpleRating[] ratings2)
      {
             var ratings = ratings2
                    .Join(ratings1, r2 => r2.MovieId, r1 => r1.MovieId, (r2, r1) =>
new
                    {
                          Point1 = r1.Point,
                          Point2 = r2.Point
                    })
                    .ToArray();

             int n = ratings.Length;

             if (n == 0)
             {
                    return 0;
             };

             int minLength = Math.Min(ratings1.Length, ratings2.Length);
             int maxLength = Math.Max(ratings1.Length, ratings2.Length);

             int sum1 = ratings.Sum(r => r.Point1);
             int sum2 = ratings.Sum(r => r.Point2);

             int sum1Sq = ratings.Sum(r => r.Point1 * r.Point1);
             int sum2Sq = ratings.Sum(r => r.Point2 * r.Point2);

             int pSum = ratings.Sum(r => r.Point1 * r.Point2);

             double num = pSum - sum1 * sum2 / (double)n;
             double den = Math.Sqrt((sum1Sq - Math.Pow(sum1, 2) / n) * (sum2Sq -
Math.Pow(sum2, 2) / n));

             double result = (den == 0)
                    ? minLength / (double)maxLength
                    : (num / den * minLength / maxLength);

             return result;
      }

      private static double PearsonCorrelation(SimpleReaction[] reactions1,
SimpleReaction[] reactions2)
      {
             var ratings = reactions2
                    .Join(reactions1, r2 => r2.ReviewId, r1 => r1.ReviewId, (r2, r1)
=> new
                    {
```

```
                        IsLiked1 = r1.IsLiked ? 1 : 0,
                        IsLiked2 = r2.IsLiked ? 1 : 0
                  })
                  .ToArray();

            int n = ratings.Length;

            if (n == 0)
            {
                  return 0;
            };

            int minLength = Math.Min(reactions1.Length, reactions2.Length);
            int maxLength = Math.Max(reactions1.Length, reactions2.Length);

            int sum1 = ratings.Sum(r => r.IsLiked1);
            int sum2 = ratings.Sum(r => r.IsLiked2);

            int sum1Sq = ratings.Sum(r => r.IsLiked1 * r.IsLiked1);
            int sum2Sq = ratings.Sum(r => r.IsLiked2 * r.IsLiked2);

            int pSum = ratings.Sum(r => r.IsLiked1 * r.IsLiked2);

            double num = pSum - sum1 * sum2 / (double)n;
            double den = Math.Sqrt((sum1Sq - Math.Pow(sum1, 2) / n) * (sum2Sq -
Math.Pow(sum2, 2) / n));

            double result = (den == 0)
                  ? minLength / (double)maxLength
                  : (num / den * minLength / maxLength);

            return result;
      }
}


public class GenreRepository(Context context) : Repository<Genre, Guid>(context),
IGenreRepository
{
      public async Task<Dictionary<Guid, int>> GetFavoriteGenreIdsAsync(Guid
userId, int positivePoint)
      {
            return await _context.Ratings.AsNoTracking()
                  .Where(r => r.UserId == userId && r.Point >= positivePoint)
                  .SelectMany(r => r.Movie!.Genres!)
                  .GroupBy(g => g.Id)
                  .Select(group => new
                  {
                        GenreId = group.Key,
                        Count = group.Count()
                  })
                  .ToDictionaryAsync(g => g.GenreId, g => g.Count);
      }
}


public class ReactionRepository(Context context) : Repository<ReviewReaction,
Guid>(context), IReactionRepository
{
      public async Task<Dictionary<Guid, SimpleReaction[]>>
GetRelatedReactionsAsync(Guid userId)
      {
            var reviewIds = _context.Reviews.AsNoTracking()
                  .Where(r => r.Reactions!.Any(rr => rr.UserId == userId))
                  .Select(r => r.Id);
```

```
            return await _dbSet.AsNoTracking()
                  .Where(rr => reviewIds.Contains(rr.ReviewId))
                  .GroupBy(rr => rr.UserId)
                  .ToDictionaryAsync(
                        g => g.Key,
                        g => g.Select(rr => new SimpleReaction
                        {
                              IsLiked = rr.IsLiked,
                              ReviewId = rr.ReviewId
                        }).ToArray()
                  );
      }

      public async Task<SimpleReaction[]> GetForUserAsync(Guid userId)
      {
            return await _dbSet.AsNoTracking()
                  .Where(r => r.UserId == userId)
                  .Select(r => new SimpleReaction
                  {
                        IsLiked = r.IsLiked,
                        ReviewId = r.ReviewId
                  })
                  .ToArrayAsync();
      }
}


public class RatingRepository(Context context) : Repository<Rating, Guid>(context),
IRatingRepository
{
      public async Task<Dictionary<Guid, SimpleRating[]>>
GetRelatedRatingsAsync(Guid userId)
      {
            var movieIds = _context.Movies.AsNoTracking()
                  .Where(m => m.Ratings!.Any(r => r.UserId == userId))
                  .Select(m => m.Id);

            return await _dbSet.AsNoTracking()
                  .Where(r => movieIds.Contains(r.MovieId))
                  .GroupBy(r => r.UserId)
                  .ToDictionaryAsync(
                        g => g.Key,
                        g => g.Select(r => new SimpleRating
                        {
                              Point = r.Point,
                              MovieId = r.MovieId
                        }).ToArray()
                  );
      }

      public async Task<Dictionary<Guid, SimpleRating[]>>
GetUnrelatedRatingsAsync(Guid userId, Expression<Func<Movie, bool>>? predicate =
null)
      {
            var query = _context.Movies.AsNoTracking();

            if (predicate != null)
            {
                  query = query.Where(predicate);
            }

            var userIds = _context.Movies.AsNoTracking()
                  .Where(m => m.Ratings!.Any(r => r.UserId == userId))
                  .SelectMany(m => m.Ratings!)
```

```
                .Select(r => r.UserId)
                .Distinct();

            var movieIds = query
                .Where(m => m.Ratings!.All(r => r.UserId != userId))
                .Select(m => m.Id);

            var res = await _dbSet.AsNoTracking()
                .Where(r => userIds.Contains(r.UserId) &&
movieIds.Contains(r.MovieId))
                .GroupBy(r => r.UserId)
                .ToDictionaryAsync(
                    g => g.Key,
                    g => g.Select(r => new SimpleRating
                    {
                        Point = r.Point,
                        MovieId = r.MovieId
                    }).ToArray()
                );

            return res;
        }

        public async Task<SimpleRating[]> GetForUserAsync(Guid userId)
        {
            return await _dbSet.AsNoTracking()
                .Where(r => r.UserId == userId)
                .Select(r => new SimpleRating
                {
                    Point = r.Point,
                    MovieId = r.MovieId
                })
                .ToArrayAsync();
        }
}



public class ReviewRepository(Context context) : Repository<Review, Guid>(context),
IReviewRepository
{
        public async Task<Dictionary<Guid, SimpleReview[]>>
GetSimpledStacksAsync(Expression<Func<Movie, bool>>? predicate = null)
        {
            var query = _dbSet.AsNoTracking();

            if (predicate != null)
            {
                var movies = _context.Movies.AsNoTracking().Where(predicate);
                query = query.Join(movies, r => r.MovieId, m => m.Id, (r, m) =>
r);
            }

            return await query
                .GroupJoin(_context.ReviewReactions.AsNoTracking(),
                    r => r.Id,
                    rr => rr.ReviewId,
                    (review, reactions) => new
                    {
                        review.MovieId,
                        review.Point,
                        Likes = reactions.Count(rr => rr.IsLiked),
                        Dislikes = reactions.Count(rr => !rr.IsLiked)
                    })
                .GroupBy(a => a.MovieId)
                .ToDictionaryAsync(
```

```
                        g => g.Key,
                        g => g.Select(r => new SimpleReview
                        {
                              Point = r.Point,
                              Likes = r.Likes,
                              Dislikes = r.Dislikes,
                        }).ToArray()
            );
      }
}


public class MovieRepository(Context context) : Repository<Movie, Guid>(context),
IMovieRepository
{
      public async Task<Movie[]> GetFilteredListAsync(Guid[] movieIds)
      {
            return await _dbSet.AsNoTracking().Where(movie =>
movieIds.Contains(movie.Id)).ToArrayAsync();
      }

      public async Task<SimpleMovie[]> GetSimpledListAsync(Expression<Func<Movie,
bool>>? predicate = null)
      {
            var query = _dbSet.AsNoTracking();

            if (predicate != null)
            {
                  query = query.Where(predicate);
            }

            return await query.Include(m => m.Genres)
                  .Select(m => new SimpleMovie
                  {
                        Id = m.Id,
                        GenreIds = m.Genres != null
                              ? m.Genres.Select(g => g.Id).ToArray()
                              : Array.Empty<Guid>()
                  }).ToArrayAsync();
      }
}
```