МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій і аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: ***«Розробка кросплатформного додатку «Домашній кінотеатр»»***

Виконав: студент 4 курсу, групи КН-41
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Семенюк Валерій Сергійович

Керівник: *викладач кафедри ЕММІТ, Ляховчук С.В.*

Рецензент: *кандидат технічних наук, доцент, доцент кафедри прикладної математики Донецького національного університету імені Василя Стуса Загоруйко Любов Василівна*

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних
_____ (проф., д.е.н. Кривицька О.Р.)
Протокол № 11 від «20» травня 2026 р.

Острог, 2026

АНОТАЦІЯ

кваліфікаційної роботи

на здобуття освітнього ступеня бакалавра

Тема: Розробка кросплатформного додатку «Домашній кінотеатр».

Автор: Семенюк Валерій Сергійович

Науковий керівник: викладач кафедри економіко-математичного моделювання та інформаційних технологій Ляховчук Сергій Васильович.

Захищена «.....»..... 20__ року.

Пояснювальна записка до кваліфікаційної роботи: 73с., 25 рис., 6 табл., 8 додатків, 21 джерел.

Ключові слова: СТРІМІНГОВИЙ СЕРВІС, KOTLIN, KTOR, FLUTTER, POSTGRESQL, EXPOSED, RIVERPOD, REST API, CROSS-PLATFORM, ВІДЕОПОТОКОВЕ ВІДТВОРЕННЯ, BCrypt, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА.

Короткий зміст праці:

Кваліфікаційна робота присвячена проєктуванню та розробці клієнт-серверного застосунку Datflix — онлайн-кінотеатру для перегляду фільмів. Об'єктом дослідження є процеси створення клієнт-серверного застосунку для перегляду фільмів, а предметом — методи та засоби розробки: архітектура REST API, проєктування реляційної бази даних, реалізація бізнес-логіки на стороні сервера, організація взаємодії з клієнтом, управління станом у мобільному додатку.

У роботі проаналізовано український та міжнародний ринок стрімінгових сервісів та інтерфейсні рішення платформ-аналогів Netflix, Megogo, Sweet.tv. На основі аналізу сформульовано функціональні та нефункціональні вимоги до Datflix, побудовано ER-діаграму бази даних, діаграму розгортання системи, діаграму шарів архітектури.

У практичній частині реалізовано серверну частину на Kotlin з використанням Ktor 2.3.7, PostgreSQL 18, Exposed 0.41.1, HikariCP 5.0.1 та BCrypt 0.9.0. Клієнтську частину створено на Flutter 3.x з використанням Riverpod 2.3.2, video_player, chewie, url_launcher, google_fonts та flutter_localizations. Застосунок містить систему автентифікації, каталог фільмів із фільтрацією за жанром, типом, роком випуску, детальні сторінки фільмів та акторів, списки обраного та історії переглядів, тематичні підбірки, систему коментарів та оцінок (10-бальна

шкала), відеоплеєр із підтримкою повноекранного режиму, локалізацію (українська/англійська) та дві кольорові теми (фіолетова/синя).

Результати тестування підтверджують працездатність усіх основних сценаріїв. Виконано міграцію даних з Firestore до PostgreSQL. Усунено вісім характерних помилок: проблему кодування кирилиці, помилку 405 Method Not Allowed, дублювання даних між користувачами, помилку серіалізації LocalDateTime, overflow в інтерфейсі, дублювання ключа при створенні коментарів, лаги через вбудований YouTubePlayer, помилку "Invalid constant value". Оптимізовано продуктивність шляхом заміни вбудованого YouTubePlayer на відкриття трейлерів у браузері. Результати роботи можуть бути використані як основа для подальшого розвитку навчально-практичного прототипу стрімінгового сервісу.

_____

(підпис автора)

ANNOTATION

of the qualification work

for obtaining the educational degree of Bachelor

Topic: Development of Cross-Platform Application "Home Cinema".

Author: Valerii Serhiyovych Semeniuk

Scientific Supervisor: Lecturer of the Department of Economic-Mathematical Modeling and Information Technology, Serhii Vasyliovych Liakhovchuk.

Defended on "....." ..... 20__.

Explanatory note to the qualification work: 73 p., 25 fig., 6 tab., 8 appendices, 21 sources.

Keywords: STREAMING SERVICE, KOTLIN, KTOR, FLUTTER, POSTGRESQL, EXPOSED, RIVERPOD, REST API, CROSS-PLATFORM, VIDEO PLAYBACK, BCrypt, CLIENT-SERVER ARCHITECTURE.

Brief content of the work:

The qualification work is devoted to the design and development of the Datflix client-server application — an online cinema for watching movies. The object of the research is the processes of creating a client-server application for watching movies, while the subject is the methods and tools of development: REST API architecture, relational

database design, implementation of business logic on the server side, organization of client interaction, state management in mobile applications.

The work analyzes the Ukrainian and international streaming service market and the interface solutions of comparable platforms: Netflix, Megogo, Sweet.tv. Based on this analysis, functional and non-functional requirements for Datflix are formulated. An ER diagram of the database, system deployment diagram, and layered architecture diagram are developed.

The practical part implements the server-side using Kotlin with Ktor 2.3.7, PostgreSQL 18, Exposed 0.41.1, HikariCP 5.0.1, and BCrypt 0.9.0. The client-side is created on Flutter 3.x using Riverpod 2.3.2, video_player, chewie, url_launcher, google_fonts, and flutter_localizations. The application includes an authentication system, movie catalog with filtering by genre, type, release year, detailed movie and actor pages, favorites and watch history lists, thematic collections, comment and rating system (10-point scale), video player with full-screen mode support, localization (Ukrainian/English), and two color themes (purple/blue).

The testing results confirm the operability of all main scenarios. Data migration from Firestore to PostgreSQL was completed. Eight characteristic errors were fixed: Cyrillic encoding problem, 405 Method Not Allowed error, data duplication between users, LocalDateTime serialization error, interface overflow, key duplication when creating comments, lag due to embedded YouTubePlayer, "Invalid constant value" error. Performance was optimized by replacing the embedded YouTubePlayer with opening trailers in the browser. The results of the work can be used as a basis for further development of an educational and practical streaming service prototype.

_____

(author's signature

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Термін	Визначення
API	Application Programming Interface – інтерфейс програмування застосунків
BCrypt	Blowfish Crypt – алгоритм хешування паролів
CORS	Cross-Origin Resource Sharing – спільне використання ресурсів між різними джерелами
CRUD	Create, Read, Update, Delete – базові операції з даними
DSL	Domain-Specific Language – предметно-орієнтована мова програмування
ER	Entity-Relationship – сутність-зв'язок (тип діаграми)
HTTP	HyperText Transfer Protocol – протокол передачі гіпертексту
ISO	International Organization for Standardization – Міжнародна організація зі стандартизації
JDBC	Java Database Connectivity – інтерфейс підключення до баз даних
JSON	JavaScript Object Notation – текстовий формат обміну даними
JVM	Java Virtual Machine – віртуальна машина Java
ORM	Object-Relational Mapping – об'єктно-реляційне відображення
REST	Representational State Transfer – архітектурний стиль побудови розподілених систем
UI	User Interface – інтерфейс користувача
UUID	Universally Unique Identifier – універсальний унікальний ідентифікатор
UX	User Experience – досвід користувача
WCAG	Web Content Accessibility Guidelines – настанови з доступності вебконтенту

ВСТУП

Сучасний ринок відеостримінгу демонструє стрімке зростання. За даними аналітичної платформи Statista, у 2024 році його обсяг перевищив 100 млрд доларів США [16]. Згідно з прогнозами Grand View Research, очікується щорічне зростання на рівні 15% [17]. Це підтверджує актуальність розробки власних клієнт-серверних рішень, подібних до представленого в цій роботі.

Сучасні споживачі очікують не просто можливості перегляду відеоконтенту, а повноцінного персоналізованого досвіду: збереження історії, формування списків улюблених фільмів, отримання рекомендацій, можливості оцінювати та коментувати контент. Створення такого застосунку є комплексним завданням, що охоплює як розробку серверної частини з базою даних, так і побудову зручного клієнтського інтерфейсу.

Актуальність роботи зумовлена необхідністю отримання практичних навичок розробки сучасних клієнт-серверних застосунків із використанням актуальних технологій: мови Kotlin, фреймворку Ktor, СУБД PostgreSQL та кросплатформного інструментарію Flutter. Особливу увагу приділено питанням безпеки (власна автентифікація замість готових рішень) та оптимізації роботи з даними.

Об'єктом дослідження є процеси створення клієнт-серверного застосунку для перегляду фільмів, включаючи етапи проєктування, реалізації, тестування та налагодження.

Предметом дослідження виступають методи та засоби розробки: архітектура REST API, проєктування реляційної бази даних, реалізація бізнес-логіки на стороні сервера, організація взаємодії з клієнтом, управління станом у мобільному додатку

Метою роботи є проєктування та програмна реалізація клієнт-серверного застосунку "Datflix", який забезпечує функціонал каталогізації фільмів, їх фільтрації, створення персоналізованих списків, перегляду інформації про акторів, формування тематичних підбірок, а також систему коментарів та оцінок. У межах роботи програмний продукт отримав назву Datflix.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Провести аналіз предметної області та сформулювати функціональні вимоги до застосунку.
2. Спроєктувати архітектуру системи, розробити структуру бази даних та специфікацію API.
3. Реалізувати серверну частину на мові Kotlin з використанням фреймворку Ktor та ORM Exposed.
4. Розробити клієнтську частину на Flutter, забезпечивши керування станом за допомогою Riverpod.
5. Інтегрувати систему автентифікації на основі хешування паролів BCrypt.
6. Реалізувати функціонал "Обрані", "Історія переглядів", розширені фільтри, модуль акторів, підбірки та коментарі.
7. Провести тестування розробленого застосунку, виявити та усунути помилки.

У ході виконання кваліфікаційної роботи використовувалися системи контролю версій (Git), інструменти для роботи з API (Postman), середовища розробки (Android Studio), системи керування базами даних (pgAdmin). При проєктуванні системи використовувалися підходи, описані в працях М. Фаулера [21] та Р. Мартіна [20], а також патерни проєктування з канонічної книги «Банди чотирьох» [19].

РОЗДІЛ 1 ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1. Аналіз предметної області та постановка завдання

1.1.1. Опис предметної області

Предметна область охоплює функціонування онлайн-кінотеатрів або стрімінгових сервісів, які надають користувачам доступ до каталогу фільмів. На відміну від традиційних форм поширення відеоконтенту, онлайн-платформи забезпечують миттєвий доступ до великих бібліотек контенту без необхідності фізичного носія, що значно підвищує зручність використання. Крім того, такі системи надають можливість персоналізації: кожен користувач може формувати власний список переглянутого та бажаного до перегляду контенту.

Основними учасниками розробленої системи є:

- Користувачі - переглядають контент, формують власні списки, залишають відгуки.
- Система - забезпечує зберігання інформації про фільми, акторів, жанри, користувачів, обробку запитів, автентифікацію

Ключові бізнес-процеси системи включають:

- реєстрація та вхід користувача;
- перегляд каталогу фільмів з можливістю фільтрації;
- отримання детальної інформації про фільм (опис, трейлер, актори);
- додавання фільмів до списку "Обрані";
- автоматичне збереження історії переглядів;
- перегляд інформації про акторів та їх фільмографію;
- перегляд тематичних підбірок;
- додавання коментарів та оцінок.

1.1.2. Аналіз існуючих рішень

На ринку представлено багато стрімінгових сервісів, як-от Netflix, Megogo, Sweet.tv. Проведений аналіз цих платформ дозволив виявити їхні переваги, недоліки та визначити напрями для розробки власного рішення. Нижче наведено порівняльну таблицю.

Таблиця 1.1

Порівняльна таблиця Netflix/Megogo/Sweet.tv/Datflix

Критерій	**Netflix**	**MEGOGO**	**SWEET.TV**	**Datflix**
Персоналізовані колекції	+	+	±	+
Власна дизайн-система	+	-	-	+
Кастомізація теми	-	-	-	+
Мобільний застосунок	+	+	+	+
Відекрите API для сторонніх розробників	-	-	-	+
Відкрита архітектура	-	-	-	+

Спільними недоліками існуючих рішень є відсутність відкритого API для сторонніх розробників, неможливість розширення функціоналу під конкретні потреби, а також закритість даних. Це обґрунтовує доцільність створення власного застосунку з відкритою архітектурою, яка дозволяє гнучко налаштовувати та розширювати функціонал.

1.1.3. Постановка задачі та функціональні вимоги

На основі аналізу предметної галузі та існуючих рішень сформульовано наступні функціональні вимоги до системи:

Автентифікація: реєстрація нового користувача (email, пароль, ім'я), вхід зареєстрованого користувача, вихід з системи.

Робота з фільмами: отримання списку всіх фільмів, фільтрація за жанром, типом контенту (фільм, серіал, аніме, мультфільм), роком випуску, пошук за назвою, отримання детальної інформації про фільм (опис, трейлер, відео, актори).

Персоналізація: додавання/видалення фільмів зі списку "Обрані", перегляд списку обраних фільмів, автоматичне додавання переглянутих фільмів до історії, перегляд історії переглядів.

Актори: перелік усіх акторів з можливістю пошуку за іменем, сторінка актора з біографією, фотографією, списком фільмів.

Підбірки: перегляд списку тематичних підбірок, перегляд фільмів, що входять до підбірки.

Коментарі та оцінки: отримання списку коментарів до фільму, додавання коментаря (текст + оцінка від 1 до 10),відображення оцінки у вигляді зірок.

Окрім функціональних, визначено такі нефункціональні вимоги:

- Безпека: паролі користувачів мають зберігатися виключно у хешованому вигляді з використанням алгоритму BCrypt (cost factor не менше 10). Час генерації хешу не повинен перевищувати 300 мс на сучасному обладнанні.

- Продуктивність серверної частини: час відповіді API для 95% запитів (отримання списку фільмів, детальна інформація) у локальному середовищі не повинен перевищувати 200 мс під навантаженням до 100 умовних користувачів.

- Розмір відповіді API: для запиту списку фільмів (до 50 записів) обсяг переданих даних має бути не більше 2 МБ у форматі JSON.

- Розмежування даних: система повинна забезпечувати ізоляцію персональних списків (обране, історія) між різними користувачами на основі унікального ідентифікатора (X-User-Id).

- Адаптивність інтерфейсу: клієнтський застосунок має коректно відображатися на екранах з роздільною здатністю від 720×1080 до 1440×2560, підтримувати портретну та ландшафтну орієнтацію.

- Локалізація: підтримка двох мов (українська, англійська) з можливістю зміни мови без перезапуску застосунку.

- Час хешування пароля при реєстрації: не більше 300 мс.

- Пікове споживання оперативної пам'яті клієнтським застосунком: не більше 200 МБ на пристроях з Android 6.0+ / iOS 13+.

1.1.4. Діаграма варіантів використання

Для візуалізації функціональних вимог та ідентифікації акторів, які взаємодіють із системою, розроблено діаграму варіантів використання (Use Case Diagram), наведену на рис. 1.1. Вона відображає дві категорії акторів: «Незареєстрований користувач» та «Зареєстрований користувач», а також основні варіанти використання, доступні кожному з них.

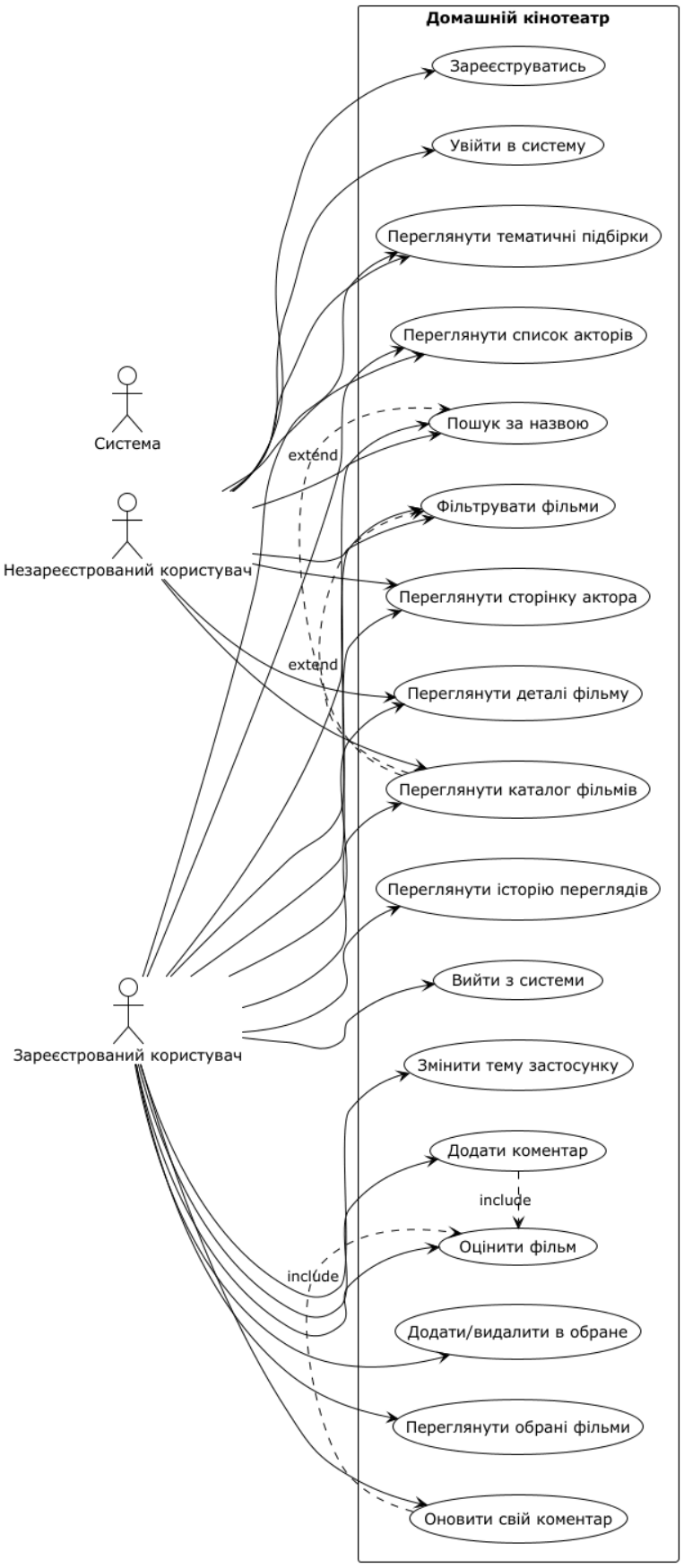

Рис. 1.1. Діаграма варіантів використання системи «Домашній кінотеатр»

Незареєстрований користувач має доступ до наступних функцій: реєстрація, вхід, перегляд каталогу фільмів, фільтрація та пошук за назвою, перегляд детальної інформації про фільм (опис, трейлер, актори), перегляд списку акторів та сторінки актора, а також перегляд тематичних підбірок.

Зареєстрований користувач додатково отримує можливість: додавати/видаляти фільми до списку «Обрані», переглядати обрані фільми та історію переглядів, додавати коментарі, оновлювати власні коментарі, оцінювати фільми (10-бальна шкала), змінювати тему застосунку (фіолетова/синя) та виходити з системи.

Зв’язок «include» між варіантами «Додати коментар» та «Оцінити фільм» означає, що додавання коментаря завжди супроводжується виставленням оцінки. Відношення «extend» між переглядом каталогу та фільтрацією/пошуком вказує, що ці дії є необов’язковими розширеннями основного сценарію.

1.1.5. Вибір технологічного стеку

Вибір технологічного стеку є одним з ключових рішень при проектуванні системи, оскільки він безпосередньо впливає на продуктивність, масштабованість та зручність розробки. Після аналізу доступних інструментів для реалізації обрано такі технології:

- Серверна частина: Kotlin 1.9.22 як основна мова програмування завдяки лаконічному синтаксису та повній сумісності з JVM, Ktor 2.3.7 як асинхронний фреймворк для побудови REST API, PostgreSQL 18 як надійна реляційна СУБД, Exposed 0.41.1 як ORM-бібліотека для типобезпечних SQL-запитів, HikariCP 5.0.1 для пулу підключень до бази даних, BCrypt 0.9.0 для безпечного хешування паролів. [1, 2, 3, 4, 5, 6]
- Клієнтська частина: Flutter 3.x як кросплатформний фреймворк, Riverpod 2.3.2 для управління станом, http 0.13.6 для мережевих запитів, shared_preferences 2.2.0 для локального збереження даних,video_player 2.5.1 та chewie 1.5.0 для

відтворення відео, url_launcher 6.2.6 для відкриття посилань,google_fonts 4.0.3 та flutter_localizations для локалізації. [7, 8, 9, 10, 11, 12, 13]

- Інструменти розробки: IntelliJ IDEA, Android Studio, Postman, pgAdmin, Git. [14, 15]

Вибір обґрунтовано необхідністю створення масштабованої, продуктивної та безпечної системи з сучасним інтерфейсом, яка при цьому залишається зручною у розробці та подальшій підтримці.

1.1.6. Проєктування дизайн-системи

Для забезпечення візуальної цілісності застосунку було розроблено власну дизайн-систему на основі темної теми з підтримкою кастомізації акцентного кольору. Особливу увагу приділено читабельності інтерфейсу, контрастності елементів та адаптації під мобільні пристрої.
Анімований фон (PetalBackground) : 30-40 частинок-пелюсток, які плавно рухаються за допомогою CustomPaint, створюючи ефект «живого» інтерфейсу без значного навантаження на CPU.

Нижня навігаційна панель (BottomNavBar) : 7 пунктів (Головна, Фільми, Обране, Історія, Колекції, Актори, Налаштування). Панель має градієнтний фон та заокруглені кути (30px зверху).

Картка фільму (MovieCard) : містить постер, назву та кнопку «обране» у вигляді іконки. При натисканні на картку відкривається детальна сторінка фільму.

РОЗДІЛ 2 ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Архітектура клієнт-серверної взаємодії

Система побудована за класичною трирівневою архітектурою: клієнт (Presentation Tier), сервер застосунків (Application Tier) та база даних (Data Tier). Такий підхід є загальноприйнятим стандартом при розробці сучасних веб- та мобільних застосунків, оскільки забезпечує чіткий розподіл відповідальностей між компонентами системи. Загальну архітектуру системи наведено на рис. 2.1.

Взаємодія між клієнтом і сервером здійснюється через REST API з використанням HTTP-протоколу. Дані передаються у форматі JSON, що забезпечує легкість парсингу на будь-якій платформі та зручність налагодження. Кожен HTTP-запит є незалежним, що відповідає принципу stateless REST-архітектури.

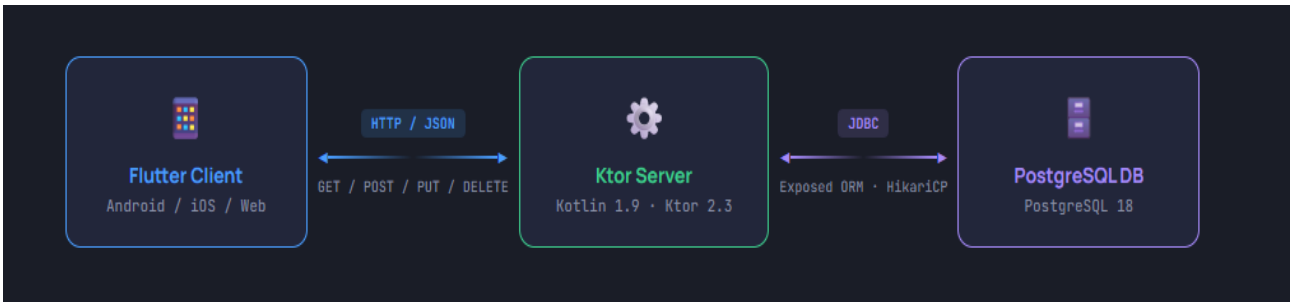

Рис. 2.1. Діаграма розгортання системи

Такий підхід забезпечує слабку зв'язність компонентів, дозволяє незалежно розвивати клієнтську та серверну частини і спрощує масштабування. У разі необхідності серверну частину можна замінити або розширити без жодних змін у клієнтському застосунку, і навпаки.

2.2. Проєктування бази даних

2.2.1. ER-діаграма та опис сутностей

Проєктування бази даних розпочалося зі створення ER-діаграми, яка відображає основні сутності та зв'язки між ними. ER-діаграму бази даних наведено на рис. 2.2. ER-діаграма є важливим інструментом моделювання, що дозволяє наочно

продемонструвати структуру даних ще до етапу реалізації та виявити потенційні проблеми в ранні терміни розробки.

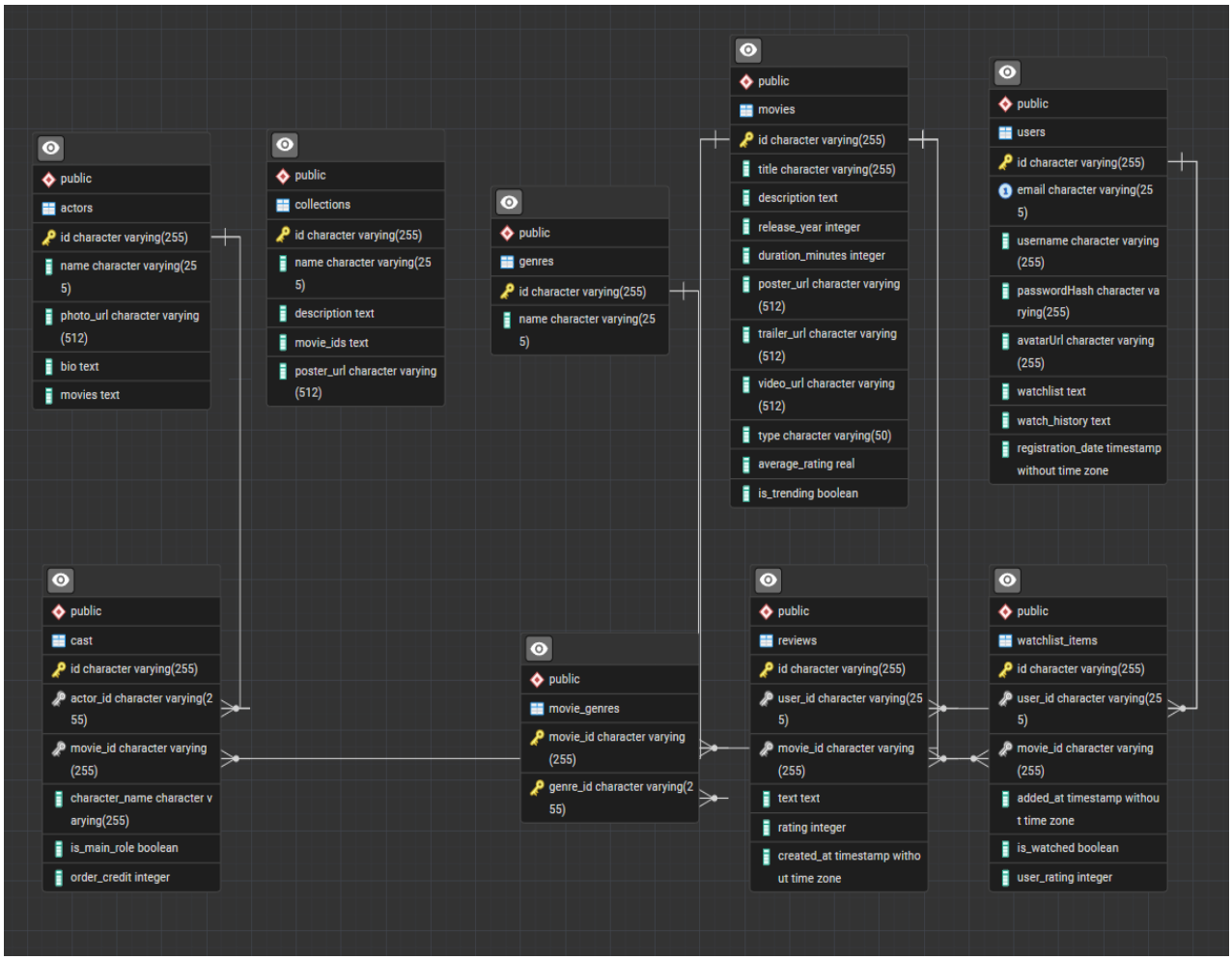

Рис. 2.2. ER-діаграма бази даних

Виділено такі основні сутності: users - користувачі системи, movies - фільми, genres - жанри, movie_genres - зв'язок багато-до-багатьох між фільмами та жанрами, actors - актори, cast - зв'язок між акторами та фільмами з додатковою інформацією про роль, reviews - коментарі користувачів, collections - тематичні підбірки.

2.2.2. Схема таблиць та зв'язків

На основі ER-діаграми розроблено фізичну схему бази даних. Нижче наведено детальний опис основних таблиць із зазначенням полів, їх типів та призначення.

Таблиця 2.1

Структура таблиці `users`

Назва поля	**Тип**	**Опис**
id	VARCHAR(255)	Первинний ключ, UUID
email	VARCHAR(255)	Email користувача, унікальний
username	VARCHAR(255)	Ім'я користувача
passwordHash	VARCHAR(255)	Хеш пароля (BCrypt)
avatarUrl	VARCHAR(512)	Посилання на аватар
watchlist	TEXT	JSON-список ID обраних фільмів
watchHistory	TEXT	JSON-список ID переглянутих фільмів
registrationDate	TIMESTAMP	Дата реєстрації

Таблиця 2.2

Структура таблиці `movies`

Назва поля	**Тип**	**Опис**
id	VARCHAR(255)	Первинний ключ, UUID
title	VARCHAR(255)	Назва фільму

Продовження табл. 2.2

title	VARCHAR(255)	Назва фільму
description	TEXT	Опис
releaseYear	INTEGER	Рік випуску
durationMinutes	INTEGER	Тривалість у хвилинах
posterUrl	VARCHAR(512)	Постер
trailerUrl	VARCHAR(512)	Трейлер (YouTube)
videoUrl	VARCHAR(512)	Посилання на відеофайл
type	VARCHAR(50)	Тип (movie, series, anime, cartoon)
averageRating	REAL	Середня оцінка
isTrending	BOOLEAN	Чи є популярним

Таблиця 2.3

Структура таблиці `cast`

Назва поля	**Тип**	**Опис**
id	VARCHAR(255)	Первинний ключ, UUID
actorId	VARCHAR(255)	Зовнішній ключ до actors.id

Продовження табл. 2.3

movieId	VARCHAR(255)	Зовнішній ключ до movies.id
characterName	VARCHAR(255)	Ім'я персонажа
isMainRole	BOOLEAN	Чи є головною роллю
orderCredit	INTEGER	Порядок у титрах

Зв'язки між таблицями реалізовано через зовнішні ключі. Таблиця cast є сполучною між movies та actors і додатково зберігає інформацію про роль актора у фільмі та порядок у титрах. Таблиця movie_genres забезпечує зв'язок багато-до-багатьох між фільмами та жанрами.

2.2.3. UML-діаграма класів

Для візуалізації структури програмної системи та взаємозв’язків між основними об’єктами предметної області було розроблено UML-діаграму класів. Діаграма відображає основні класи системи, їх атрибути та асоціативні зв’язки між ними. UML-діаграму класів наведено на рис. 2.3.

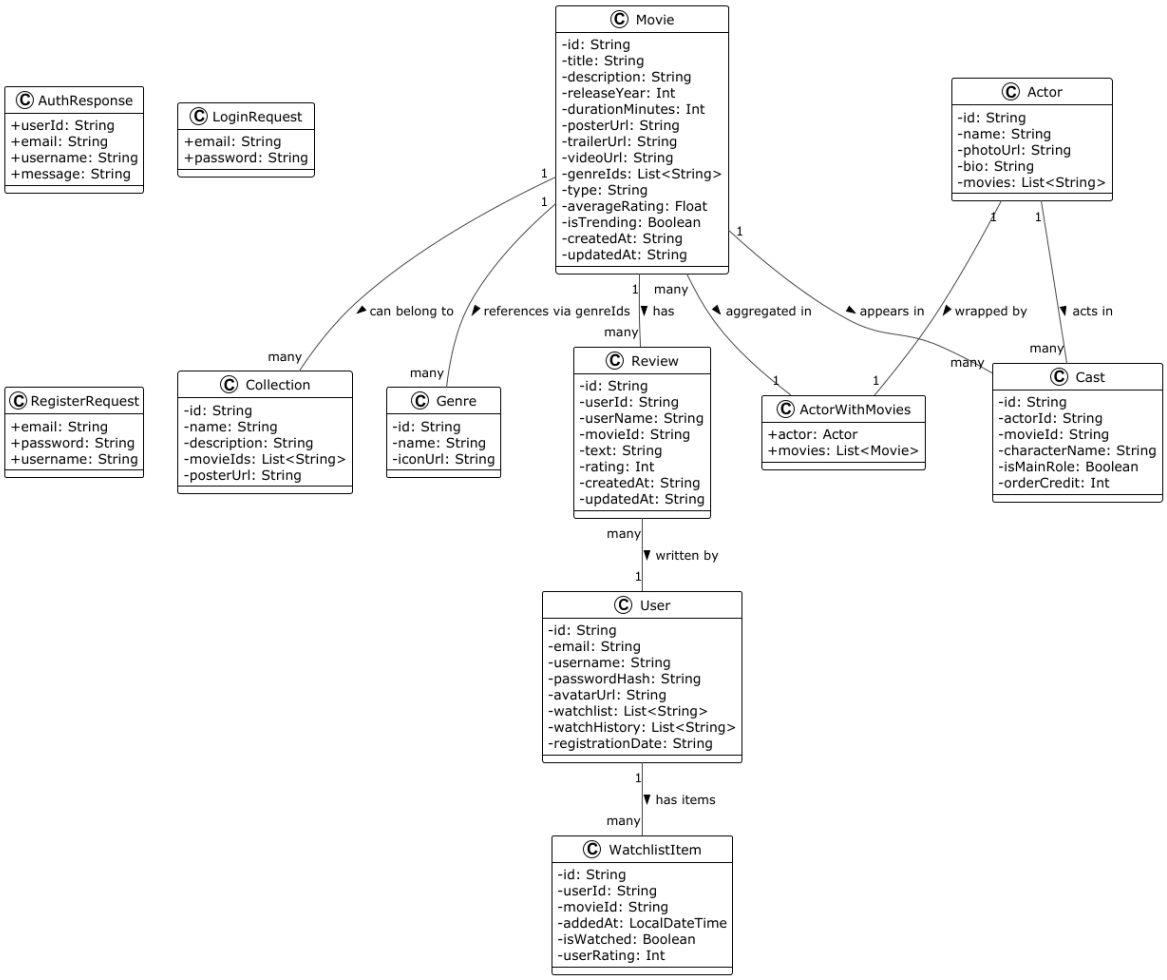

Рис. 2.3. UML-діаграма класів системи

Основними класами системи є User, Movie, Actor, Review, Genre та Collection. Клас Movie є центральним елементом системи та пов’язаний із жанрами, акторами й коментарями користувачів. Клас Review забезпечує зберігання оцінок і текстових відгуків користувачів до фільмів. Такий підхід дозволяє формалізувати структуру програмної системи ще до етапу реалізації та спрощує подальшу підтримку коду.

2.3. Проєктування API

2.3.1. Специфікація REST-ендпоінтів

Розроблено специфікацію API відповідно до принципів REST. Ендпоінти розподілено на публічні (доступні без автентифікації) та захищені (потребують

передачі ідентифікатора користувача через заголовок X-User-Id). Такий підхід забезпечує можливість перегляду контенту незареєстрованими користувачами, але обмежує доступ до персональних даних.

Таблиця 2.4.

Основні ендпоінти API

Метод	**URL**	**Опис**	**Доступ**
POST	/auth/register	Реєстрація нового користувача	публічний
POST	/auth/login	Вхід користувача	публічний
GET	/movies	Отримання списку фільмів (з фільтрами)	публічний
GET	/movies/{id}	Деталі фільму	публічний
GET	/actors	Список акторів	публічний
GET	/actors/{id}/with-movies	Актор з його фільмами	Публічний
GET	/cast/movie/{movieId}	Актори фільму	публічний
GET	/reviews/movie/{movieId}	Коментарі до фільму	публічний
POST	/reviews	Додавання коментаря	X-User-Id
POST	/users/favorites/{movieId}	Додати в обране	X-User-Id
DELETE	/users/favorites/{movieId}	Видалити з обраного	X-User-Id

Продовження табл. 2.4

GET	/users/favorites	Список обраних	X-User-Id
POST	/users/history/{movieId}	Додати в історію	X-User-Id
GET	/users/history	Історія переглядів	X-User-Id
GET	/collections	Список підбірок	публічний

2.3.2. Формати запитів та відповідей

Всі дані передаються у форматі JSON. Використання єдиного формату обміну даними спрощує розробку та підтримку як клієнтської, так і серверної частини.

Приклад тіла запиту для реєстрації:

```
{
  "email": "user@example.com",
  "password": "securepassword",
  "username": "user123"
}
```

Приклад успішної відповіді сервера:

```
{
  "userId": "a1b2c3d4-e5f6-7890-1234-567890abcdef",
  "email": "user@example.com",
  "username": "user123"
}
```

У разі виникнення помилки сервер повертає відповідний HTTP-статус код та JSON-об'єкт з полем error, що містить опис помилки. Це дозволяє клієнту коректно обробляти помилкові ситуації та відображати їх користувачу.

2.4. Архітектура програмних модулів

Серверна частина організована за шаруватою архітектурою (Layered Architecture). Шар маршрутизації (Routes) приймає вхідні HTTP-запити та передає їх у відповідні сервіси. Шар сервісів (Services) реалізує бізнес-логіку застосунку. Шар репозиторіїв (Repositories) відповідає за взаємодію з базою даних і повністю інкапсулює SQL-запити. Діаграму шарів архітектури наведено на рис. 2.4.

Такий поділ забезпечує незалежність шарів між собою, можливість тестування кожного шару окремо, а також легкість підтримки та розширення системи у майбутньому. Завдяки цій архітектурі, наприклад, заміна бази даних потребуватиме змін лише у шарі репозиторіїв.

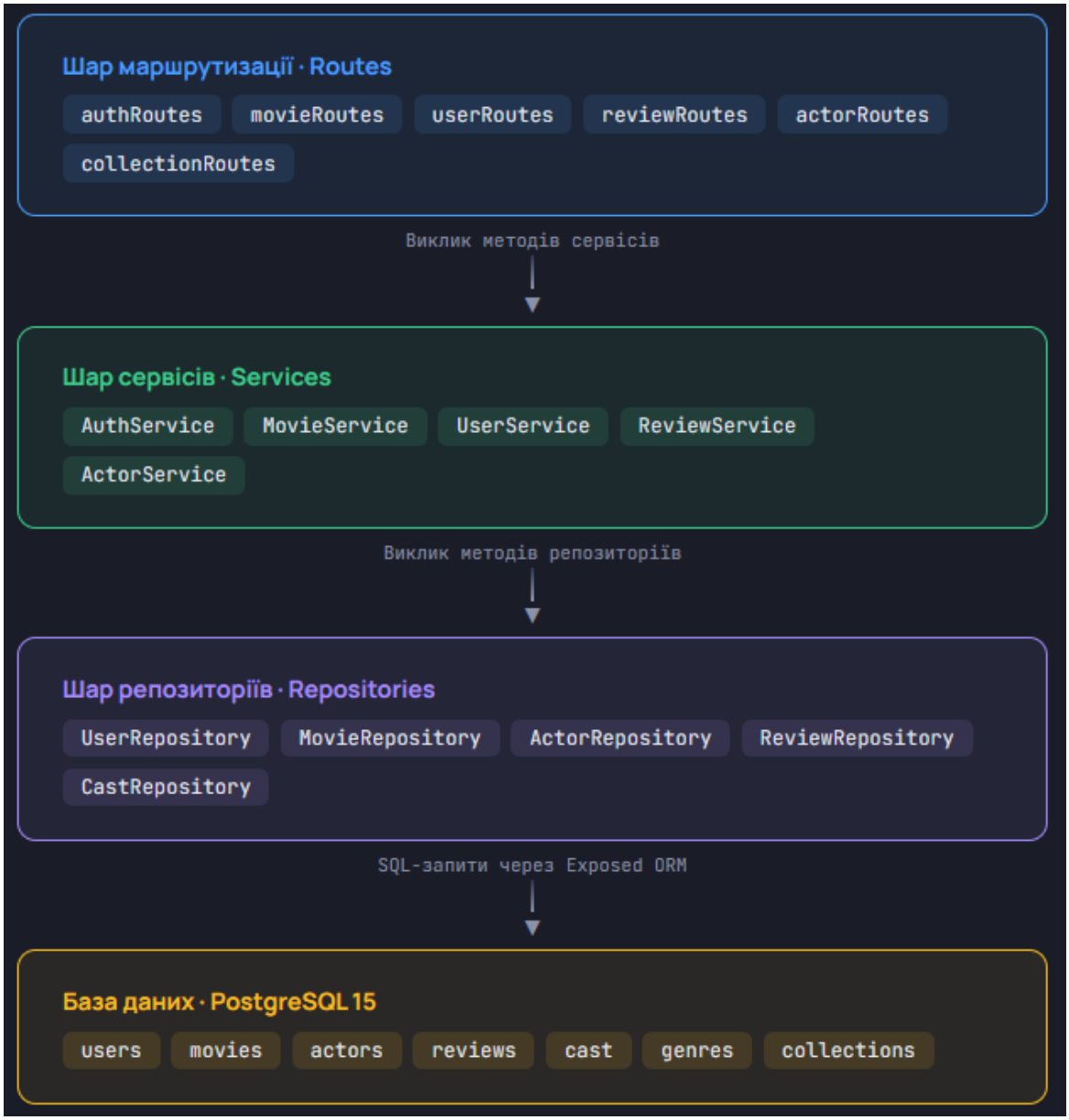

Рис. 2.4. Діаграма шарів архітектури (Layered Architecture)

2.5. Математичне та алгоритмічне забезпечення

2.5.1. Алгоритм формування динамічного SQL-запиту (getWithFilters)

Метод *getWithFilters* реалізує динамічне формування SQL-запиту на основі необов'язкових параметрів фільтрації. Алгоритм будує предикатний вираз за схемою кон'юнкції незалежних умов.

Вхідні параметри: genre: String?, type: String?, year: Int?

Результат: список об'єктів Movie, що задовольняють умови.

Алгоритм:

1. Ініціалізувати базовий вираз: *query ← SELECT * FROM movies WHERE TRUE*
2. Якщо genre ≠ null:
   query ← query AND id IN (SELECT movie_id FROM movie_genres WHERE genre_name = genre)
3. Якщо type ≠ null:
   query ← query AND type = type
4. Якщо year ≠ null:
   query ← query AND release_year = year
5. Виконати query, повернути список Movie.

Формально предикат запиту записується формулою (2.1):

$$P(m) = P\_genre(m) \wedge P\_type(m) \wedge P\_year(m) \qquad \textbf{(2.1)}$$

де кожна умова P_i є або тотожно істинною (якщо відповідний параметр не передано), або конкретним обмеженням. Такий підхід дозволяє генерувати від 1 до $2^3 = 8$ варіантів запиту без дублювання коду.

2.5.2. Алгоритм текстового пошуку фільмів

Пошук фільмів за назвою реалізовано через нечутливе до регістру часткове збігання підрядка. Алгоритм:

Вхідний параметр: query: String

1. Перетворити query у нижній регістр: *q ← toLower(query)*
2. Сформувати умову: *LOWER(title) LIKE '%' || q || '%'*
3. Виконати запит, повернути відповідні записи.

Складність пошуку за відсутності повнотекстового індексу становить **O(n · |title|)**, де n — кількість фільмів у базі, |title| — довжина назви. Для масштабування понад ~10 000 записів рекомендується замінити LIKE на PostgreSQL-оператор @@ з використанням *tsvector*-індексу, що суттєво зменшує час пошуку завдяки використанню індексів.

2.5.3. Формула обчислення середньої оцінки фільму

Середня оцінка фільму обчислюється як середнє арифметичне всіх виставлених оцінок за 10-бальною шкалою. Нехай $R = \{r_1, r_2, \ldots, r_n\}$ — множина оцінок, де кожна $r_i \in \{1, 2, \ldots, 10\}$. Тоді згідно з формулою (2.2):

$$\overline{\mathbf{R}} = \frac{\mathbf{1}}{\mathbf{n}} \sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{r_i}, \qquad \mathbf{n \geq 1} \qquad \mathbf{(2.2)}$$

де n — кількість оцінок для конкретного фільму.

Якщо оцінок немає (n = 0), значення averageRating залишається рівним 0.0. Після кожного нового відгуку поле averageRating у таблиці movies оновлюється транзакційно (лістинг 2.1):

Лістинг 2.1 – SQL-запит оновлення середньої оцінки фільму

```
UPDATE movies
SET average_rating = (SELECT AVG(rating) FROM reviews WHERE movie_id = :id)
WHERE id = :id
```

Діапазон значення: якщо є хоча б одна оцінка (n≥1), то $\overline{\mathbf{R}} \in [1.0, 10.0]$; у разі відсутності оцінок (n=0) значення *averageRating* дорівнює 0.0, що сигналізує про відсутність рейтингу. Такий підхід забезпечує вищу диференціацію порівняно з 5-бальною шкалою (стандартне відхилення на тій самій вибірці є вдвічі більшим).

2.5.4. Алгоритм генерації UUID та забезпечення унікальності первинних ключів

Для ідентифікації сутностей (користувачі, фільми, актори, коментарі) застосовано UUID версії 4 (RFC 4122). UUID версії 4 є випадковим 128-бітним числом з фіксованими бітами версії та варіанту, що описується формулою (2.3):

$$xxxxxxxx - xxxx - 4xxx - yxxx - xxxxxxxxxxxx \quad (2.3)$$

де x — довільна шістнадцяткова цифра, 4 — маркер версії, y ∈ {8, 9, a, b} — маркер варіанту.

Алгоритм генерації (JVM, java.util.UUID):

1. Згенерувати 128 випадкових біт через *SecureRandom* (криптографічно стійкий ГПСЧ).
2. Встановити біти 12–15 у значення 0100 (версія 4).
3. Встановити біти 62–63 у значення 10 (варіант RFC 4122).
4. Відформатувати як рядок з дефісами: 8-4-4-4-12 символів.

У коді серверної частини генерація виконується безпосередньо перед збереженням запису (лістинг 2.2):

Лістинг 2.2 – Генерація UUID

```
val newId = if (review.id.isNullOrBlank())
    UUID.randomUUID().toString() else review.id!!
```

Імовірність колізії двох випадково згенерованих UUID версії 4 при n записах обчислюється за формулою (2.4):

$$\boldsymbol{P}(\text{колізія}) \approx \frac{\boldsymbol{n^2}}{\boldsymbol{2 \cdot 2^{122}}} \qquad \textbf{(2.4)}$$

При n = 10^6 записів ця ймовірність становить приблизно 10^{-28}, що для практичних цілей вважається нульовою.

2.5.5. Алгоритм хешування паролів (BCrypt)

Автентифікація користувачів базується на алгоритмі BCrypt, що реалізує адаптивне хешування з параметром складності (cost factor) c. Час обчислення хешу зростає експоненційно: **T ∝ 2^c**.

Функція хешування відповідно до формули (2.5):

$$\boldsymbol{H = BCrypt(password,\ salt,\ c)} \qquad \textbf{(2.5)}$$

де salt — 128-бітна криптографічна сіль, яка генерується автоматично та зберігається в складі хешу; c = 10 (значення за замовчуванням бібліотеки *at.favre.lib:bcrypt:0.9.0*), що забезпечує ~100–300 мс на генерацію хешу на сучасному обладнанні.

Алгоритм перевірки пароля при вході:

1. Отримати з БД запис користувача за email.
2. Якщо запис не знайдено — повернути 401 Unauthorized.
3. Викликати *BCrypt.verifyer().verify(inputPassword, storedHash)*.
4. Якщо результат *verified = false* — повернути 401 Unauthorized.
5. Інакше — повернути дані користувача (200 OK).

Оскільки сіль вбудована у рядок хешу, зловмисник не може використати попередньо обчислені таблиці (rainbow tables) для злому паролів.

РОЗДІЛ 3 ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Реалізація серверної частини

3.1.1. Налаштування проєкту та залежності

Проєкт серверної частини створено з використанням системи збірки Gradle з Kotlin DSL, що забезпечує типобезпечне налаштування та зручну роботу в IntelliJ IDEA. Файл build.gradle.kts містить усі необхідні залежності та налаштування плагінів. Нижче наведено фрагмент конфігураційного файлу (лістинг 3.1):

Лістинг 3.1 – Фрагмент конфігураційного файлу build.gradle.kts

```
plugins {
    kotlin("jvm") version "1.9.22"
    kotlin("plugin.serialization") version "1.9.22"
    application
}
dependencies {
    implementation("io.ktor:ktor-server-core:2.3.7")
    implementation("io.ktor:ktor-server-netty:2.3.7")
    implementation("io.ktor:ktor-server-content-
negotiation:2.3.7")
    implementation("io.ktor:ktor-serialization-kotlinx-
json:2.3.7")
    implementation("org.postgresql:postgresql:42.6.0")
    implementation("org.jetbrains.exposed:exposed-
core:0.41.1")
```

```
    implementation("org.jetbrains.exposed:exposed-
jdbc:0.41.1")
    implementation("org.jetbrains.exposed:exposed-java-
time:0.41.1")
    implementation("com.zaxxer:HikariCP:5.0.1")
    implementation("at.favre.lib:bcrypt:0.9.0")
    implementation("org.jetbrains.kotlinx:kotlinx-
serialization-json:1.6.3")
    // ... інші залежності
}
```

3.1.2. Моделі даних

3.1.2.1. Модель користувача

Модель User визначено як data-клас з анотацією @Serializable, що забезпечує автоматичну серіалізацію та десеріалізацію об'єктів при передачі через мережу. Його структура наведена в лістингу 3.2. Використання data-класів у Kotlin дозволяє автоматично генерувати методи equals, hashCode та toString, що спрощує порівняння та логування об’єктів.

Лістинг 3.2 – Модель даних User

```
@Serializable
data class User(
    var id: String? = null,
    var email: String? = null,
    var username: String? = null,
```

```
    var passwordHash: String? = null,
    var avatarUrl: String? = null,
    var watchlist: List<String> = emptyList(),
    var watchHistory: List<String> = emptyList(),
    var registrationDate: String? = null
)
```

Поле registrationDate зберігається як рядок для спрощення серіалізації, оскільки робота з LocalDateTime викликала проблеми з кодуванням. Списки watchlist та watchHistory зберігаються у базі даних як JSON-рядки, а при зчитуванні десеріалізуються у колекції Kotlin.

3.1.2.2. Моделі фільмів, жанрів, акторів

Модель Movie містить усі необхідні поля для відображення інформації про фільм, включаючи посилання на медіаресурси. Модель Actor забезпечує збереження біографічних даних та фільмографії актора. Інформація про фільми, включно з посиланнями на медіафайли, представлена класом Movie (лістинг 3.3). Для акторів передбачено клас Actor, який містить біографічні дані та список ідентифікаторів фільмів (лістинг 3.4.).

Модель Movie:

Лістинг 3.3 – Модель даних Movie

```
@Serializable
data class Movie(
    var id: String? = null,
```

```
    var title: String? = null,
    var description: String? = null,
    var releaseYear: Int? = null,
    var durationMinutes: Int? = null,
    var posterUrl: String? = null,
    var trailerUrl: String? = null,
    var videoUrl: String? = null,
    var genreIds: List<String>? = null,
    var type: String? = null,
    var averageRating: Float = 0f,
    var isTrending: Boolean = false
)
```

Модель Actor:

Лістинг 3.4 – Модель даних Actor

```
@Serializable
data class Actor(
    var id: String? = null,
    var name: String? = null,
    var photoUrl: String? = null,
    var bio: String? = null,
    var movies: List<String> = emptyList()
```

```
)
```

3.1.2.3. Моделі для автентифікації

Для обробки запитів на реєстрацію та вхід до системи створено окремі класи-запити та клас відповіді. Розділення моделей запиту і відповіді є хорошою практикою, яка дозволяє чітко визначити контракт між клієнтом і сервером та приховати внутрішні деталі реалізації. Окремі класи-запити та відповіді забезпечують чіткий контракт між клієнтом і сервером (лістинг 3.5).

Лістинг 3.5 – Моделі запитів та відповіді для автентифікації

```
@Serializable
data class RegisterRequest(
    val email: String,
    val password: String,
    val username: String
)

@Serializable
data class LoginRequest(
    val email: String,
    val password: String
)

 @Serializable
```

```
data class AuthResponse(
    val userId: String,
    val email: String,
    val username: String,
    val message: String? = null
)
```

3.1.3. Реалізація репозиторіїв

3.1.3.1. Базовий репозиторій та CRUD-операції

Всі репозиторії реалізують спільний інтерфейс Repository<T, ID>, який визначає базові CRUD-операції. Завдяки цьому досягається однорідність коду та можливість написання узагальненої логіки. Всі операції є підвішеними (suspend), що дозволяє виконувати їх у корутинах без блокування основного потоку. Нижче наведено приклад реалізації UserRepository (лістинг 3.6):

Лістинг 3.6 – Приклад реалізації UserRepository (метод getById)

```
class UserRepository : Repository<User, String> {
    private val json = Json { ignoreUnknownKeys = true }
    override suspend fun getById(id: String): User? =
withContext(Dispatchers.IO) {
        transaction {
            Users.select { Users.id eq id }
                .map { rowToUser(it) }
                .singleOrNull()
```

```
        }
    }
    // ... інші методи
}
```

3.1.3.2. Спеціалізовані методи

Окрім стандартних CRUD-операцій, репозиторії містять спеціалізовані методи для виконання більш складних запитів. Наприклад, метод getWithFilters у MovieRepository дозволяє динамічно формувати SQL-запит залежно від переданих параметрів фільтрації (лістинг 3.7):

Лістинг 3.7 – Метод динамічної фільтрації фільмів (getWithFilters)

```
suspend fun getWithFilters(genreId: String?, type: String?,
year: Int?): List<Movie> =
    withContext(Dispatchers.IO) {
    transaction {
        var query = Movies.selectAll()
        if (genreId != null) {
            val movieIds =
MovieGenres.slice(MovieGenres.movieId)
                .select { MovieGenres.genreId eq genreId }
                .map { it[MovieGenres.movieId] }
            query = query.andWhere { Movies.id inList movieIds
}
```

```
        }
        if (type != null) { query = query.andWhere { Movies.type eq type } }
        if (year != null) { query = query.andWhere { Movies.releaseYear eq year } }
        query.map { rowToMovie(it) }
    }
}
```

Перевага такого підходу полягає в тому, що при відсутності будь-якого фільтра відповідна умова до запиту просто не додається, і запит повертає всі наявні записи. Це дозволяє використовувати один метод для різних комбінацій фільтрів.

3.1.4. Реалізація сервісів

3.1.4.1. Сервіс автентифікації

AuthService відповідає за реєстрацію та вхід користувачів до системи. При реєстрації сервіс спочатку перевіряє унікальність email-адреси, після чого хешує пароль за допомогою алгоритму BCrypt та зберігає нового користувача в базі даних (лістинг 3.8). BCrypt є галузевим стандартом для зберігання паролів завдяки вбудованій функції «солення» та налаштовуваному параметру cost factor: [6]

Лістинг 3.8 – Реєстрація користувача з хешуванням пароля BCrypt

```
suspend fun register(request: RegisterRequest): AuthResponse {
    val existingUser = userRepository.findByEmail(request.email)
```

```
    if (existingUser != null) {
        throw IllegalArgumentException("Користувач з таким email вже існує")
    }
    val passwordHash = BCrypt.withDefaults()
        .hashToString(12, request.password.toCharArray())
    val newUser = User(
        email = request.email, passwordHash = passwordHash,
        username = request.username, watchlist = emptyList(),
        watchHistory = emptyList(),
        registrationDate = LocalDateTime.now().toString()
    )
    val createdUser = userRepository.create(newUser)
    return AuthResponse(userId = createdUser.id!!,
        email = createdUser.email!!, username = createdUser.username!!)
}
```

3.1.4.2. Сервіс роботи з фільмами

MovieService надає методи для отримання фільмів з урахуванням заданих фільтрів (лістинг 3.9). Сервіс делегує виконання запиту до відповідного репозиторію, при цьому відповідаючи за валідацію вхідних параметрів та формування результату:

Лістинг 3.9 – Отримання списку фільмів з урахуванням фільтрів

```
suspend fun getMovies(genreId: String?, type: String?, year: Int?): List<Movie> =
    movieRepository.getWithFilters(genreId, type, year)
```

3.1.4.3. Сервіси для акторів та коментарів

ActorService містить метод getActorWithMovies, який повертає комплексний об'єкт - актора разом зі списком фільмів, у яких він знімався (лістинг 3.10). Такий підхід дозволяє мінімізувати кількість мережевих запитів з боку клієнта:

Лістинг 3.10 – Отримання актора разом з його фільмами

```
suspend fun getActorWithMovies(actorId: String): Pair<Actor?, List<Movie>> {
    val actor = actorRepository.getById(actorId) ?: return null to emptyList()
    val castList = castRepository.getByActor(actorId)
    val movieIds = castList.map { it.movieId }.filterNotNull()
    val movies = movieRepository.getByIds(movieIds)
    return actor to movies
}
```

3.1.5. Маршрутизація

3.1.5.1. Публічні маршрути

Публічні маршрути обробляють запити без будь-якої перевірки автентифікації. Параметри фільтрації передаються через query-параметри URL, що відповідає стандартам REST. Приклад маршруту для отримання списку фільмів (лістинг 3.11):

Лістинг 3.11 – Маршрут для отримання списку фільмів

```
get("/movies") {
    val genreId = call.request.queryParameters["genre"]
    val type = call.request.queryParameters["type"]
    val year =
call.request.queryParameters["year"]?.toIntOrNull()
    val movies = movieService.getMovies(genreId, type, year)
    call.respond(movies)
}
```

3.1.5.2. Захищені маршрути

Захищені маршрути перевіряють наявність та коректність заголовка X-User-Id перед виконанням будь-якої операції. У разі відсутності заголовка повертається відповідь з кодом 401 Unauthorized. Приклад маршруту для додавання фільму до списку обраних (лістинг 3.12):

Лістинг 3.12 – Захищений маршрут для додавання фільму до обраного

```
post("/users/favorites/{movieId}") {
    val userId = call.request.headers["X-User-Id"]
    if (userId.isNullOrBlank()) {
        call.respond(HttpStatusCode.Unauthorized,
```

```
            mapOf("error" to "Missing user ID"))
        return@post
    }
    // ... логіка додавання
}
```

3.1.6. Робота з базою даних

3.1.6.1. Підключення через HikariCP

Для ефективного керування підключеннями до бази даних використано бібліотеку HikariCP, яка є одним з найшвидших пулів підключень для JVM. Пул дозволяє повторно використовувати вже встановлені з'єднання, що суттєво знижує накладні витрати на відкриття нових підключень при кожному запиті (лістинг 3.13). Конфігурацію підключення винесено в окремий об'єкт DatabaseFactory: [5]

Лістинг 3.13 – Конфігурація пулу підключень HikariCP

```
object DatabaseFactory {
    private val dataSource by lazy {
        HikariDataSource(HikariConfig().apply {
            jdbcUrl = "jdbc:postgresql://localhost:5432/datflix"
            driverClassName = "org.postgresql.Driver"
            username = System.getenv("DB_USERNAME") ?: "postgres"
    password = System.getenv("DB_PASSWORD") ?: "password"
```

```
            maximumPoolSize = 10
            isAutoCommit = false
        })
    }
    fun init() { Database.connect(dataSource) }
}
```

У production-середовищі конфіденційні параметри підключення (логін, пароль, URL бази даних) повинні передаватися через змінні середовища (environment variables) або систему керування секретами, а не зберігатися безпосередньо у вихідному коді.

3.1.6.2. Використання Exposed для запитів

Для формування SQL-запитів використовується DSL Exposed від JetBrains, що забезпечує типобезпечну роботу з базою даних та дозволяє писати запити у стилі Kotlin замість рядків SQL (лістинг 3.14). Це знижує ймовірність синтаксичних помилок у запитах. Приклад запиту для отримання коментарів фільму разом з іменами користувачів через JOIN: [4]

Лістинг 3.14 – Отримання коментарів фільму з іменами користувачів через JOIN

```
Reviews.innerJoin(Users, { Reviews.userId }, { Users.id })
    .slice(Reviews.columns + Users.username)
    .select { Reviews.movieId eq movieId }
    .map { rowToReview(it) }
```

3.1.6.3. Міграція даних з Firestore

На початковому етапі проєктування планувалося використання хмарної бази даних Firestore. Однак у ході розробки було прийнято рішення перейти на PostgreSQL для отримання більшого контролю над даними та уникнення залежності від зовнішніх сервісів. Для перенесення накопичених даних написано скрипт Migration.kt, який читає документи з Firestore, перетворює їх на об'єкти моделей і вставляє в таблиці PostgreSQL.

Лістинг 3.15 – Міграція фільмів з Firestore до PostgreSQL

```
private fun migrateMovies() {
    println("Міграція фільмів")
    try {
        val snapshot =
firestore.collection("movies").get().get()
        val documents = snapshot.documents
        transaction {
            for (document in documents) {
                val id = document.id
                val title = document.getString("title") ?:
""
                val description =
document.getString("description")
                val releaseYear =
document.getLong("releaseYear")?.toInt()
                val durationMinutes =
document.getLong("durationMinutes")?.toInt()
                val posterUrl =
document.getString("posterUrl")
                val trailerUrl =
```

```
document.getString("trailerUrl")
                val videoUrl =
document.getString("videoUrl")
                val type = document.getString("type")
                val averageRating =
(document.getDouble("averageRating") ?: 0.0).toFloat()
                val isTrending =
document.getBoolean("isTrending") ?: false

                @Suppress("UNCHECKED_CAST")
                val genreIds = (document.get("genreIds") as?
List<String>) ?: emptyList()

                Movies.insertIgnore {
                    it[Movies.id] = id
                    it[Movies.title] = title
                    it[Movies.description] = description
                    it[Movies.releaseYear] = releaseYear
                    it[Movies.durationMinutes] =
durationMinutes
                    it[Movies.posterUrl] = posterUrl
                    it[Movies.trailerUrl] = trailerUrl
                    it[Movies.videoUrl] = videoUrl
                    it[Movies.type] = type
                    it[Movies.averageRating] = averageRating
                    it[Movies.isTrending] = isTrending
```

3.2. Реалізація клієнтської частини

3.2.1. Структура проєкту

Проєкт Flutter організовано за функціональним принципом, де кожна директорія відповідає за окремий аспект застосунку: lib/models містить класи даних, lib/providers - Riverpod провайдери для управління станом, lib/services - класи для виконання HTTP-запитів до API, lib/screens - екрани застосунку, lib/widgets - перевикористовувані UI-компоненти, lib/l10n - файли локалізації. Структуру проєкту в середовищі Android Studio наведено на рис. 3.1.

Такий поділ дозволяє легко орієнтуватись у коді та мінімізує зв'язність між компонентами. Кожен модуль може бути змінений або замінений незалежно від інших.

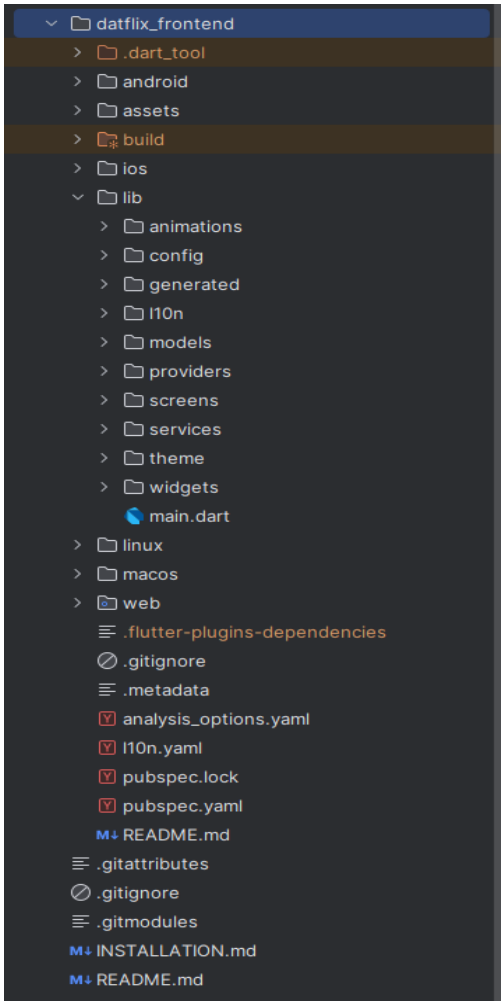

Рис. 3.1. Структура проєкту в Android Studio

3.2.2. Моделі даних на клієнті

Клієнтські моделі даних відповідають серверним і містять фабричний конструктор fromJson для десеріалізації відповідей API. Наприклад, модель Movie (лістинг 3.16). Використання фабричного конструктора є стандартною практикою у Flutter для перетворення JSON на об'єкти Dart. Наприклад, модель Movie:

Лістинг 3.16 – Модель даних Movie

```
class Movie {
  final String id;
  final String title;
  final String? posterUrl;
  final List<String>? genreIds;
  final int? releaseYear;
  final String? description;
  final String? trailerUrl;
  final String? videoUrl;
  final String? type;

  Movie({...});

  factory Movie.fromJson(Map<String, dynamic> json) {
    return Movie(
      id: json['id'].toString(),
      title: json['title'] ?? '',
```

```
      posterUrl: json['posterUrl']?.toString(),
      genreIds: (json['genreIds'] as List?)?.map((e) =>
e.toString()).toList(),
      releaseYear: json['releaseYear'] as int?,
      description: json['description']?.toString(),
      trailerUrl: json['trailerUrl']?.toString(),
      videoUrl: json['videoUrl']?.toString(),
      type: json['type']?.toString(),
    );
  }
}
```

3.2.3. Організація мережевих запитів

3.2.3.1. Класи для роботи з API

Для кожної групи ендпоінтів створено окремий клас-сервіс, що відповідає принципу єдиної відповідальності. Класи інкапсулюють логіку HTTP-запитів, формування URL та обробку відповідей. Наприклад, AuthService містить методи для реєстрації та входу (лістинг 3.17):

Лістинг 3.17 – Клас AuthService з методом login

```
class AuthService {
  final String baseUrl;
  AuthService({required this.baseUrl});
```

```
  Future<Map<String, dynamic>> login(String email, String
password) async {
    final response = await http.post(
      Uri.parse('$baseUrl/auth/login'),
      headers: {'Content-Type': 'application/json'},
      body: jsonEncode({'email': email, 'password':
password}),
    );
    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);
      final prefs = await SharedPreferences.getInstance();
      await prefs.setString('userId', data['userId']);
      return data;
    } else {
      throw Exception(jsonDecode(response.body)['error'] ??
'Login failed');
    }
  }
  // ... інші методи
}
```

3.2.3.2. Обробка відповідей та помилок

Усі мережеві виклики обгорнуті в блоки try-catch, що запобігає аварійному завершенню застосунку при мережевих помилках. Помилки відображаються користувачу через компонент SnackBar, який показує короткі інформаційні повідомлення у нижній частині екрану. Перевірка HTTP-статус кодів дозволяє розрізняти успішні відповіді від помилкових та відповідно реагувати.

3.2.4. Управління станом за допомогою Riverpod

3.2.4.1. Провайдери для фільмів

Для управління станом застосунку обрано бібліотеку Riverpod, яка є сучасною альтернативою Provider з покращеною типобезпекою та підтримкою асинхронних операцій. Провайдер moviesProvider відповідає за завантаження списку фільмів з урахуванням активних фільтрів (лістинг 3.18):

Лістинг 3.18 – Провайдер moviesProvider та клас MoviesNotifier

```
final moviesProvider = StateNotifierProvider<
    MoviesNotifier, AsyncValue<List<Movie>>>((ref) {
  return MoviesNotifier('http://10.0.2.2:8000', ref);
});


class MoviesNotifier extends
StateNotifier<AsyncValue<List<Movie>>> {
  // ... реалізація
}
```

3.2.4.2. Провайдери для обраного та історії

favoritesProvider та historyProvider побудовані аналогічно до providera фільмів і містять додатковий метод reload(), який викликається після зміни авторизованого користувача. Це гарантує, що нові дані завантажуються для відповідного користувача і попередні дані не залишаються у стані після виходу з системи.

3.2.4.3. Провайдери для акторів та коментарів

Для акторів та коментарів використано FutureProvider.family, оскільки ці дані завантажуються один раз при відкритті відповідної сторінки і не потребують складного управління станом. Приклад такого провайдера показано в лістингу 3.19.Параметр family дозволяє передавати аргумент (ID актора або фільму) безпосередньо до провайдера:

Лістинг 3.19 – Провайдер actorWithMoviesProvider (FutureProvider.family)

```
final actorWithMoviesProvider = FutureProvider.family<
    Map<String, dynamic>?, String>((ref, actorId) async {
  final response = await http.get(
      Uri.parse('http://10.0.2.2:8000/actors/$actorId/with-
movies'));
  // ... обробка
});
```

3.2.5. Реалізація екранів

3.2.5.1. Екран автентифікації

LoginPage реалізує єдиний екран для реєстрації та входу з перемиканням між режимами через булеву змінну стану _isLogin. Форма містить поля для введення email та пароля (і додатково імені для реєстрації). Після успішної автентифікації

відбувається перехід на головний екран застосунку. Скріншоти інтерфейсу подано на рис. 3.2 та рис. 3.3.

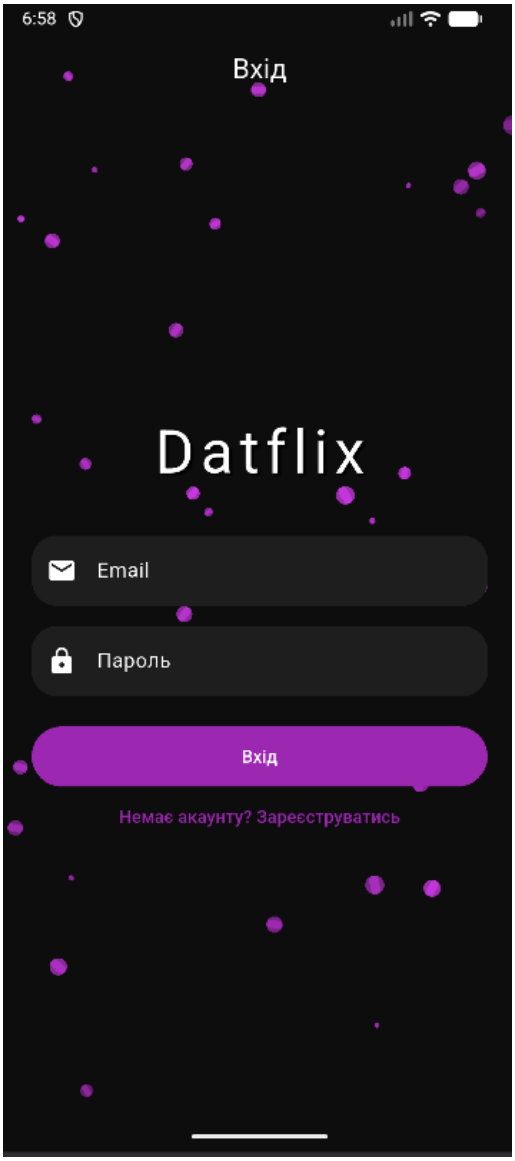

Рис. 3.2. Сторінка входу

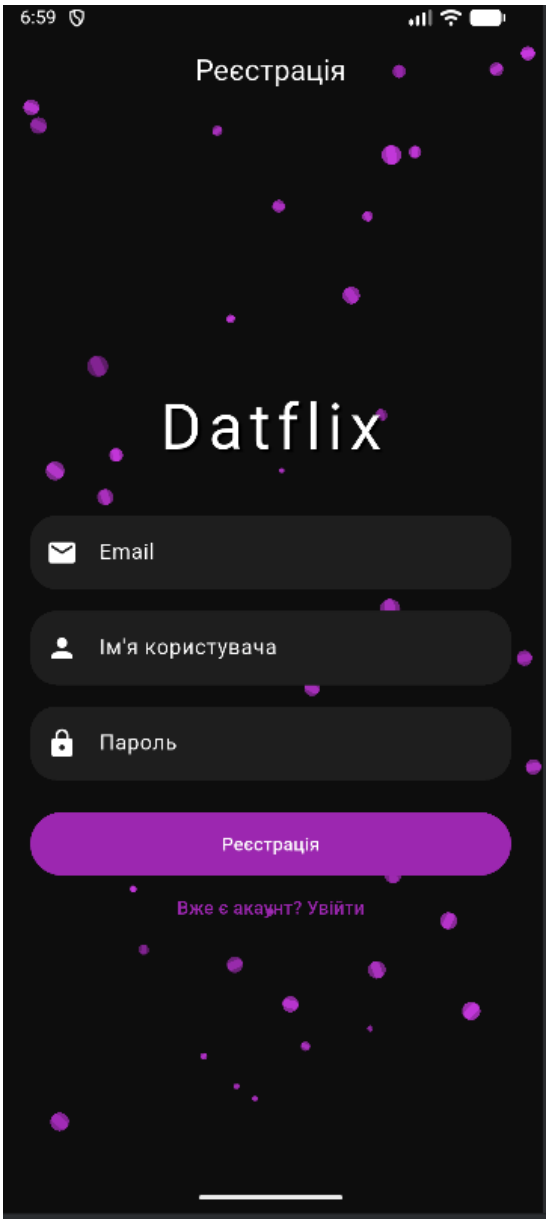

Рис. 3.3. Сторінка реєстрації

3.2.5.2. Головний екран

HomePage відображає горизонтальні каруселі з фільмами: загальну карусель зі всіма фільмами та окремі каруселі для кожного жанру. Дані для відображення отримуються з moviesProvider та genresProvider. Горизонтальне прокручування каруселей забезпечує зручний перегляд великої кількості контенту без необхідності скролити всю сторінку. Інтерфейс головного екрану наведено на рис. 3.4.

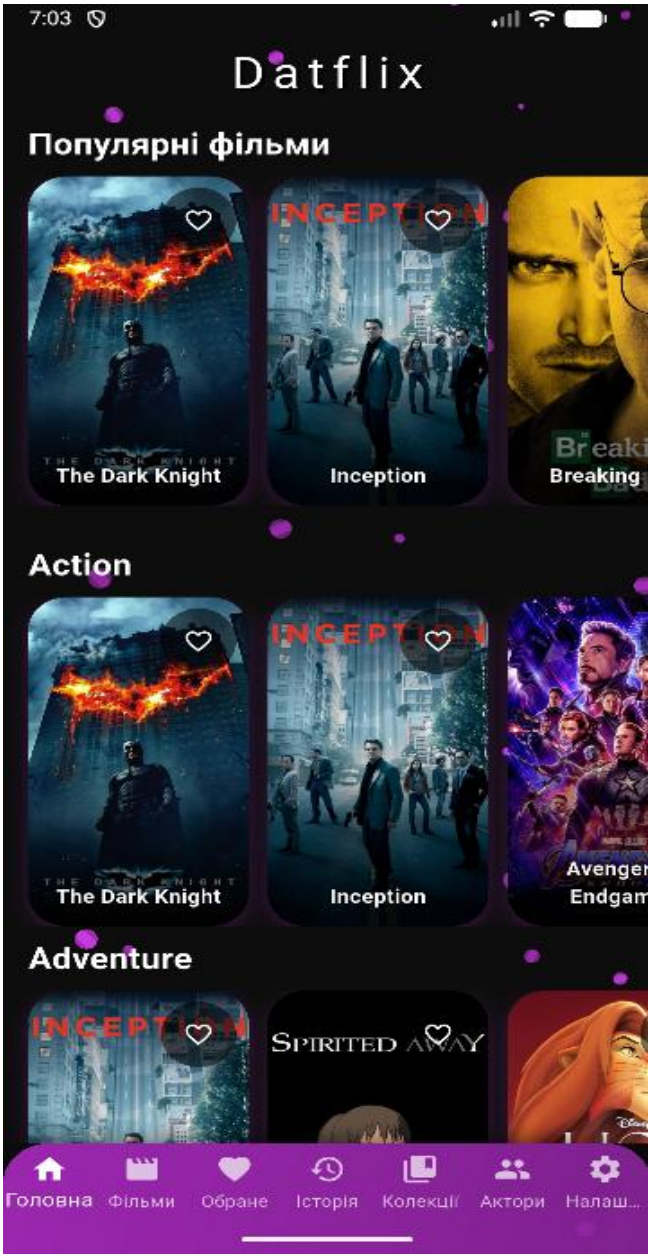

Рис. 3.4. Головний екран застосунку

3.2.5.3. Екран списку фільмів з фільтрацією

MoviesListPage є основним екраном для пошуку та перегляду каталогу фільмів. Вона містить текстове поле для пошуку за назвою та кнопку виклику діалогу фільтрів. У діалозі можна обрати жанр, тип контенту та рік випуску. При застосуванні фільтрів оновлюється filtersProvider, що автоматично ініціює перезавантаження списку через реактивний механізм Riverpod. Інтерфейс екрану подано на рис. 3.5.

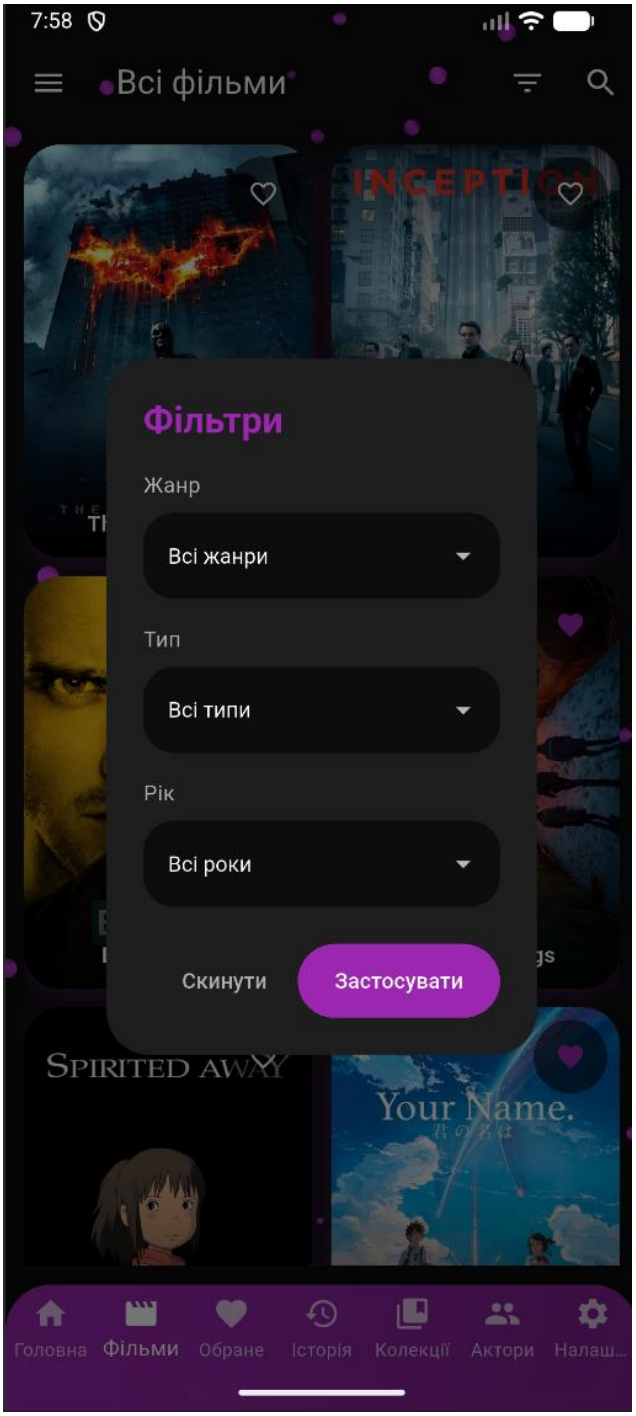

Рис. 3.5. Екран списку фільмів з фільтрацією

3.2.5.4. Детальна сторінка фільму

Сторінка MovieDetailPage є найскладнішою в застосунку. Вона реалізована як ConsumerStatefulWidget і отримує об'єкт Movie через параметр. Асинхронні дані (актори, коментарі) завантажуються через FutureProvider.family.

Інтерфейс сторінки побудований у вигляді послідовності логічних блоків (рис. 3.6–3.13):

1. Постер з градієнтом – зверху сторінки, з накладеним градієнтом для покращення читабельності тексту. Назва, рік, тривалість та тип відображаються у вигляді чипів (рис. 3.6).
2. Опис фільму – представлений у скляній картці з напівпрозорим фоном (рис. 3.7).
3. Відеоплеєр (Chewie) – відображається лише за наявності videoUrl та підтримує повноекранний режим і зміну швидкості відтворення (рис. 3.8).
4. Блок оцінки – містить середню оцінку, кількість оцінок, 10-зіркову шкалу та кнопку «Оцінити», яка відкриває RatingDialog. Після відправки оцінки виконується запит POST/PUT до /reviews, після чого середнє значення оновлюється (рис. 3.9).
5. Блок трейлера – замість вбудованого YouTube-плеєра використано відкриття відео через браузер за допомогою url_launcher, що підвищило продуктивність на слабких пристроях (рис. 3.10).
6. Список акторів – горизонтальний ListView.builder з аватаром, іменем та роллю актора. При натисканні відкривається ActorDetailPage (рис. 3.11).
7. Коментарі – список відгуків із аватаром, ім’ям користувача, датою у відносному форматі та текстом. Система оцінювання винесена у верхній блок (рис. 3.12).
8. Форма додавання коментаря – містить 10-зіркову оцінку та текстове поле. Після відправки виконується POST /reviews, а список оновлюється без перезавантаження сторінки (рис. 3.13).

Рис. 3.6. Постер з градієнтом

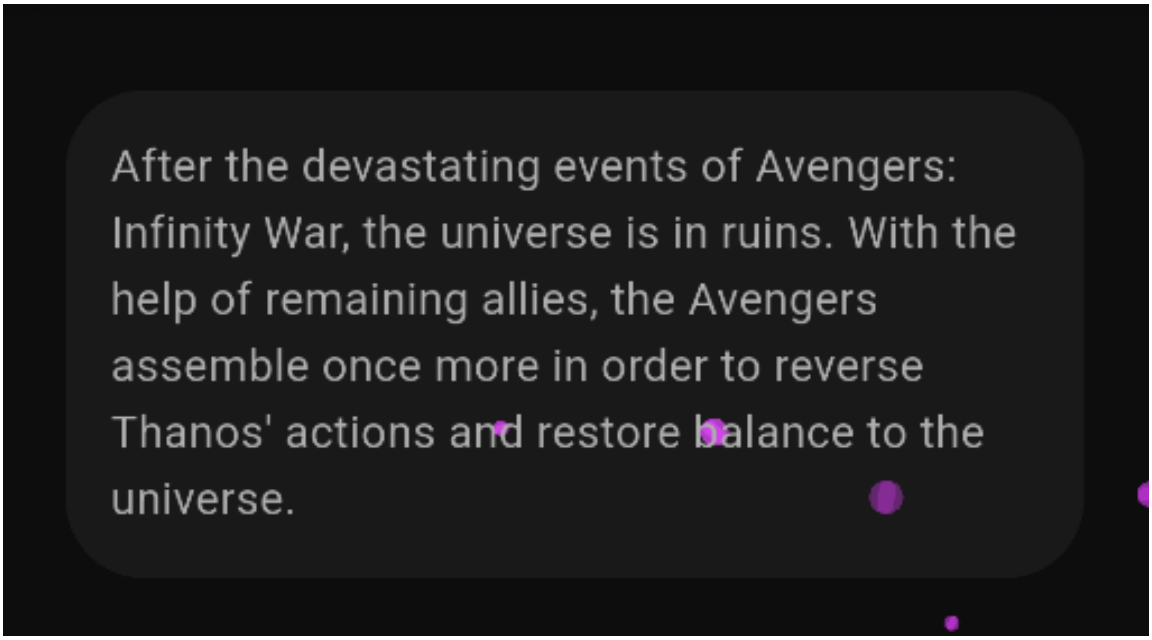

Рис. 3.7 Опис фільму

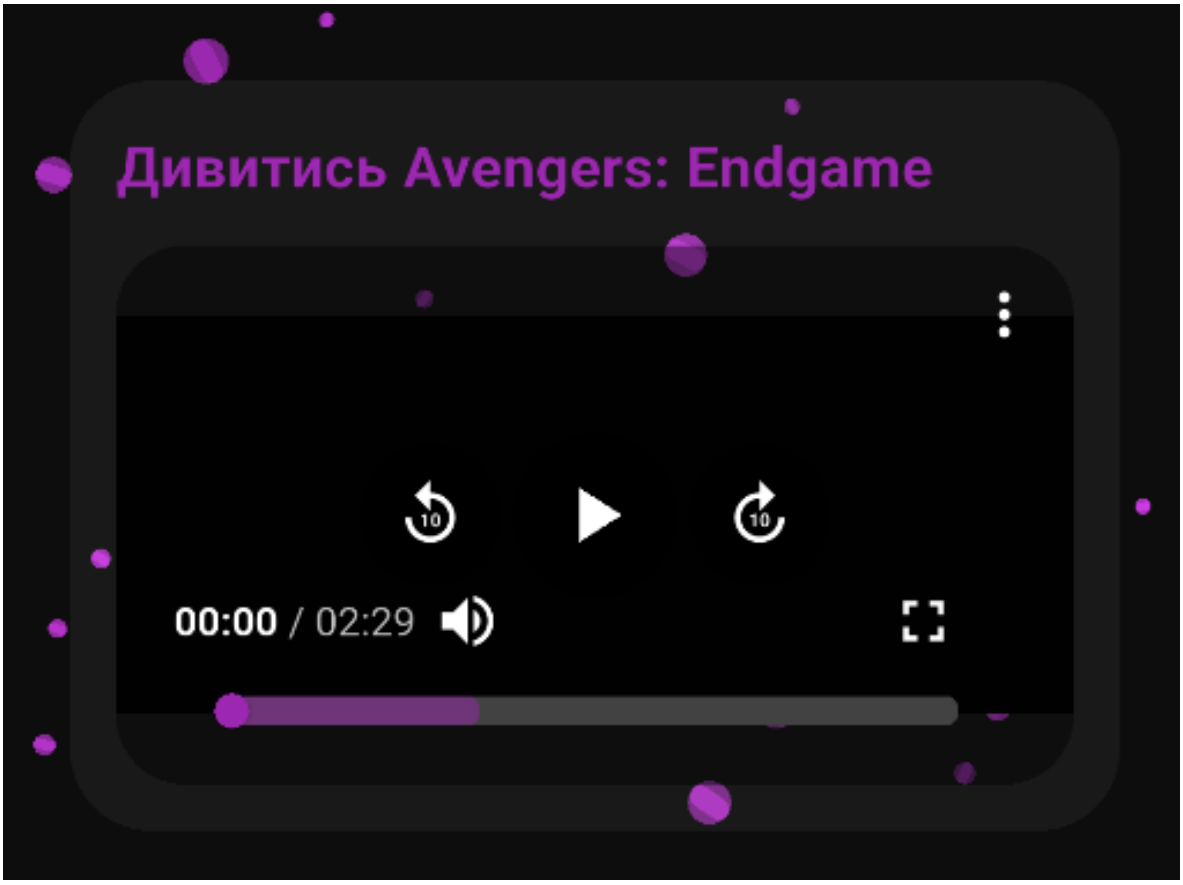

Рис. 3.8. Відеоплеєр

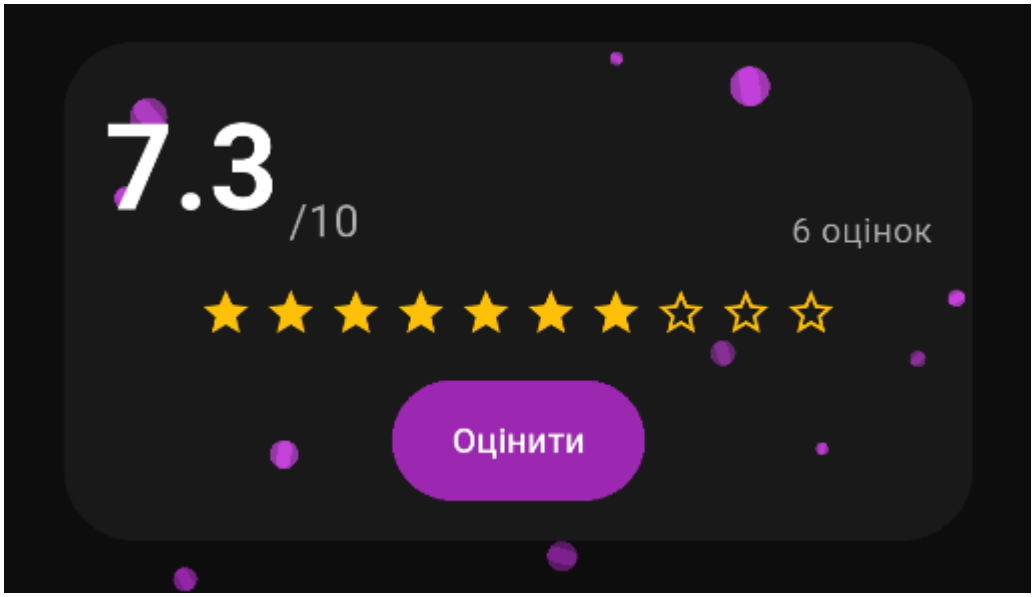

Рис. 3.9. Блок оцінки

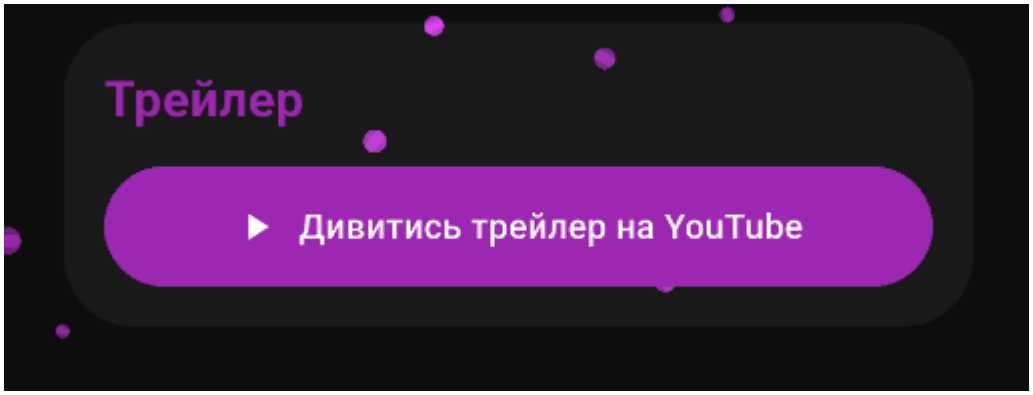

Рис. 3.10. Блок трейлера

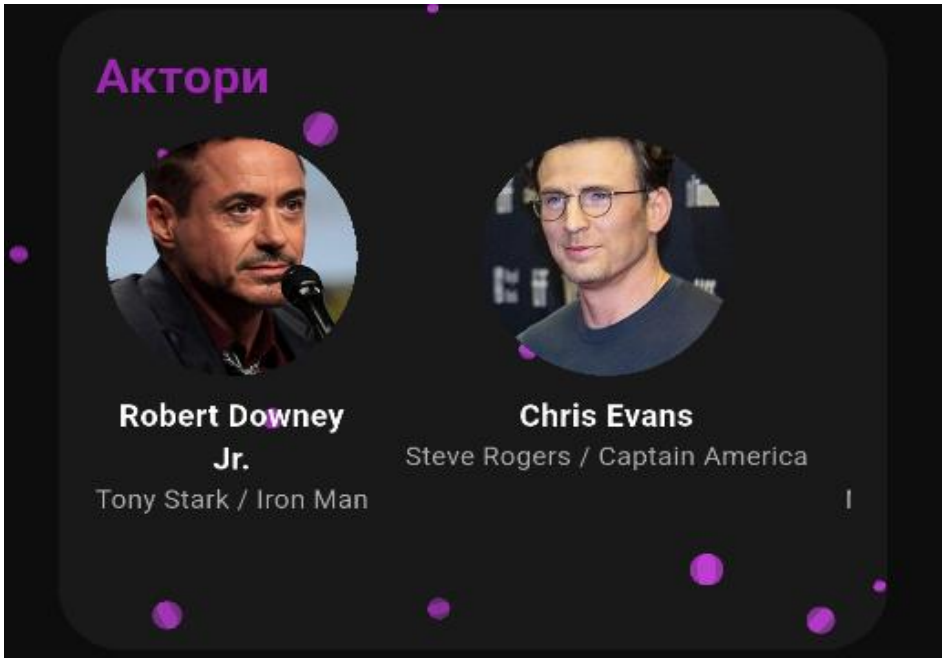

Рис. 3.11. Список акторів

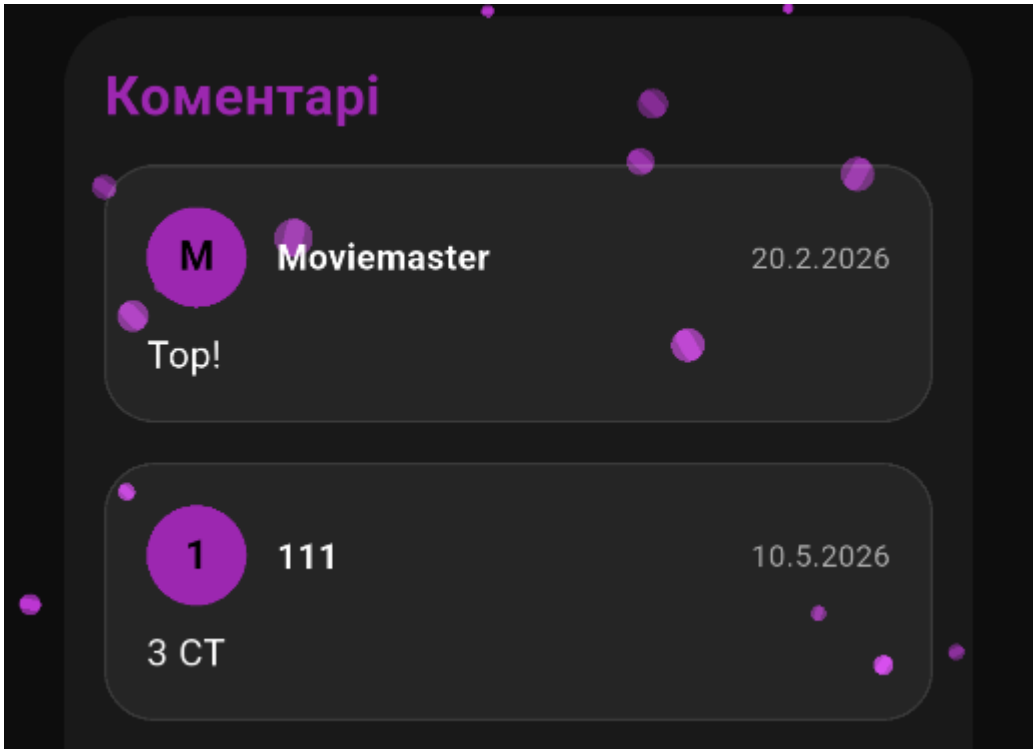

Рис. 3.12. Коментарі

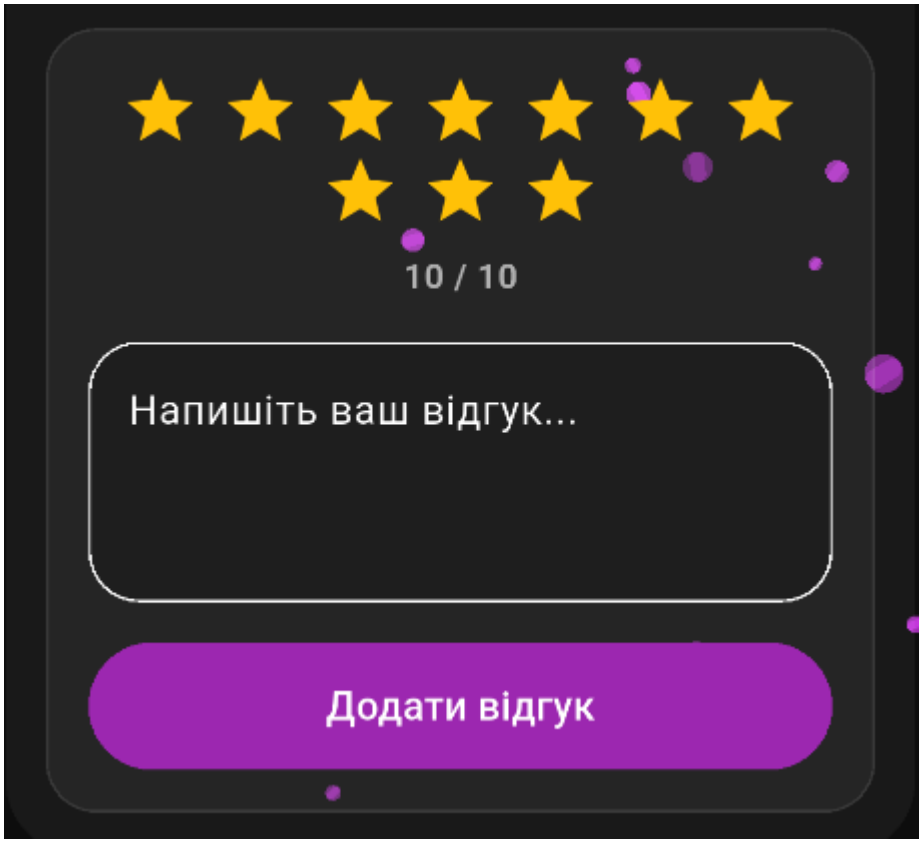

Рис. 3.13. Форма додавання нового коментаря

Оптимізація продуктивності:

- Відмовлено від вбудованого YoutubePlayer через його високу ресурсозатратність (WebView).
- Усунуто помилки Invalid constant value шляхом видалення зайвих const у місцях, де використовуються неконстантні значення (наприклад, AppTheme.primaryColor).
- Анімації налаштовано так, щоб вони не викликали перебудову всього дерева віджетів.

3.2.5.5. Сторінка актора

ActorDetailPage відображає повну інформацію про актора: фотографію, ім'я, біографію та сітку постерів фільмів з його участю. Натискання на будь-який постер

відкриває детальну сторінку відповідного фільму, що забезпечує зручну навігацію між пов'язаними об'єктами. Інтерфейс сторінки подано на рис. 3.14.

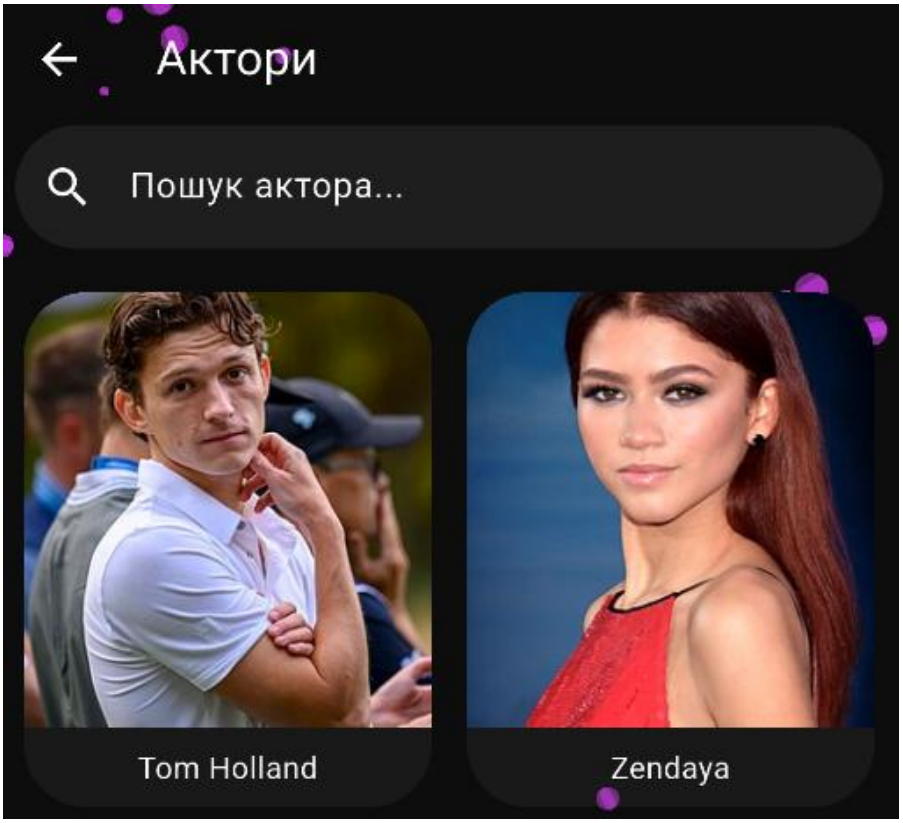

Рис. 3.14. Сторінка актора

3.2.5.6. Сторінки обраного, історії, підбірок

Сторінки обраного, історії переглядів та тематичних підбірок реалізовані за єдиним шаблоном: вони отримують дані з відповідного провайдера та відображають їх у вигляді GridView з постерами фільмів. Такий підхід забезпечує консистентний вигляд і зручність навігації по всіх розділах застосунку. Приклад реалізації інтерфейсу подано на рис. 3.15–3.17.

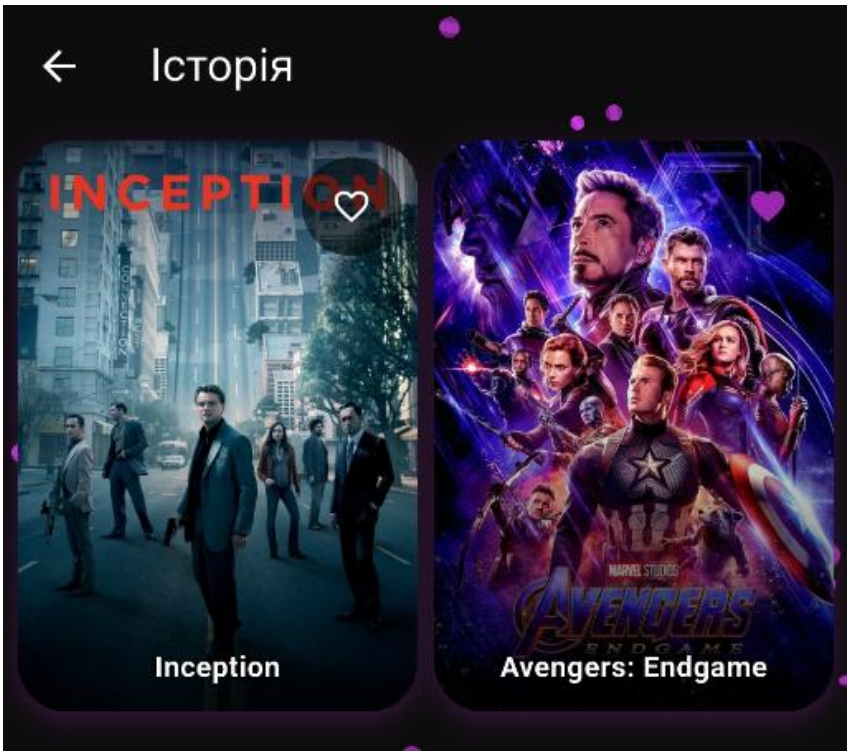

Рис. 3.15. Сторінка історії переглядів

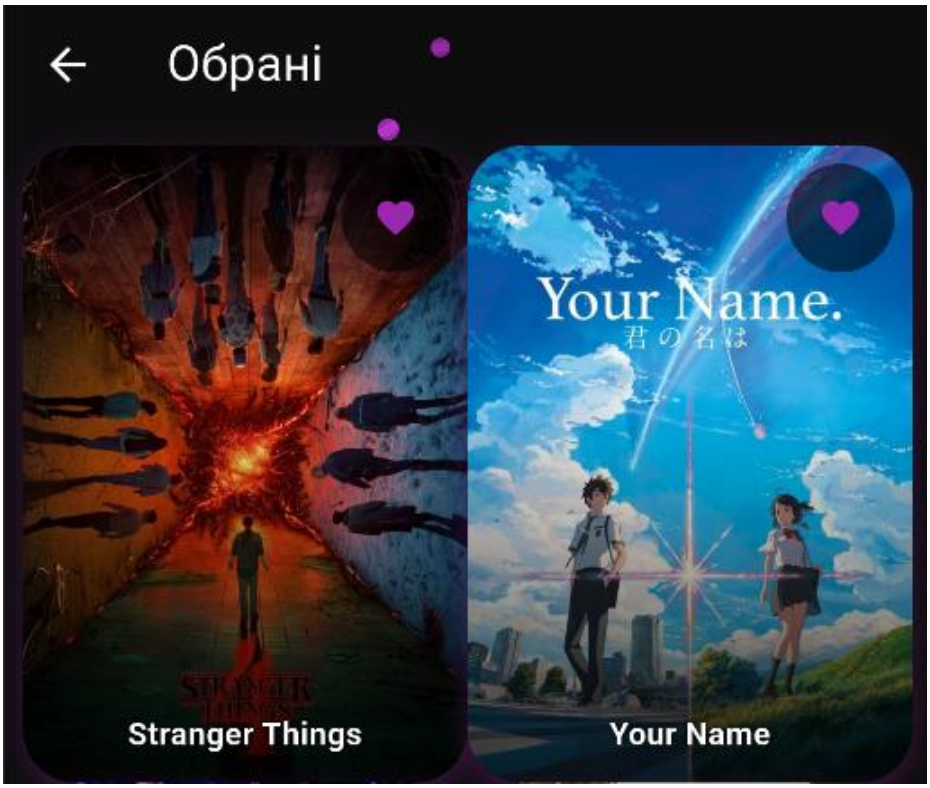

Рис. 3.16. Сторінка обраного

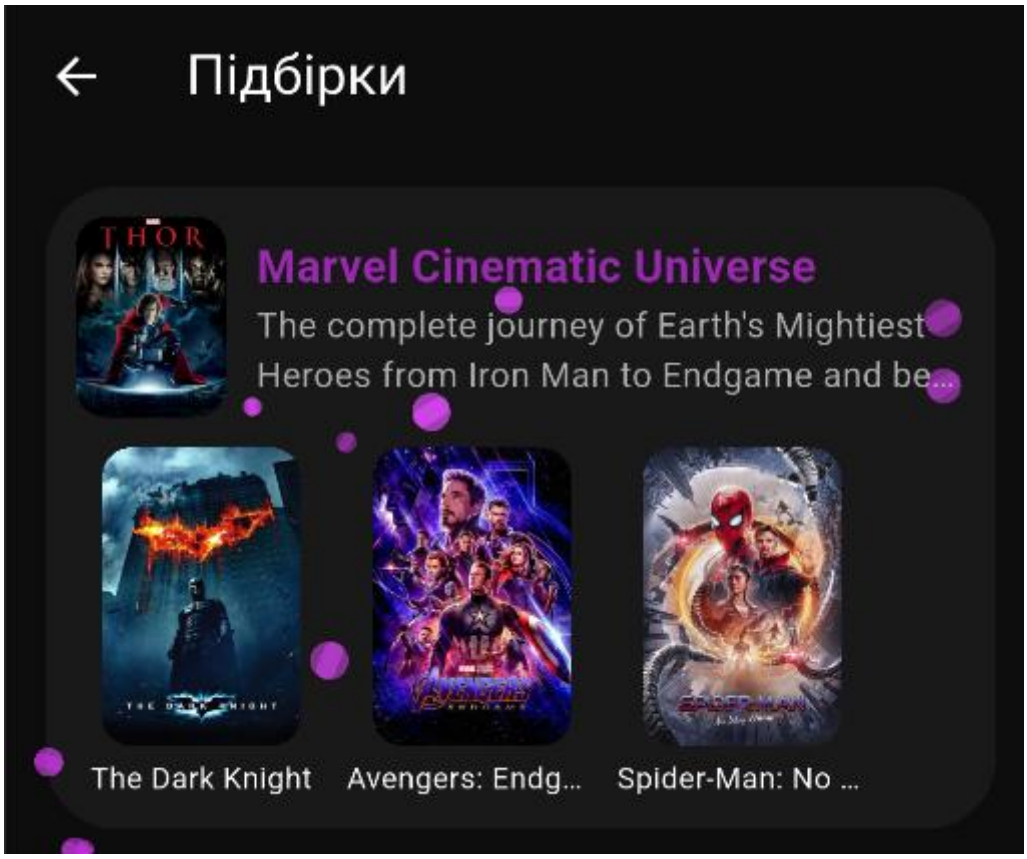

Рис. 3.17. Сторінка тематичних підбірок

3.2.6. Навігація та локалізація

Для навігації між екранами використовується **нижня панель BottomNavBar** з 7 пунктами. При натисканні на пункт викликається Navigator.pushReplacementNamed, що дозволяє уникнути накопичення маршрутів у стеку. Інтерфейс навігаційної панелі подано на рис. 3.18.

Маршрути визначені у main.dart:

```
routes: {

  '/login': (context) => const LoginPage(),
  '/home': (context) => const HomePage(),
  '/movies': (context) => const MoviesListPage(),
```

```
  '/favorites': (context) => const FavoritesPage(),
  '/history': (context) => const HistoryPage(),
  '/collections': (context) => const CollectionsPage(),
  '/actors': (context) => const ActorsListPage(),
  '/settings': (context) => const SettingsPage(),
}
```

Локалізація реалізована через flutter_localizations з генерацією класу AppLocalizations. Підтримуються українська (uk) та англійська (en) мови. Користувач може перемикати мову через BottomNavBar (збережено для сумісності) або в налаштуваннях.

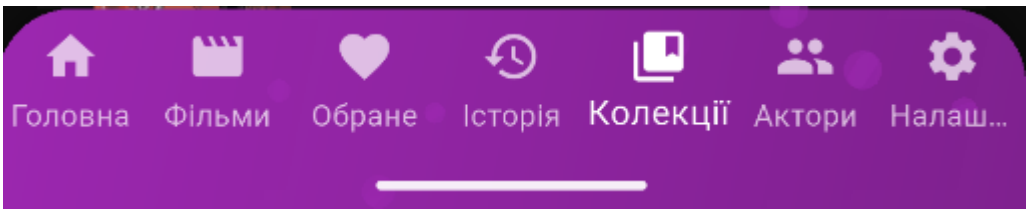

Рис. 3.18. BottomNavBar

3.2.7. Керування темою застосунку

Застосунок підтримує дві кольорові теми: «Фіолетова» (акцент #9C27B0) та «Синя» (акцент #2196F3). Перемикання теми реалізовано через colorThemeProvider (Riverpod) та зберігається у SharedPreferences для відновлення після перезапуску.

На сторінці SettingsPage користувач може обрати тему за допомогою двох круглих кольорових індикаторів. При зміні теми оновлюється глобальний AppTheme.primaryColor, і всі компоненти (кнопки, акценти, індикатори) автоматично перемальовуються. Інтерфейс керування темою подано на рис. 3.19.

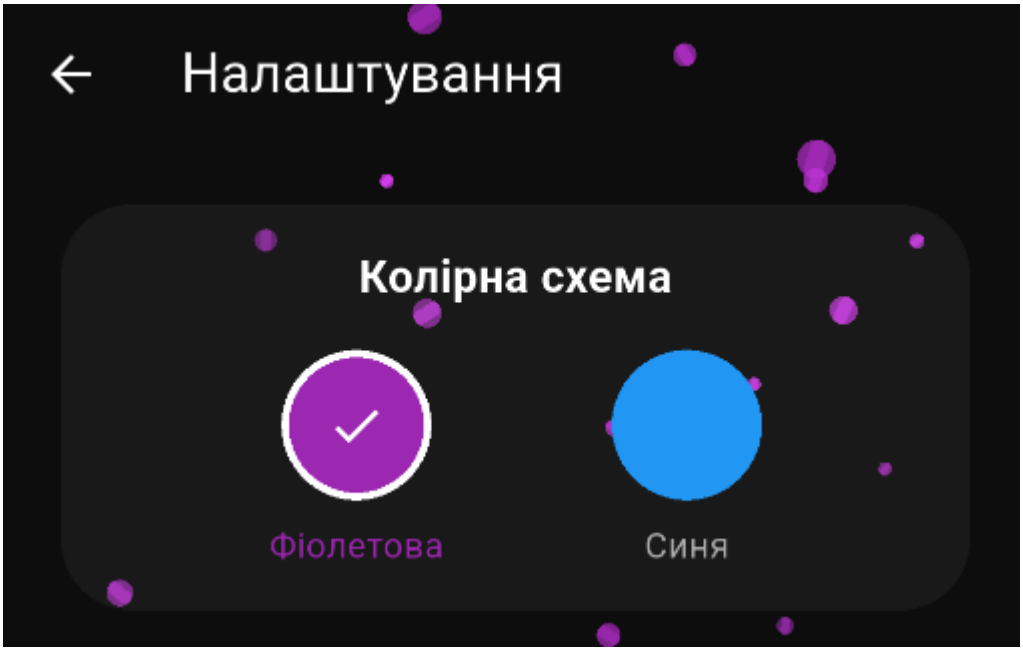

Рис. 3.19. Керування темою застосунку

3.3. Тестування та налагодження

3.3.1 Тестування API за допомогою Postman

Для тестування всіх ендпоінтів серверної частини використовувався інструмент Postman. Для кожного методу (GET, POST, PUT, DELETE) створено окремі HTTP-запити з відповідними заголовками та тілами. Тестування проводилося паралельно з розробкою за принципом «написав - перевір», що дозволяло виявляти помилки на ранніх етапах. [14]

Для захищених маршрутів у заголовки запитів додавався X-User-Id з ідентифікатором тестового користувача. Перевірялися позитивні сценарії (коректні дані) та негативні (відсутні поля, неіснуючі ID, повторна реєстрація з тим самим email). Приклад успішного виконання POST-запиту для створення нового фільму наведено на рис. 3.20.

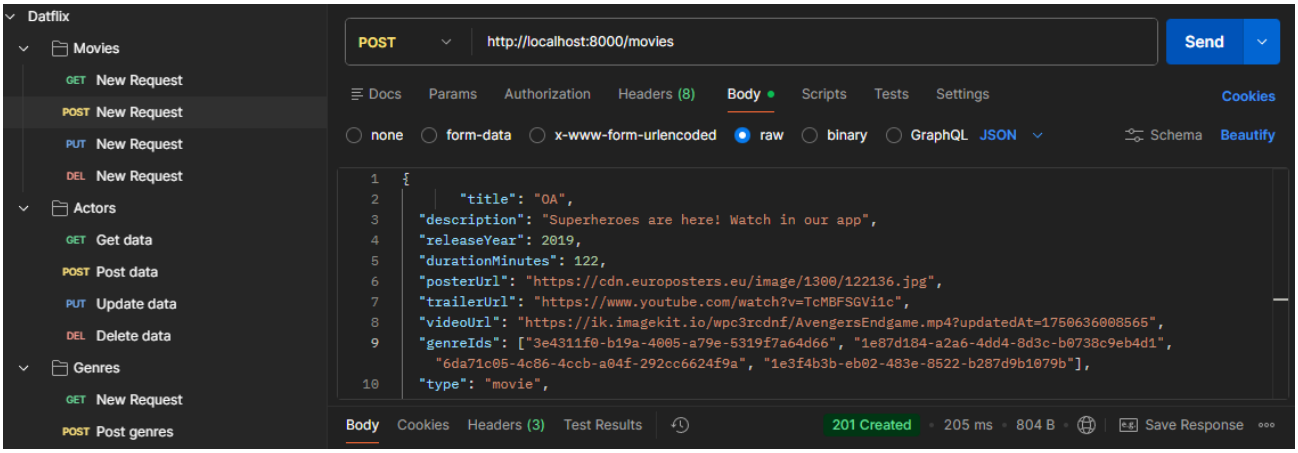

Рис. 3.20. Успішне створення нового фільму

3.3.2. Виправлення типових помилок

3.3.2.1. Проблема кодування символів в PostgreSQL

При перенесенні даних з Firestore виявилося, що символи кирилиці відображаються некоректно - замість літер з'являлися знаки питання або нечитабельні символи. Причиною була невідповідність кодування бази даних. Проблему вирішено шляхом видалення існуючої бази та створення нової з явно вказаним кодуванням UTF8 (лістинг 3.20):

Лістинг 3.20 – Створення бази даних PostgreSQL з кодуванням UTF8

```
CREATE DATABASE datflix WITH ENCODING 'UTF8'
  LC_COLLATE='uk_UA.utf8' LC_CTYPE='uk_UA.utf8'
  TEMPLATE template0;
```

3.3.2.2. Помилка 405 при використанні PUT/DELETE

Сервер повертав помилку 405 Method Not Allowed при спробі виконати PUT або DELETE запити, хоча відповідні маршрути були визначені. Причиною виявилася відсутність необхідних імпортів для цих HTTP-методів у файлі маршрутизації Ktor. Після додавання імпортів проблема зникла:

import io.ktor.server.routing.put

import io.ktor.server.routing.delete

3.3.2.3. Дублювання даних між користувачами

Після входу нового користувача у застосунку відображалися списки обраного та історія попереднього користувача. Причиною була відсутність оновлення стану провайдерів після зміни авторизованого користувача. Проблему вирішено додаванням методу reload() до favoritesProvider та historyProvider з подальшим його викликом одразу після успішної автентифікації.

3.3.2.4. Помилка серіалізації LocalDateTime

Firebase Admin SDK не міг коректно десеріалізувати об'єкти типу LocalDateTime під час міграції даних. Проблему вирішено шляхом зміни підходу до зберігання дат: замість об'єкта LocalDateTime дата тепер зберігається як рядок у ISO 8601 форматі та при необхідності конвертується на стороні клієнта.

3.3.2.5. Overflow в інтерфейсі

На детальній сторінці фільму текст біографії актора виходив за межі відведеного простору, що спричиняло помилку RenderFlex overflowed. Проблему вирішено шляхом обгортання текстового віджету в Flexible та встановлення обмеження maxLines з можливістю розгортання тексту за потреби.

3.3.2.6. Дублювання ключа при створенні коментарів

При спробі додати коментар сервер повертав помилку дублювання первинного ключа. Причиною було те, що клієнт передавав порожній рядок як значення поля id, і для всіх коментарів генерувався однаковий ключ. Виправлення полягало у тому, щоб сервер самостійно генерував UUID для нових записів (лістинг 3.21):

Лістинг 3.21 – Генерація UUID для нового коментаря

```
val newId = if (review.id.isNullOrBlank())
```

```
    UUID.randomUUID().toString() else review.id!!
```

3.3.2.7. Лаги через вбудований YouTubePlayer

Під час тестування на емуляторі Android та слабких фізичних пристроях спостерігалися значні пропуски кадрів (Skipped frames) при відкритті сторінки фільму з трейлером. Причиною було використання пакету youtube_player_flutter, який запускає важкий WebView.

Вирішення: заміна вбудованого плеєра на звичайну кнопку, яка відкриває трейлер у зовнішньому браузері через url_launcher. Це повністю усунуло лаги, оскільки браузер працює в окремому процесі.

3.3.2.8. Помилка "Invalid constant value"

При компіляції виникали помилки Invalid constant value у місцях, де використовувався const з неконстантними значеннями (наприклад, const TextStyle(color: AppTheme.primaryColor)).

Вирішення: видалення зайвих const у відповідних місцях. const залишено тільки для справжніх констант (Colors.amber, EdgeInsets.all(16) тощо).

3.3.3. Результати тестування

Після виправлення всіх виявлених помилок застосунок працює стабільно в усіх перевірених сценаріях. Усі функціональні вимоги, визначені у розділі 1.3, успішно реалізовані та перевірені. Дані різних користувачів коректно розмежовано, інтерфейс адаптивно відображається на пристроях.

3.4. Керівництво користувача

3.4.1. Запуск застосунку

Для запуску клієнтської частини необхідно мати встановлений Flutter SDK та підключений пристрій (емулятор або фізичний телефон). Виконайте команду `flutter run` у корені проєкту `datflix_frontend`. Серверна частина запускається командою `./gradlew run` з папки `datflix-backend`. Перед першим запуском переконайтеся, що PostgreSQL запущено, а базу даних створено згідно з міграціями.

3.4.2. Реєстрація та вхід

При першому запуску відкривається сторінка автентифікації (рис. 3.2). Користувач може: - перейти до реєстрації, натиснувши «Немає акаунту? Зареєструватись»; - заповнити поля email, пароль (і, за потреби, ім’я користувача) та натиснути «Зареєструватись» або «Увійти». Після успішного входу програма автоматично переходить на головний екран.

3.4.3. Головний екран (HomePage)

На головному екрані (рис. 3.4) розташовані горизонтальні каруселі:

- «Популярні фільми» – перші 10 фільмів з позначкою `isTrending`;

- каруселі за кожним жанром (завантажуються з API).

Натискання на картку фільму відкриває детальну сторінку. У нижній частині екрана розташована навігаційна панель (BottomNavBar) для швидкого переходу.

3.4.4. Каталог фільмів (MoviesListPage)

Екран (рис. 3.5) показує всі фільми у вигляді сітки. Доступні:

- пошук за назвою через поле введення (кнопка лупи);

- фільтрація за жанром, типом контенту, роком випуску (кнопка фільтрів).

Після застосування фільтрів список оновлюється автоматично.

3.4.5. Детальна сторінка фільму (MovieDetailPage)

Сторінка (рис. 3.6–3.13) містить:

- постер з накладанням назви, року, тривалості, типу (чипи);

- опис у скляній картці;

- відеоплеєр (якщо фільм має `videoUrl`) – підтримує повноекранний режим та зміну швидкості;

- блок оцінки – середній бал, кількість оцінок, 10 зірок, кнопка «Оцінити» (доступна тільки авторизованим). Натискання відкриває діалог з 10 зірками;

- блок трейлера – кнопка, яка відкриває YouTube у зовнішньому браузері (для уникнення лагів);

- список акторів (горизонтальний) – фото, ім’я, роль. Натискання веде на сторінку актора;

- коментарі – список відгуків (аватар, ім’я, дата, текст) та форма додавання нового коментаря з 10-зірковою оцінкою.

3.4.6. Сторінка актора (ActorDetailPage)

Показує (рис. 3.14): фотографію, ім’я, біографію та сітку фільмів, у яких знімався актор. Натискання на будь-який фільм переводить на його детальну сторінку.

3.4.7. Сторінки «Обране», «Історія», «Підбірки»

Ці сторінки (рис. 3.15–3.17) побудовані аналогічно: відображають сітку фільмів, отриману з відповідних провайдерів (`favoritesProvider`, `historyProvider`, `collectionsProvider`). Додавання до обраного/історії відбувається автоматично при перегляді фільму або натисканні кнопки «серце».

3.4.8. Налаштування (SettingsPage)

На сторінці (рис. 3.19) можна змінити кольорову тему застосунку:

- Фіолетова (акцент #9C27B0);

- Синя (акцент #2196F3).

Вибір зберігається в SharedPreferences і відновлюється після перезапуску.

3.4.9. Вихід із системи

Для виходу потрібно натиснути кнопку «Вийти» в налаштуваннях. Після підтвердження дії всі локальні дані авторизації очищаються, і користувач перенаправляється на сторінку входу.

3.5. Вимоги до технічного та програмного забезпечення

3.5.1. Апаратні вимоги для запуску клієнта

Для коректної роботи клієнтського застосунку необхідно забезпечити відповідність мінімальним апаратним вимогам.

Для пристроїв на базі Android підтримується операційна система Android 6.0 (API 23) або новіше. Мінімальний обсяг оперативної пам’яті становить 2 ГБ. Рекомендована роздільна здатність екрана — 720×1080 пікселів або більше.

Для пристроїв на базі iOS підтримується операційна система iOS 13.0 або новіше. Мінімально підтримуваним пристроєм є iPhone SE першого покоління або новіші моделі.

Для тестування застосунку можуть використовуватися емулятори Android Studio AVD (архітектура x86_64) із виділеними 2 ГБ оперативної пам’яті або iOS Simulator.

3.5.2. Програмне забезпечення для розробки та розгортання

Для розробки, запуску та розгортання системи використовуються такі програмні компоненти:

Таблиця 3.1

Таблиця програмних компонентів

Компонент	Технологія	Версія	Призначення
Операційна система	Windows / macOS / Linux	Windows 10/11, macOS 11+, Ubuntu 20.04+	Середовище виконання

Продовження табл. 3.1

Компонент	Технологія	Версія	Призначення
Серверна платформа	JVM (OpenJDK)	21	Запуск Ktor-сервера
База даних	PostgreSQL	18	Зберігання даних
ORM	Exposed	0.41.1	SQL DSL для Kotlin
Система збірки сервера	Gradle	8.5	Управління залежностями
Клієнтське середовище	Flutter SDK	3.19+	Збірка та запуск застосунку
Мова програмування клієнта	Dart	3.3+	Вихідна мова клієнтської частини
Менеджер стану	Riverpod	2.6.1	Реактивне управління станом
Менеджер залежностей клієнта	pub.dev	—	Керування пакетами згідно з pubspec.yaml

3.5.3. Мережеві вимоги

Під час локального запуску серверна частина повинна бути доступною за адресою http://10.0.2.2:8000 для Android-емулятора або http://localhost:8000 для реального пристрою чи вебверсії застосунку.

Для віддаленого розгортання необхідна наявність статичної IP-адреси або доменного імені з відкритим портом 8000, а також налаштованого CORS для забезпечення доступу клієнтського застосунку до серверних ресурсів.

Мінімальна рекомендована швидкість інтернет-з’єднання становить 1 Мбіт/с для відтворення потокового відео. Для перегляду відео у HD-якості рекомендовано використовувати з’єднання зі швидкістю не менше 5 Мбіт/с.

3.5.4. Вимоги до безпеки

Паролі користувачів зберігаються виключно у хешованому вигляді із застосуванням алгоритму BCrypt та параметром cost factor 10.

Передача конфіденційних даних між клієнтською та серверною частинами в production-середовищі повинна здійснюватися через захищений протокол HTTPS.

Для доступу до захищених маршрутів використовується заголовок X-User-Id, який не повинен передаватися через незахищені канали зв’язку.

3.5.5. Рекомендації щодо продуктивності

Для підвищення продуктивності бази даних при роботі з таблицею movies, яка містить понад 10 000 записів, рекомендується створити індекси за полями type та releaseYear, а також повнотекстовий індекс tsvector для оптимізації пошуку за назвою.

Клієнтський застосунок оптимізований для забезпечення плавності анімацій. Для цього використовуються компоненти RepaintBoundary, а також const-конструктори у всіх можливих випадках.

Відтворення відеотрейлерів здійснюється через браузер, що дозволяє уникнути зниження FPS та перевантаження застосунку при використанні WebView.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було спроектовано клієнт-серверний застосунок "Datflix" - власний аналог стрімінгового сервісу для перегляду фільмів. Виконана робота охоплює повний цикл розробки програмного продукту - від аналізу вимог до налагодження готового застосунку. Основні результати роботи:

1. Проведено аналіз предметної області, вивчено існуючі рішення (Netflix, Megogo, Sweet.tv), сформульовано функціональні та нефункціональні вимоги, обрано оптимальний технологічний стек.
2. Спроектовано трирівневу архітектуру системи, розроблено ER-діаграму бази даних з восьми сутностей та специфікацію REST API з шістнадцяти ендпоінтів.
3. Реалізовано серверну частину на Kotlin з використанням фреймворку Ktor та ORM Exposed. Забезпечено автентифікацію з хешуванням паролів BCrypt, фільтрацію контенту за жанром/типом/роком, роботу з акторами, підбірками та коментарями.
4. Розроблено клієнтську частину на Flutter з реактивним управлінням станом через Riverpod. Реалізовано адаптивний інтерфейс, локалізацію (українська/англійська), інтеграцію з відеоплеєром chewie.
5. Виконано міграцію даних з Firestore до PostgreSQL, що дозволило отримати повний контроль над даними та відмовитися від залежності від зовнішнього хмарного сервісу.
6. Проведено комплексне тестування через Postman, виявлено та усунено вісім характерних помилок: проблему кодування, помилку 405, дублювання даних між користувачами, помилку серіалізації, overflow та дублювання ключів, лаги через вбудований YouTubePlayer та помилку "Invalid constant value".

7. Реалізовано сучасний темний дизайн з анімованим фоном (PetalBackground), скляними картками, нижньою навігацією (BottomNavBar) та можливістю перемикання кольорової теми (фіолетова/синя) через SettingsPage.
8. Покращено систему оцінювання – замість 5-зіркової шкали впроваджено 10-бальну, що дає змогу точніше виражати думку користувача. Оцінка обчислюється як середнє арифметичне всіх коментарів.
9. Оптимізовано продуктивність – усунено лаги, спричинені вбудованим YoutubePlayer (замінено на кнопку з відкриттям у браузері), виправлено помилки Invalid constant value, зменшено кількість перебудов через розбиття сторінок на дрібні віджети.

Практична цінність роботи полягає у створенні повнофункціонального застосунку з реальною архітектурою, який може бути використаний як основа для подальшого розвитку. Набутий досвід охоплює всі ключові аспекти сучасної розробки: проектування API, роботу з реляційними базами даних, кросплатформну мобільну розробку та налагодження розподілених систем.

Пропозиції щодо вдосконалення системи:

- впровадити систему ролей (адміністратор, модератор) для керування контентом;
- додати кешування даних на клієнті для роботи в офлайн-режимі;
- оптимізувати SQL-запити шляхом додавання індексів та складніших JOIN-конструкцій;
- реалізувати відновлення пароля через email-підтвердження;
- налаштувати CI/CD pipeline для автоматичного тестування та розгортання;
- розробити рекомендаційну систему на основі історії переглядів та оцінок.

Отриманий досвід є цінним внеском у формування практичних навичок як фахівця з комп'ютерних наук та стане міцною основою для подальшої професійної діяльності у сфері розробки програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] JetBrains. Kotlin Official Documentation. – JetBrains, 2025. – URL: https://kotlinlang.org/docs/home.html (дата звернення: 27.01.2026).

[2] JetBrains. Ktor Framework for Connected Systems. – JetBrains, 2025. – URL: https://ktor.io/docs/welcome.html (дата звернення: 28.01.2026).

[3] PostgreSQL Global Development Group. PostgreSQL 18 Documentation. – PostgreSQL, 2025. – URL: https://www.postgresql.org/docs/ (дата звернення: 30.01.2026).

[4] JetBrains. Exposed – Kotlin SQL Framework. – GitHub, 2025. – URL: https://github.com/JetBrains/Exposed (дата звернення: 01.02.2026).

[5] Wooldridge B. HikariCP – Ultimate JDBC Connection Pool. – GitHub, 2025. – URL: https://github.com/brettwooldridge/HikariCP (дата звернення: 02.02.2026).

[6] Favre P. BCrypt – Java Implementation of bcrypt Password Hash Function. – GitHub, 2025. – URL: https://github.com/patrickfav/bcrypt (дата звернення: 04.02.2026).

[7] Flutter Team. Flutter Official Documentation. – Google, 2025. – URL: https://docs.flutter.dev/ (дата звернення: 09.02.2026).

[8] Riverpod Authors. Riverpod – Simple, Safe and Reactive State Management. – Riverpod.dev, 2025. – URL: https://riverpod.dev/ (дата звернення: 11.02.2026).

[9] Flutter Community. video_player – Flutter Package for Video Playback. – pub.dev, 2025. – URL: https://pub.dev/packages/video_player (дата звернення: 13.02.2026).

[10] Flutter Community. chewie – Flutter Video Player with Controls. – pub.dev, 2025. – URL: https://pub.dev/packages/chewie (дата звернення: 14.02.2026).

[11] Flutter Community. url_launcher – Flutter Package for Launching URLs. – pub.dev, 2025. – URL: https://pub.dev/packages/url_launcher (дата звернення: 15.02.2026).

[12] Google Fonts Team. Google Fonts – Flutter Package. – pub.dev, 2025. – URL: https://pub.dev/packages/google_fonts (дата звернення: 16.02.2026).

[13] Flutter Community. shared_preferences – Flutter Package for Persistent Storage. – pub.dev, 2025. – URL: https://pub.dev/packages/shared_preferences (дата звернення: 17.02.2026).

[14] Postman Inc. Postman API Platform. – Postman, 2025. – URL: https://www.postman.com/ (дата звернення: 19.02.2026).
[15] pgAdmin Development Team. pgAdmin – PostgreSQL Tools. – pgAdmin.org, 2025. – URL: https://www.pgadmin.org/ (дата звернення: 20.02.2026).
[16] Statista Research Department. Video Streaming (SVoD) – Worldwide. – Statista, 2024. – URL: https://www.statista.com/outlook/dmo/digital-media/video-streaming/worldwide (дата звернення: 28.05.2026).
[17] Grand View Research. Video Streaming Market Size Report, 2024–2030. – Grand View Research, 2024. – URL: https://www.grandviewresearch.com/industry-analysis/video-streaming-market (дата звернення: 28.05.2026).
[18] Massé M. REST API Design Rulebook. – O'Reilly Media, 2011. – 118 p.
[19] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley, 1994. – 395 p.
[20] Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Prentice Hall, 2017. – 432 p.
[21] Fowler M. Patterns of Enterprise Application Architecture. – Addison-Wesley, 2002. – 533 p.

ДОДАТКИ

Додаток А

Структура проєкту серверної частини Datflix

```
datflix-backend/
├── build.gradle.kts        # Gradle-конфігурація та залежності
├── src/
│   └── main/
│       ├── kotlin/
│       │   ├── com.datflix/
│       │   │   ├── Application.kt          # Точка входу Ktor-сервера
│       │   │   ├── models/                 # Моделі даних
│       │   │   │   ├── User.kt
│       │   │   │   ├── Movie.kt
│       │   │   │   ├── Actor.kt
│       │   │   │   ├── Genre.kt
│       │   │   │   ├── Review.kt
│       │   │   │   ├── Collection.kt
│       │   │   │   └── Cast.kt
│       │   │   ├── repositories/           # Шар доступу до даних
│       │   │   │   ├── Repository.kt
│       │   │   │   ├── UserRepository.kt
│       │   │   │   ├── MovieRepository.kt
│       │   │   │   ├── ActorRepository.kt
│       │   │   │   ├── GenreRepository.kt
│       │   │   │   ├── ReviewRepository.kt
│       │   │   │   ├── CollectionRepository.kt
│       │   │   │   └── CastRepository.kt
```

Продовження додатку А

```
│   │   │   ├── services/              # Бізнес-логіка
│   │   │   │   ├── AuthService.kt
│   │   │   │   ├── MovieService.kt
│   │   │   │   ├── ActorService.kt
│   │   │   │   └── ReviewService.kt
│   │   │   ├── routes/                # REST API маршрути
│   │   │   │   ├── AuthRoutes.kt
│   │   │   │   ├── MovieRoutes.kt
│   │   │   │   ├── ActorRoutes.kt
│   │   │   │   ├── UserRoutes.kt
│   │   │   │   ├── ReviewRoutes.kt
│   │   │   │   └── CollectionRoutes.kt
│   │   │   ├── database/              # Конфігурація БД
│   │   │   │   ├── DatabaseFactory.kt
│   │   │   │   └── Tables.kt          # Exposed-таблиці
│   │   │   └── utils/
│   │   │       └── Migration.kt       # Скрипт міграції з Firestore
│   └── resources/
│       └── application.conf           # Налаштування Ktor
```

Додаток Б

Основні маршрути API Datflix

Маршрут	Метод	Призначення	Доступ
/auth/register	POST	Реєстрація нового користувача	Публічний
/auth/login	POST	Вхід користувача	Публічний
/movies	GET	Список фільмів з фільтрацією	Публічний
/movies/{id}	GET	Деталі фільму за ID	Публічний
/actors	GET	Список усіх акторів	Публічний
/actors/{id}/with-movies	GET	Актор з його фільмографією	Публічний
/cast/movie/{movieId}	GET	Актори конкретного фільму	Публічний
/reviews/movie/{movieId}	GET	Коментарі до фільму	Публічний
/reviews	POST	Додавання коментаря та оцінки	X-User-Id
/users/favorites/{movieId}	POST	Додати фільм до обраного	X-User-Id
/users/favorites/{movieId}	DELETE	Видалити фільм з обраного	X-User-Id
/users/favorites	GET	Список обраних фільмів	X-User-Id
/users/history/{movieId}	POST	Додати фільм до історії	X-User-Id
/users/history	GET	Історія переглядів	X-User-Id
/collections	GET	Список тематичних підбірок	Публічний
/collections/{id}/movies	GET	Фільми з підбірки	Публічний

Додаток В

Структура проєкту клієнтської частини (Flutter)

```
datflix-client/
├── pubspec.yaml          # Залежності Flutter-проєкту
└── lib/
    ├── main.dart         # Точка входу застосунку
    ├── models/           # Моделі даних на клієнті
    │   ├── movie.dart
    │   ├── actor.dart
    │   ├── genre.dart
    │   ├── review.dart
    │   └── collection.dart
    ├── services/         # Класи для роботи з API
    │   ├── auth_service.dart
    │   ├── movie_service.dart
    │   ├── actor_service.dart
    │   └── review_service.dart
    ├── providers/        # Riverpod-провайдери
    │   ├── movies_provider.dart
    │   ├── favorites_provider.dart
    │   ├── history_provider.dart
    │   ├── actors_provider.dart
    │   ├── genres_provider.dart
    │   ├── filters_provider.dart
    │   └── color_theme_provider.dart
    ├── screens/          # Екрани застосунку
    │   ├── login_page.dart
    │   ├── home_page.dart
```

Продовження додатку В

```
│   ├── movies_list_page.dart
│   ├── movie_detail_page.dart
│   ├── actor_detail_page.dart
│   ├── actors_list_page.dart
│   ├── favorites_page.dart
│   ├── history_page.dart
│   ├── collections_page.dart
│   └── settings_page.dart
├── widgets/            # Перевикористовувані компоненти
│   ├── movie_card.dart
│   ├── bottom_nav_bar.dart
│   ├── petal_background.dart
│   ├── rating_dialog.dart
│   └── filters_dialog.dart
├── theme/
│   └── app_theme.dart   # Налаштування теми та дизайну
└── l10n/               # Файли локалізації (uk/en)
    ├── app_en.arb
    └── app_uk.arb
```

Додаток Г

Основні команди для запуску проєкту

Команда	Призначення
./gradlew run	Запуск серверної частини (Ktor)
./gradlew build	Збірка серверної частини
flutter run	Запуск клієнтської частини на емуляторі/пристрої
flutter build apk	Збірка APK для Android
flutter test	Запуск тестів Flutter-застосунку
psql -U postgres -d datflix	Підключення до бази даних PostgreSQL
flutter pub get	Встановлення залежностей Flutter
./gradlew clean	Очищення попередніх збірок Gradle

Додаток Д

Структура таблиць бази даних PostgreSQL

Таблиця 2.1

Структура таблиці `users`

Назва поля	Тип	Опис
id	VARCHAR(255)	Первинний ключ, UUID
email	VARCHAR(255)	Email користувача, унікальний
username	VARCHAR(255)	Ім'я користувача
passwordHash	VARCHAR(255)	Хеш пароля (BCrypt)
avatarUrl	VARCHAR(512)	Посилання на аватар
watchlist	TEXT	JSON-список ID обраних фільмів
watchHistory	TEXT	JSON-список ID переглянутих фільмів
registrationDate	TIMESTAMP	Дата реєстрації

Таблиця 2.2

Структура таблиці `movies`

Назва поля	Тип	Опис
id	VARCHAR(255)	Первинний ключ, UUID

Продовження додатку Д

Продовження табл. 2.2

title	VARCHAR(255)	Назва фільму
description	TEXT	Опис
releaseYear	INTEGER	Рік випуску
durationMinutes	INTEGER	Тривалість у хвилинах
posterUrl	VARCHAR(512)	Постер
trailerUrl	VARCHAR(512)	Трейлер (YouTube)
videoUrl	VARCHAR(512)	Посилання на відеофайл
type	VARCHAR(50)	Тип (movie, series, anime, cartoon)
averageRating	REAL	Середня оцінка
isTrending	BOOLEAN	Чи є популярним

Таблиця 2.3

Структура таблиці `cast`

Назва поля	**Тип**	**Опис**
Id	VARCHAR(255)	Первинний ключ, UUID
actorId	VARCHAR(255)	Зовнішній ключ до actors.id

Продовження додатку Д

Продовження табл. 2.3

movieId	VARCHAR(255)	Зовнішній ключ до movies.id
characterName	VARCHAR(255)	Ім'я персонажа
isMainRole	BOOLEAN	Чи є головною роллю
orderCredit	INTEGER	Порядок у титрах

Додаток Е

Ілюстративні матеріали, використані для підтвердження реалізації

Позначення	Матеріал	Місце подання
Рис. 1.1.	Діаграма варіантів використання системи «Домашній кінотеатр»	Розділ 1
Рис. 2.1.	Діаграма розгортання системи	Розділ 2
Рис. 2.2.	ER-діаграма бази даних	Розділ 2
Рис. 2.3.	UML-діаграма класів системи	Розділ 2
Рис. 2.4.	Діаграма шарів архітектури	Розділ 2
Рис. 3.1.	Структура проєкту в Android Studio	Розділ 3
Рис. 3.2.	Сторінка входу	Розділ 3
Рис. 3.3.	Сторінка реєстрації	Розділ 3
Рис. 3.4.	Головний екран застосунку	Розділ 3
Рис. 3.5.	Екран списку фільмів з фільтрацією	Розділ 3
Рис. 3.6.	Постер з градієнтом	Розділ 3
Рис. 3.7.	Опис фільму	Розділ 3
Рис. 3.8.	Відеоплеєр	Розділ 3
Рис. 3.9.	Блок оцінки	Розділ 3
Рис. 3.10.	Блок трейлера	Розділ 3
Рис. 3.11.	Список акторів	Розділ 3
Рис. 3.12.	Коментарі	Розділ 3
Рис. 3.13.	Форма додавання нового коментаря	Розділ 3
Рис. 3.14.	Сторінка актора	Розділ 3

Продовження додатку Е

Позначення	Матеріал	Місце подання
Рис. 3.15.-3.17.	Сторінки історії, обраного, підбірок	Розділ 3
Рис. 3.18.	BottomNavBar	Розділ 3
Рис. 3.19.	Керування темою застосунку	Розділ 3
Рис. 3.20.	Успішне створення нового фільму (Postman)	Розділ 3

Додаток Ж

Ключові файли програмної реалізації

Серверна частина:

Файл	Функціональне призначення
Application.kt	Налаштування Ktor-сервера, CORS, ContentNegotiation
DatabaseFactory.kt	Підключення до PostgreSQL через HikariCP
AuthService.kt	Реєстрація, вхід, хешування паролів BCrypt
UserRepository.kt	CRUD-операції з користувачами, робота з watchlist/watchHistory
MovieRepository.kt	Фільтрація фільмів за жанром, типом, роком; пошук за назвою
ActorRepository.kt	Отримання акторів та їх фільмографії
ReviewRepository.kt	Додавання/отримання коментарів, обчислення середньої оцінки
Migration.kt	Скрипт міграції даних з Firestore до PostgreSQL

Клієнтська частина:

Файл	Функціональне призначення
main.dart	Точка входу, налаштування маршрутизації, ProviderScope

Продовження додатку Ж

Файл	Функціональне призначення
movies_provider.dart	StateNotifier для списку фільмів та фільтрів
favorites_provider.dart	Управління списком обраних фільмів
history_provider.dart	Управління історією переглядів
filters_provider.dart	Синхронізація фільтрів (жанр, тип, рік)
movie_detail_page.dart	Детальна сторінка фільму з плеєром, акторами, коментарями
login_page.dart	Екран автентифікації (вхід/реєстрація)
bottom_nav_bar.dart	Нижня навігаційна панель з 7 пунктами
petal_background.dart	Анімований фон з частинками-пелюстками
app_theme.dart	Налаштування темної теми, кольорові акценти

Додаток Й

Використані технології та бібліотеки

Серверна частина:

- **Kotlin** 1.9.22 — основна мова програмування
- **Ktor** 2.3.7 — асинхронний фреймворк для REST API
- **PostgreSQL** 18 — реляційна база даних
- **Exposed** 0.41.1 — ORM-бібліотека для Kotlin
- **HikariCP** 5.0.1 — пул підключень до БД
- **BCrypt** 0.9.0 — хешування паролів

Клієнтська частина:

- **Flutter** 3.x — кросплатформний фреймворк
- **Riverpod** 2.3.2 — управління станом
- **http** 0.13.6 — мережеві запити
- **video_player** 2.5.1 — відтворення відео
- **chewie** 1.5.0 — обгортка для video_player з контролами
- **url_launcher** 6.2.6 — відкриття посилань
- **google_fonts** 4.0.3 — кастомні шрифти (Metropolis, Inter)
- **shared_preferences** 2.2.0 — локальне збереження даних
- **flutter_localizations** — підтримка локалізації (uk/en)

Інструменти розробки:

- IntelliJ IDEA — розробка серверної частини
- Android Studio — розробка Flutter-застосунку
- Postman — тестування API
- pgAdmin — адміністрування PostgreSQL
- Git — контроль версій