МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

Желудченка Дмитра Олександровича
(прізвище, ім'я, по батькові здобувача)

на тему «Розробка програмного забезпечення для збереження та спільного використання списків фільмів»
(повна назва теми)

за матеріалами праць провідних спеціалістів з розробки ПЗ та проектування БД
(повна назва бази дослідження)

науковий керівник к.т.н., доцент _____ Хоцкіна В.Б.
(наук. ступінь, вчене звання) *(підпис)* *(прізвище, ініціали)*

Робота допущена до захисту в ЕК
Протокол засідання кафедри
від 11.06.2025 р. № 12
Завідувач кафедри _____
(підпис)
д.т.н., професор Зеленський О.С.
Наук. ступень, вчене звання *Ініціали, прізвище*

Кривий Ріг – 2025

ЗГОДА здобувача вищої освіти

Державного університету економіки і технологій про перевірку кваліфікаційної роботи на прояви академічного плагіату та розміщення в Репозитарії Університету

Я, Желудченко Дмитро Олександрович, підтримую політику Державного університету економіки і технологій з академічної доброчесності і відкритого доступу.

Засвідчую, що кваліфікаційна бакалаврська робота «Розробка програмного забезпечення для збереження та спільного використання списків фільмів» виконана самостійно та не містить академічного плагіату. Я не надавав і не одержував недозволену допомогу під час підготовки цієї роботи. Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело.

Із чинним Положенням про запобігання та виявлення академічного плагіату в роботах здобувачів вищої освіти Державного університету економіки і технологій ознайомлений. Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі порушення норм академічної доброчесності робота не допускається до захисту або оцінюється незадовільно.

Також я поінформований, що відповідно до «Положення про Репозитарій (електронну базу даних) Державного університету економіки і технологій» зазначена робота буде розміщена в Електронному архіві Університету (Репозитарії ДУЕТ). З умовами такого розміщення ознайомлений.

Дата підпис ініціали, прізвище

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЕКОНОМІКИ І ТЕХНОЛОГІЙ

ННІ/факультет	Інформаційних технологій
Кафедра	Інформатики і прикладного програмного забезпечення
Спеціальність	Інженерія програмного забезпечення
Форма навчання	Денна

«ЗАТВЕРДЖУЮ»
Завідувач кафедри _____ Зеленський О.С.
(підпис) (Прізвище, ініціали)
« 11 » червня 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ

1. Тема роботи «Розробка програмного забезпечення для збереження та спільного використання списків фільмів»

Керівник роботи к.т.н., доцент Хоцкіна В.Б.
затверджені наказом закладу вищої освіти від «04» квітня 2025 р. № 222-ст

2. Строк подання здобувачем роботи до «09» червня 2025 р.

3. Зміст кваліфікаційної роботи, об’єкт, предмет та мета дослідження:

Розділ 1. Постановка задачі

Розділ 2. Розробка алгоритму розв’язання задачі

Розділ 3. Організація інформаційного забезпечення

Розділ 4. Розробка застосунків

Об’єкт дослідження: програмні засоби для управління контентом користувача в галузі фільмових рекомендацій

Предмет дослідження: технічна реалізація та користувацька взаємодія з системою збереження й спільного використання списків фільмів

Мета кваліфікаційної роботи: *створення програмного забезпечення, яке забезпечує зручний спосіб роботи з особистими кінодобірками, дозволяє зберігати, структурувати та використовувати списки у веб- і десктоп-форматі, а також вводить елемент гри у процес вибору*

5. Дата видачі завдання «04» квітня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів МДР	Строк виконання етапів роботи	Відмітка керівника про виконання етапів (дата, підпис)
1	Підготовка розділу 1	04.04.2025-13.04.2025	
2	Підготовка розділу 2	14.04.2025-26.04.2025	
3.	Підготовка розділу 3	27.04.2025-15.05.2025	
4	Підготовка розділу 4	16.05.2025-08.06.2025	
5	Реєстрація завершеної кваліфікаційної роботи	09.06.2025	Реєстраційний №___ «09»червня 2025 р.
6	Отримання відгуку від наукового керівника	10.06.2025	
7	Подання кваліфікаційної роботи на перегляд завідувачу кафедри	11.06.2025	
8	Отримання зовнішньої рецензії	12.06.2025	
9	Попередній захист кваліфікаційної роботи на кафедрі	13.06.2025	
10	Підготовка до захисту в ЕК	16.06.2025-21.06.2025	

Завдання підготував науковий керівник _____ __Хоцкіна В.Б.___
(підпис) (прізвище та ініціали)

Завдання одержав _____ __Желудченко Д.О.
_
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

на бакалаврську кваліфікаційну роботу

«Розробка програмного забезпечення для збереження та спільного використання списків фільмів»

Желудченка Дмитра Олександровича

Кваліфікаційна бакалаврська робота на здобуття освітньо-кваліфікаційного рівня бакалавра зі спеціальності 121 «Інженерія програмного забезпечення» – Державний університет економіки і технологій – Кривий Ріг, 2025.

У бакалаврській дипломній роботі реалізовано програмне забезпечення для збереження, візуалізації та вибору фільмів. Складовими проєкту є вебзастосунок для створення, редагування та перегляду списків фільмів, а також окремий десктопний модуль з тривимірним інтерфейсом, у якому фільми виводяться на гранях інтерактивного куба. Текстури граней формуються на основі афіш обраних фільмів. Після завершення обертання куба визначається грань, повернута до камери, – відповідний фільм вважається вибраним. Вебчастину створено з використанням JavaScript, віндовс частину – мовою C# із застосуванням OpenGL.

Ключові слова: OpenGL, вебзастосунок, 3D-графіка, база даних, C#, інтерфейс, список фільмів.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

Скорочення	Розшифрування
API	Прикладний програмний інтерфейс — це набір чітко визначених методів для взаємодії різних компонентів.
БД(база даних)	Впорядкований набір логічно взаємопов’язаних даних, що використовуються спільно та призначені для задоволення інформаційних потреб користувачів.
СУБД	Система управління базами даних.
ПЗ	Програмне забезпечення.

ЗМІСТ

ВСТУП

Сучасна людина щодня стикається з великою кількістю інформації та розмаїттям вибору — це стосується й перегляду фільмів. Багато користувачів проводять значну кількість часу в пошуках того, що переглянути, прокручуючи списки, рейтинги, рекомендації, але так і не зупиняються на конкретному варіанті. Попри наявність популярних онлайн-ресурсів із каталогами фільмів, проблема зручного збереження власних добірок та швидкого вибору залишається відкритою.

У відповідь на цю ситуацію було вирішено створити два взаємопов’язані програмні продукти: вебзастосунок для створення й упорядкування списків фільмів та настільну програму, що допомагає обрати фільм у формі інтерактивної гри. Завдяки підключенню до API ресурсу The Movie Database (TMDb), користувачі маютимуть змогу шукати фільми, додавати їх до списків, зберігати добірки, а також взаємодіяти з ними в нетиповий спосіб — через механіку "кістки", яка обирає випадковий фільм зі списку. Об’єктом дослідження є програмні засоби для управління контентом користувача в галузі фільмових рекомендацій. Предмет дослідження — технічна реалізація та користувацька взаємодія з системою збереження й спільного використання списків фільмів. Метою роботи є створення програмного забезпечення, яке забезпечує зручний спосіб роботи з особистими кінодобірками, дозволяє зберігати, структурувати та використовувати списки у веб- і десктоп-форматі, а також вводить елемент гри у процес вибору.

Для досягнення цієї мети були поставлені такі завдання:

1. реалізувати вебзастосунок з можливістю пошуку фільмів, створення списків і додавання фільмів до них;
2. розробити десктоп-додаток із функцією авторизації та реєстрації користувачів;
3. впровадити інтерактивний механізм вибору фільму на основі випадковості;

РОЗДІЛ 1
ПОСТАНОВКА ЗАДАЧІ

1.1. Характеристика задачі

У процесі розробки програмного забезпечення для збереження та використання списків фільмів буде поставлено завдання вирішити декілька паралельних і пов’язаних між собою проблем: структуризацію персональних уподобань користувача, швидкий доступ до фільмових добірок, зменшення часу на прийняття рішення про перегляд і створення цікавої механіки в самому процесі вибору.

Ця задача полягатиме в тому, щоб врахувати специфіку того, як люди сприймають вибір у контексті великої кількості схожих варіантів. Користувачеві не вистачатиме просто каталогу — йому буде потрібен спосіб фільтрувати, пам’ятати, відкладати й несподівано повертатися до обраного. Тому вирішення задачі включатиме не тільки інженерну реалізацію, а й проектування поведінкової логіки.

У проєкті буде створено два окремих застосунки, що розв’язуватимуть задачу з різних сторін. Вебзастосунок зосередиться на доступі до великої кількості фільмів, пошуку за ключовими словами, формуванні персональних добірок. Десктопна програма моделюватиме момент безпосереднього вибору фільму у форматі кидання кубика.

У межах вебінтерфейсу користувач буде отримувати доступ до інформаційної бази фільмів, використовуючи публічне API TMDb. Задача полягатиме не лише в тому, щоб відобразити контент, а й у тому, щоб забезпечити зручне керування добірками. Користувач зможе створювати списки, додавати туди фільми, змінювати порядок, групувати їх за власними критеріями.

У рамках задачі буде розроблено систему пошуку, яка дозволить швидко орієнтуватися в наявному масиві даних. Алгоритми не будуть нав’язувати

рекомендації, а надаватимуть лише базовий фільтр по жанру. Функціонал буде спрямований не на прогнозування смаку, а на збереження вже сформованих інтересів.

Особливу увагу буде приділено питанню збереження та доступу до списків. Частина користувачів не реєструватиметься, тому буде реалізовано локальне збереження даних у браузері. Для користувачів, які матимуть облікові записи, буде передбачено варіант розміщення даних на сервері. Але основна логіка застосунку не залежатиме від цього — вона працюватиме автономно.

У межах задачі буде також вирішено питання візуальної структури: як представити списки, як показати фільми, як мінімізувати кількість дій, потрібних для додавання фільму до списку. Це вплине на загальну модель використання застосунку, яка передбачатиме швидке, «м’яке» занурення в роботу без потреби довгого навчання або навігації.

Друга частина задачі буде зосереджена на іншому сценарії. Після того як користувач збере певну кількість фільмів, виникатиме нова проблема — вибір одного з них. Програма моделюватиме ситуацію, коли користувач не може визначитися, що саме подивитися, і передасть це рішенню випадку. Але на відміну від простої генерації випадкового числа, вона буде візуалізувати процес.

У рамках задачі буде розроблено механіку, яка дозволить користувачеві прикріпити фільми до віртуального кубика. Кожна грань буде представляти один фільм — через обкладинку або короткий опис. Після запуску кубик обертатиметься й зупинятиметься на випадковому боці, таким чином «вибираючи» фільм.

Цей підхід дозволить уникнути складних рішень, перетворивши вибір на дію, близьку до гри. Сам механізм буде максимально простим: обрати фільми, натиснути кнопку, отримати результат. Але за цією простотою стоятиме серйозне конструкторське завдання: як зробити взаємодію швидкою, стабільною, з достатнім візуальним відгуком.

Також буде розроблено систему автентифікації користувача. Після реєстрації користувач зможе зберігати свої добірки для наступного використання. Застосунок працюватиме локально, без обов’язкової синхронізації, що забезпечить автономність та приватність. Архітектура проєкту дозволить у майбутньому реалізувати обмін даними з вебверсією, але на першому етапі це не буде розглянуто.

Уся система в цілому вирішуватиме задачу, яка лежить на межі між інформаційним менеджментом і взаємодією людини з інформацією на емоційному рівні. З одного боку, йтиметься про збереження чітких, структурованих даних, як списків, фільмів, метаданих. З іншого — про вплив на поведінку користувача в момент прийняття рішення.

Задача розробки охоплює кілька функціональних компонентів:

1. Інтелектуальний пошук.

Реалізація механізму взаємодії з великим обсягом відкритих даних про фільми. Пошуковий інтерфейс забезпечує обробку запиту користувача та фільтрацію результатів за заданими критеріями жанру.

2. Керування добірками. створення, редагування, оновлення та зберігання індивідуальних списків фільмів.

Забезпечується локальне чи віддалене збереження даних користувача з подальшою можливістю інтерактивного керування вмістом.

3. Альтернативний інтерфейс вибору.

Реалізація моделі вибору фільму не у вигляді стандартного списку, а через механізм випадкової візуалізації. Вибір відбувається шляхом обертання тривимірного куба, кожна грань якого містить текстуру з афішею одного з фільмів. Після завершення анімації визначається активна грань та, відповідно, вибраний фільм.

У процесі формування вимог враховано такі обмеження та технічні цілі:

- Оптимізація навігації у великому обсязі варіантів. Система мінімізує когнітивне навантаження через категоризацію фільмів, візуальні підказки, а також використання механіки випадкового вибору.
- Мінімізація часу взаємодії. Проєктування інтерфейсу з урахуванням принципів швидкої реакції — з моменту запиту до отримання результату минає мінімум кроків. Доступ до перегляду фільму здійснюється без потреби тривалого аналізу списку.
- Адаптивність до користувацької поведінки. Вибір фільму реалізовано як інструмент, що знижує напругу, пов’язану з прийняттям рішень. Випадкове обертання куба замінює процес перегляду списку, подаючи варіанти у спрощеному, більш інтуїтивному вигляді.

1.2. Огляд існуючих рішень

Перед розробкою власного програмного забезпечення для збереження та взаємного використання списків фільмів проведено аналіз існуючих сервісів, які виконують схожі функції. Основна увага приділятиметься можливостям користувача створювати добірки, шукати контент, працювати з фільмовими метаданими та організовувати процес вибору. Також буде розглянуто зовнішній вигляд інтерфейсів, логіку навігації, рівень інтеграції з базами даних і доступність персоналізованих функцій.

Сервіс The Movie Database (https://www.themoviedb.org) є відкритою онлайн-базою фільмів, серіалів, акторів і телепередач, яку підтримує спільнота. На головній сторінці розміщується динамічний візуальний блок з постерами популярних або нових фільмів, під яким користувачеві пропонуються добірки за категоріями: популярне, у кіно, на телебаченні, тощо. У верхній частині інтерфейсу розташована панель пошуку, а також доступ до персонального профілю, мовних налаштувань і меню навігації.

Після входу користувач може створювати власні списки, додавати фільми, писати коментарі, зберігати фільми до закладок. Інтерфейс списку

містить назву, опис, прев'ю постерів і можливість сортування. Робота з такими списками реалізована на сторінці користувача, де додано функції перегляду історії, оцінювання, рейтингів та активності.

Окремим плюсом сервісу є відкрите API, яке дає змогу стороннім застосункам отримувати доступ до фільмової бази, що буде використано у розробці власного програмного продукту.

Сервіс Letterboxd (https://letterboxd.com) виконує функцію соціальної платформи для кіноглядачів. Його інтерфейс орієнтований на перегляд та оцінювання фільмів, створення списків, написання рецензій і взаємодію між користувачами. Домашня сторінка демонструє активність інших — хто що переглядав, оцінював, рецензував.

Фільми мають окремі сторінки з великим постером, коротким описом, датою випуску, трейлером, акторським складом і добіркою користувацьких оцінок. Нижче подається стрічка з рецензіями та оцінками інших користувачів. Інтерфейс особистого профілю містить розділи «Diary» (щоденник перегляду), «Lists» (створені списки), «Likes», «Reviews», «Watchlist».

Списки на Letterboxd можуть мати візуальний або текстовий формат, користувач вказує назву, обкладинку, короткий опис і набір фільмів. Є можливість сортувати за різними критеріями — датою додавання, рейтингом, абеткою. Особливістю платформи є сильна соціальна складова — підписки, стрічка новин, коментарі.

IMDb (https://www.imdb.com) — одна з найстаріших і найбільших баз фільмів у світі. Основна структура сайту побудована довкола фільмових карток, що містять опис, акторський склад, трейлери, рейтинг, жанри, відгуки. У правій частині сторінки фільму подаються рекомендовані фільми, рейтинги, галерея зображень.

Зареєстровані користувачі мають змогу створювати списки — публічні чи приватні. Ці списки мають табличну структуру, з можливістю додавати замітки до кожної позиції, сортувати, редагувати або експортувати. Механізм

пошуку доволі гнучкий: користувач може вказати жанр, рік, країну, рейтинг тощо.

Однією з найбільш вживаних функцій є Watchlist — список, куди користувач додає фільми для майбутнього перегляду. У цьому списку зручно переглядати статус перегляду, змінювати пріоритет, додавати фільми в інші добірки.

Сервіс JustWatch (https://www.justwatch.com) виконує роль агрегатора стрімінгових платформ. Його головна мета — допомогти користувачеві знайти, де можна переглянути певний фільм або серіал легально. Сервіс орієнтований переважно на користувачів, які активно користуються Netflix, Amazon Prime, Hulu, Apple TV+ та іншими сервісами.

Інтерфейс побудований на системі фільтрів: країна, платформа, жанр, якість (HD/4K), рік, ціна. Результати подаються у вигляді постерів із зазначенням платформи, ціни або доступності для передплатників.

Хоча JustWatch не пропонує повноцінних добірок чи рецензій, у нього є функція Watchlist, де користувач може відмічати фільми для перегляду. Цей список синхронізується з обраною країною та доступними сервісами.

Платформа Simkl (https://simkl.com) спеціалізується на створенні особистих профілів перегляду: телесеріали, аніме, фільми. Користувач створює акаунт, вказує сервіси, які використовує, й отримує доступ до власного кабінету, де можна візуалізувати, що вже переглянуто, що в процесі, а що заплановано.

Інтерфейс складається з дашбордів, статистики, календаря прем’єр, історії переглядів. Користувач формує списки: Custom Lists, Planned, Watching, Completed. Кожен фільм чи серіал супроводжується мінімальним набором даних — обкладинка, назва, рейтинг, жанр.

Simkl має синхронізацію з іншими платформами, такими як Netflix, MyAnimeList, що дозволяє зменшити ручне додавання. Також платформа пропонує API для розробників і інтеграцію з Discord або Telegram.

Reelgood (https://reelgood.com) — сервіс, який поєднує рекомендації та доступ до потокового контенту. Користувач обирає сервіси, які він використовує, і отримує стрічку з контентом, який доступний на цих платформах. Присутні фільтри за жанрами, популярністю, роком, а також рейтинги IMDb та Rotten Tomatoes.

Функціонал списків представлений у вигляді Watchlist та обраного. Фільми та серіали можна додавати до списків для збереження й подальшого перегляду. Інтерфейс надає відчуття суцільного потоку, де фільми швидко гортатимуться, переглядатимуться деталі й прийматиметься рішення про перегляд.

Існуючі сервіси покривають різні аспекти взаємодії користувача з фільмовим контентом. Деякі, як IMDb і TMDb, концентруються на деталях і структурі бази даних, інші — як Letterboxd — акцентують увагу на спільнотному використанні, рецензіях та емоційній складовій перегляду. JustWatch і Reelgood вирішують питання доступності перегляду, а Simkl орієнтується на особисту статистику та статус перегляду.

Натомість серед розглянутих рішень майже відсутній підхід до гейміфікованого вибору або представлення процесу вибору як окремої дії з елементом жеброкування. Також не всі сервіси дозволяють створювати списки повністю автономно — без підключення до акаунтів або облікових систем.

Ці особливості буде враховано при формуванні концепції власного застосунку, де збереження фільмів, створення персональних добірок і процес вибору відбуватимуться в різних форматах — функціональному та ігровому.

1.3. Аналіз предметної області

Сфера, що охоплює перегляд і організацію кіно- та серіального контенту, продовжує активно розвиватися. Поширення потокових сервісів, відкритих фільмових баз і рекомендаційних алгоритмів змінило поведінку користувачів: перегляд більше не обмежується лінійною телепрограмою чи

випадковим вибором. На перший план виходить здатність користувача формувати власні добірки, відстежувати переглянуте, планувати нове та ділитися цим із іншими. У межах цього середовища виникає потреба в цифрових інструментах, які допомагають не лише шукати інформацію про фільми, але й структурувати особисті вподобання.

У предметну область входять такі об’єкти:

- кіноконтент (фільми, серіали, телепередачі);
- метаінформація про контент (назва, жанр, дата виходу, постер, опис, актори, рейтинг);
- добірки та списки (власні або спільні);
- користувачі та їхня взаємодія з контентом;
- процес вибору для перегляду.

Списки фільмів — це не лише спосіб зберігати контент для майбутнього перегляду. Вони виконують кілька функцій: організують вподобання, дозволяють планувати, відображають смак або настрій, дають змогу рекомендувати іншим або повертатися до переглянутого. У контексті предметної області списки можна класифікувати як:

- персональні — створені однією особою для власного користування;
- спільні — формуються кількома користувачами (напр., компанією друзів);
- тематичні — об’єднані довкола жанру, події, режисера, настрою тощо.

Користувач у таких системах часто виконує дії: пошук, додавання, вилучення, перегляд, оцінювання, збереження, фільтрація. Ці дії не потребують складних алгоритмів, однак передбачають стабільну роботу інтерфейсу, можливість повторного звернення до створених добірок і просту модифікацію.

Окрема категорія проблеми — це процес вибору фільму. У реальному середовищі ця дія часто супроводжується тривалими обговореннями або відкладанням, коли кілька людей не можуть визначитися, що саме переглядати. Така ситуація характерна як для сімей, так і для груп друзів.

Вибір у цьому контексті виконує кілька ролей:

- він зменшує невизначеність, пропонуючи конкретний варіант;
- економить час, коли перелік варіантів заздалегідь задано;
- знижує напруження у спільному рішенні.

З огляду на це, у предметній області можна виділити поняття механізму вибору, який спирається або на випадковість, або на фільтри, або на попередні дії користувача (рейтинги, історію). В рамках розроблюваного програмного забезпечення такий механізм реалізовано через взаємодію з візуальним об’єктом — кубиком, обкладинки фільмів «наклеюються» на його сторони. Це дозволяє перетворити вибір на окрему подію, де результат стає несподіваним, але обмеженим наперед заданим набором.

Більшість сучасних застосунків, пов’язаних із фільмами, використовують відкриті бази даних через API. Найпоширеніший варіант — The Movie Database (TMDb), що надає інформацію про мільйони фільмів, серіалів, акторів та іншого контенту [8].

API TMDb охоплює:

- пошук за назвою, жанром, актором;
- отримання постерів, описів, дати виходу;
- формування добірок;
- популярні/трендові позиції.

З технічного погляду, взаємодія з API — це запити у форматі JSON, які обробляються у клієнтському або серверному застосунку. Цей підхід дозволяє зменшити обсяг власних даних і сфокусуватися на функціональній частині: створенні списків, відображенні постерів, виборі та збереженні контенту.

У предметній області користувач не лише споживає інформацію, а й активно її структурує. Його дії — основне джерело динаміки системи. Саме він створює списки, фільтрує, додає нові фільми, скидає кубик для вибору, зберігає або видаляє елементи. Таким чином, взаємодія з системою є циклічною та повторюваною. Важливо не те, скільки дій доступно, а як вони

пов’язані між собою і наскільки стабільно система реагує на послідовність цих дій.

Користувач може мати різні моделі поведінки:

- пасивна: перегляд лише трендових фільмів, без створення списків;
- структурована: створення тематичних добірок, сортування, каталогізація;
- експериментальна: випадковий вибір через кубик або подібні елементи.

Кожна з цих моделей передбачає певні функціональні компоненти в програмному забезпеченні, тож при проектуванні потрібно враховувати різні сценарії.

В предметній області велике значення має візуальне представлення. Постери фільмів, їхнє поєднання в одному просторі, вигляд кубика, що обертається, — усе це сприяє емоційній взаємодії з контентом. Об’єкти, з якими працює користувач, не є лише рядками тексту: кожен має зображення, форму, кольори, що посилюють відчуття персонального простору.

В межах десктопної частини застосунку реалізується інтерфейс, у якому фільми виводяться у вигляді граней куба. Це наближає взаємодію до гри або візуального досвіду, де користувач не лише отримує результат, а й бере участь у його формуванні.

Висновки до розділу 1

У першому розділі було визначено контекст завдання, сформульовано основні поняття предметної області та розглянуто приклади існуючих програмних рішень, що частково або повністю перетинаються з темою дослідження.

Проаналізовано особливості організації інформації про фільми в цифрових середовищах. Виявлено, що користувачі прагнуть не лише отримувати інформацію про контент, а й структурувати її відповідно до

власних потреб — шляхом створення персональних списків, пошуку за критеріями, групування за темами чи жанрами.

Огляд існуючих платформ, таких як Letterboxd, IMDb, TMDb, Netflix та JustWatch, показав, що попри багатофункціональність деяких із них, більшість не надає достатньої гнучкості у створенні списків або не фокусується на спільному використанні добірок. Окремі рішення взагалі не передбачають взаємодію між користувачами чи візуальний механізм вибору.

РОЗДІЛ 2
РОЗРОБКА АЛГОРИТМУ РОЗВ’ЯЗАННЯ ЗАДАЧІ

2.1. Проектування застосунку

Десктопна програма реалізує особливий сценарій користування, орієнтований на момент вибору фільму із заздалегідь сформованого набору. На відміну від вебверсії, де головним завданням є пошук і створення списків, десктопне рішення функціонує як візуалізатор можливостей і середовище для прийняття рішення на основі обмеженого вибору. У центрі концепції — ідея перетворити рутинний процес вибору фільму на інтерактивну дію з мінімальним когнітивним навантаженням для користувача.

Програма розробляється з використанням бібліонтеки Newtonsoft, що дозволяє поєднувати вебтехнології з можливостями нативного десктопного застосунку. Це забезпечує стабільну роботу з файловою системою, зберіганням даних локально та окремим віконним інтерфейсом. Уся взаємодія з користувачем зосереджена в одному вікні, де реалізовано механізм авторизації, вибору списку, побудови куба з обкладинок фільмів та анімації його обертання.

На етапі проєктування було обрано модель, у якій користувач спочатку проходить процедуру реєстрації або входу. Це дає змогу зберігати персональні дані у локальному сховищі, пов’язаному з певним профілем. Після входу користувач отримує доступ до набору списків, створених раніше, або може сформувати новий. Візуальний інтерфейс побудований таким чином, що кожен фільм із вибраного списку займає одну з граней тривимірного куба. Кількість граней обмежена, тому передбачається автоматичний добір або фільтрація вмісту, якщо список перевищує допустиму кількість позицій.

Особливу увагу приділено реалізації механіки обертання куба. З технічного боку, це тривимірна сцена, побудована за допомогою бібліотеки Open GL. Кожна грань куба відображає постер фільму, а результати обертання

є псевдовипадковими. У момент завершення анімації одна грань зупиняється у фронтальній площині — саме цей фільм і пропонується до перегляду. Таким чином, користувач не вибирає безпосередньо, а делегує це застосунку, що зменшує втрати часу на порівняння та вагання.

Архітектурно програма складається з двох шарів: інтерфейсного та логічного. Перший відповідає за візуальне представлення і реакцію на дії користувача — кліки, наведення, натискання кнопок. Другий — за обробку даних, зберігання профілю, вибір фільмів для куба та збереження історії виборів. Між цими шарами існує розподіл функцій, що дозволяє за потреби замінювати окремі компоненти без впливу на загальну структуру.

Дані про фільми отримуються з попередньо збережених списків, які створені безпосередньо в десктопному застосунку перед киданням кубику. Передбачено можливість офлайн-роботи: вся необхідна інформація (назви, постери, опис) кешується при первинному завантаженні і доступна без повторного звернення до зовнішніх джерел. Це дозволяє використовувати програму в будь-який момент, незалежно від наявності інтернет-з’єднання.

Користувацький досвід побудований на спрощеному сценарії, де більшість дій зводиться до одного етапу — натискання кнопки, після чого система самостійно обирає один із доступних варіантів. Такий підхід не вимагає детального аналізу або навігації по численних меню, що дозволяє сконцентруватися на кінцевій меті — визначити, що саме буде переглядатися.

Загальна модель застосунку побудована з урахуванням можливості подальшого розширення. У майбутньому синхронізація між пристроями, розширення візуальних тем, додавання голосового керування та варіантів обертання з фізичною симуляцією. Наразі ж структура залишає простір для таких змін без потреби перебудови всього ядра програми.

Алгоритм функціонування десктопного застосунку побудовано навколо основної задачі — вибору фільму із заданого списку за допомогою візуалізованого елементу, що змодельовано як куб (Рис. 2.1.). Весь процес взаємодії користувача із системою розгортається у кілька послідовних етапів,

кожен з яких активується в момент виконання попередньої дії. Така поетапність дозволяє організувати логіку роботи програми як серію станів, між якими здійснюється керований перехід.

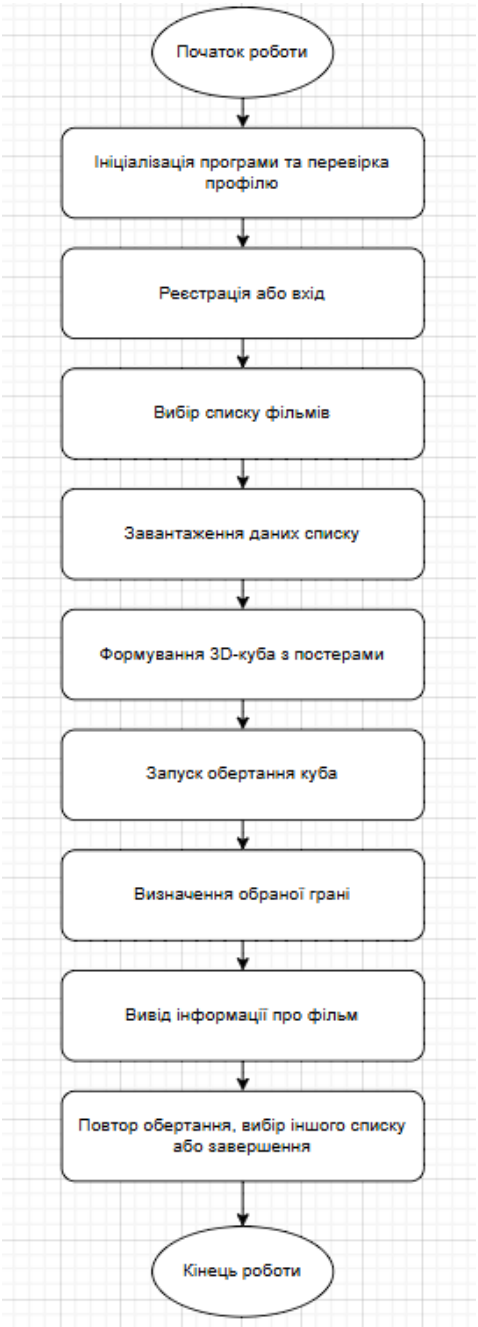

Рис. 2.1. Алгоритм роботи програми

Після запуску програма ініціалізує головне вікно, завантажує стартові ресурси й перевіряє наявність активного профілю користувача. Якщо обліковий запис відсутній, система запрошує пройти реєстрацію, під час якої створюється або зчитується локальний профіль.

Після входу користувач переходить у режим вибору списку. Програма відображає перелік доступних наборів фільмів, збережених раніше. Після вибору конкретного списку система завантажує відповідні дані: назви, зображення обкладинок, короткі описи та унікальні ідентифікатори. Ці дані обробляються програмою, і формується масив, що використовуватиметься для візуалізації.

На основі вибраного списку формується тривимірний об'єкт — куб, кожна грань якого містить обкладинку одного з фільмів. Якщо кількість фільмів у списку перевищує кількість доступних граней, застосунок обирає підмножину випадковим чином або відповідно до налаштувань користувача. На цьому етапі також генерується тривимірна сцена, підключається камера, джерела світла та механізми взаємодії з графічним середовищем.

Коли куб сформовано, інтерфейс переходить у стан готовності до обертання. Користувач ініціює обертання через натискання кнопки або взаємодію з клавіатурою. Куб починає обертатися навколо кількох осей у випадковому напрямку з обраною швидкістю. Цей процес не лише виконує анімаційну функцію, а й фактично відповідає за реалізацію вибору — залежно від завершального положення куба визначається, який фільм буде запропоновано для перегляду.

У момент завершення обертання фіксується грань, яка опинилась у фронтальній площині, відносно спостерігача. З неї зчитується унікальний ідентифікатор фільму, після чого програма звертається до локального масиву, щоб вивести на екран розширену інформацію про вибраний варіант. Відображається назва, опис, постер, рік випуску.

Після цього користувач може завершити роботу, повторити додаток обертання або обрати інший список. Історія вибору фільмів записується у профіль, що дозволяє в майбутньому зберігати результати та уникати повторів, якщо такі налаштування увімкнено.

Така структура алгоритму дозволяє створити цілісний сценарій використання програми без надмірної кількості дій з боку користувача. Уся

складність прихована в логіці програми, тоді як користувач взаємодіє з застосунком через обмежену кількість контрольованих кроків. Це спрощує сприйняття процесу вибору і водночас дозволяє зберегти елемент випадковості, що додає до досвіду використання ігровий характер.

Алгоритм також передбачає обробку ситуацій, коли дані недоступні або неповні. У таких випадках застосунок повідомляє про помилку, а також пропонує повторне завантаження або зміну списку. Таким чином забезпечується контроль над стабільністю програми та передбачуваністю її поведінки.

2.2. Проєктування вебзастосунку

Веб-застосунок планується як інтерактивна платформа, що забезпечить користувачам зручний і швидкий доступ до обширної бази фільмів із можливістю гнучкого пошуку, перегляду, сортування та створення власних списків. Основною метою є реалізація інтуїтивного інтерфейсу з мінімальним когнітивним навантаженням, що дозволить максимально спростити користувацький досвід при роботі з великими масивами інформації.

Для побудови фронтенду обрано бібліотеку React, яка є однією з найпопулярніших технологій для розробки складних односторінкових застосунків (SPA). Вона дозволяє створювати багаторазово використовувані компоненти, що відповідають за окремі логічні частини інтерфейсу — це підвищує масштабованість і підтримуваність коду. Реактивність та віртуальний DOM гарантують швидкий і плавний рендеринг при оновленні стану застосунку.

Для управління даними планується застосувати Apollo Client, який забезпечує взаємодію з серверною частиною через GraphQL. Використання GraphQL дозволить запитувати лише необхідні поля з бази даних, що мінімізує надлишковий трафік і підвищує продуктивність. Apollo Client підтримує

локальне кешування, що дозволить оптимізувати повторні запити та покращити відгук інтерфейсу.

Інтерфейсна бібліотека Material UI буде використана для забезпечення єдиного стилю і адаптивності. Вона надає готові компоненти з реалізацією принципів матеріального дизайну, що позитивно вплине на зручність користування і швидкість розробки. Зокрема, планується застосування типових елементів: кнопок, форм, діалогових вікон, панелей навігації, анімаційних переходів між сторінками.

Реалізація багатомовності буде здійснюватися з використанням бібліотеки i18next. Це дозволить надати користувачам можливість вибору мови інтерфейсу, що суттєво розширить аудиторію застосунку.

Адаптивний дизайн є пріоритетом, тому застосунок буде коректно відображатися як на великих моніторах, так і на мобільних пристроях. Це досягається через гнучке компонування, використання медіазапитів CSS і вбудованих можливостей Material UI.

Планована інтеграція з бекендом передбачає використання GraphQL-сервера на базі Apollo Server. Цей сервер буде відповідати за отримання даних із зовнішніх API (наприклад, The Movie Database) та внутрішньої бази, а також за авторизацію користувачів і збереження їхніх профілів і списків. Взаємодія з бекендом буде організована через стандартизовані GraphQL-запити і мутації, що забезпечить гнучкість та масштабованість.

Функціонал створення і редагування списків передбачатиме простий інтерфейс для організації добірок фільмів. Користувачі зможуть створювати необмежену кількість персональних колекцій, додавати або видаляти фільми, змінювати порядок їхнього відображення. Важливою частиною стане збереження цих списків на сервері, що дасть змогу зберігати дані у хмарі і синхронізувати їх між різними пристроями.

На серверній стороні буде реалізовано масштабовану систему зберігання даних, що поєднуватиме як кешування інформації з зовнішніх джерел, так і управління профілями користувачів і їхніми списками. Застосунок

використовуватиме GraphQL API, що дозволить клієнту ефективно отримувати лише необхідні дані, знижуючи навантаження на мережу і покращуючи швидкість відповіді.

Одним із важливих завдань стане інтеграція із зовнішніми API, зокрема The Movie Database, щоб отримувати оновлену інформацію про фільми, рейтинги, рецензії та постери. Ці дані будуть кешуватися на сервері, що дозволить забезпечити швидкий доступ і зменшити кількість зовнішніх запитів. У разі відсутності інтернет-з’єднання користувач зможе працювати з останніми доступними даними.

Важливою складовою буде й забезпечення плавної та стабільної роботи клієнтської частини навіть при великому навантаженні. Для цього буде застосовано оптимізації рендерингу, lazy loading для зображень і компонентів, а також розподіл логіки між сервером і клієнтом, що допоможе знизити обчислювальне навантаження на браузер користувача.

Загальна архітектура проекту буде гнучкою і модульною, що дозволить в подальшому легко додавати нові функції, покращувати інтерфейс та інтегрувати додаткові сервіси, такі як рекомендації на основі штучного інтелекту, соціальні функції або підтримку мультимедійного контенту.

Висновки до розділу 2

У цьому розділі було проаналізовано основні підходи до проєктування десктопного та веб-застосунку для вибору й перегляду фільмів. Описано принципи роботи десктопної програми, яка використовує унікальний механізм вибору через візуалізацію куба з постерами фільмів, а також подано архітектурні рішення для забезпечення стабільної роботи, автономності і простоти взаємодії користувача. Особлива увага приділена розділенню інтерфейсного та логічного шарів, а також обробці різних сценаріїв використання, що дозволяє гнучко розвивати застосунок у майбутньому.

Також детально описано концепцію веб-застосунку, який забезпечує інтерактивний пошук і керування великими масивами фільмів через сучасні технології React, Apollo Client та GraphQL. Пояснено, як передбачається реалізувати адаптивний і багатомовний інтерфейс, інтеграцію з зовнішніми API та оптимізацію продуктивності для забезпечення комфортного користувацького досвіду. Проєкт передбачає масштабовану архітектуру з можливістю подальшого розширення функціоналу, що дозволить покривати широкий спектр потреб користувачів і підтримувати синхронізацію між різними пристроями.

РОЗДІЛ 3
ОРГАНІЗАЦІЯ ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Структура бази даних

База даних відповідає за організацію та збереження інформації в межах десктопного застосунку. Вона дозволяє впорядкувати дані про користувачів, жанри, фільми та персональні добірки, забезпечуючи їх логічний поділ і зв’язки. Завдяки цьому застосунок може працювати з даними стабільно, передбачувано та узгоджено, що необхідно для реалізації таких функцій, як автентифікація, ведення історії вибору та формування власних списків перегляду.

Для реалізації цієї структури використовується SQLite — легка реляційна система управління базами даних, яка не потребує окремого сервера. Вона працює у вигляді звичайного файлу, що дозволяє зручно інтегрувати її з десктопним застосунком і розгортати разом із ним. Це рішення не потребує налаштування інфраструктури та дає змогу зберігати дані локально, без прив’язки до зовнішніх ресурсів або підключення до мережі.

SQLite добре підходить для сценаріїв, де дані опрацьовуються однією програмою, а кількість одночасних доступів є невеликою [4]. Завдяки своїй простоті вона дозволяє уникнути зайвих залежностей і зберігати структуру проєкту максимально компактною. Водночас система підтримує транзакції, зовнішні ключі та інші механізми, необхідні для гарантування цілісності та достовірності даних.

У структурі бази даних передбачено декілька таблиць, які охоплюють усі сутності, що використовуються в застосунку. Таблиця Users зберігає облікові записи, таблиця Genres — список жанрів, Movies — дані про фільми, а MovieLists — індивідуальні добірки, створені користувачами. Зв’язок між фільмами та списками реалізовано через проміжну таблицю MovieInList, що дозволяє зберігати інформацію про включення конкретного фільму до певного

списку. Такий підхід спрощує розширення функціоналу та дозволяє змінювати логіку формування списків без потреби у зміні основних таблиць.

Під час проєктування було дотримано нормалізації до третьої нормальної форми, що дозволило усунути дублювання інформації та зберігати її в упорядкованому вигляді. Кожна таблиця містить лише ті дані, які безпосередньо стосуються конкретної сутності, а зв’язки між ними реалізуються через зовнішні ключі. Це забезпечує логічну структурованість, зменшує ризик помилок та полегшує обслуговування сховища даних.

SQLite забезпечує достатній рівень захищеності даних завдяки підтримці транзакцій та обмежень на рівні схеми таблиць. Навіть у випадку аварійного завершення роботи програми можна гарантувати, що дані залишаться в узгодженому стані. Це особливо важливо для стабільної роботи застосунку, що містить персоналізовану інформацію та взаємодіє з історією вибору користувача.

Ще однією перевагою є сумісність SQLite із мовою програмування C# і бібліотеками, що застосовуються у .NET-розробці. Це дозволяє реалізовувати доступ до даних як через SQL-запити, так і через ORM-рішення, залежно від потреб і складності запитів. Оскільки база даних зберігається у вигляді одного файлу, це також спрощує її копіювання, резервування та перенесення між пристроями.

Використання SQLite забезпечує надійну роботу локального застосунку, дозволяє ефективно зберігати персональні налаштування та історію взаємодії з контентом, а також не вимагає складного налаштування або обслуговування. Завдяки цьому проєкт залишається зручним у використанні, легким у підтримці та придатним для подальшого розвитку.

3.2. Розробка бази даних застосунку

Таблиця Users призначена для зберігання інформації про користувачів, які мають доступ до десктопного застосунку. Вона реалізує основу для

аутентифікації користувачів та управління їхніми персональними даними. Наявність окремої таблиці для користувачів дозволяє забезпечити безпечний доступ до функціоналу програми, розмежувати права та зберігати індивідуальні налаштування і активність.

Використання унікального ідентифікатора для кожного користувача дає змогу ефективно організовувати зв’язки між таблицями, наприклад, для створення списків фільмів, які належать конкретному користувачу. Зберігання логіна і пароля — це базовий механізм безпеки, що дозволяє контролювати доступ. Важливою практикою є збереження пароля у зашифрованому вигляді для запобігання несанкціонованому доступу до облікових даних.

Також у таблиці зберігаються персональні дані користувача, як-от ім’я та дата народження, що можуть використовуватись для покращення користувацького досвіду, наприклад, для персоналізації рекомендацій чи визначення вікових обмежень на контент. Опис таблиці Users можна побачити в табл. 3.1

Таблиця 3.1

Опис таблиці «Users»

Назва поля	**Тип даних**	**IS NULL**	**Короткий опис**
1	2	3	4
Id	INTEGER	NOT NULL	Унікальний ідентифікатор користувача, автоматично генерується
Login	TEXT	NOT NULL	Логін користувача, унікальне значення, використовується для входу в систему

Продовження таблиці 3.1

1	2	3	4
Password	TEXT	NOT NULL	Пароль користувача, зберігається у зашифрованому вигляді для безпеки
Name	TEXT	NOT NULL	Ім’я користувача, використовується для відображення в інтерфейсі
BirthDate	TEXT	NOT NULL	Дата народження користувача, може застосовуватись для вікових обмежень та персоналізації

Таким чином, таблиця Users забезпечує основу для організації безпечної та персоналізованої взаємодії користувача із застосунком, підтримує багатокористувацький режим та сприяє гнучкому управлінню обліковими даними.

Таблиця Genres зберігає інформацію про жанри фільмів, що є важливою категоризацією у системі. Класифікація фільмів за жанрами допомагає користувачам швидко знаходити потрібний контент за своїми уподобаннями, а також підтримує фільтрацію і сортування у застосунку.

Зберігання жанрів в окремій таблиці дозволяє уникнути дублювання даних у таблиці фільмів та забезпечує цілісність інформації — кожен жанр визначений однією унікальною назвою з власним ідентифікатором. Така нормалізація структури бази даних покращує її продуктивність, оскільки зміни у назві жанру не потребують редагування кожного запису у таблиці фільмів.

Ця таблиця виступає довідником, на який посилаються інші таблиці, наприклад, Movies. Використання зовнішнього ключа GenreId у таблиці

фільмів створює зв’язок, що дозволяє системі зрозуміти, до якого жанру належить той чи інший фільм. Опис таблиці Genres знаходиться в табл. 3.2

Таблиця 3.2

Опис таблиці «Genres»

Назва поля	**Тип даних**	**IS NULL**	**Короткий опис**
Id	INTEGER	NOT NULL	Унікальний ідентифікатор жанру
Name	TEXT	NOT NULL	Назва жанру, наприклад, «Комедія», «Драма»

Таким чином, таблиця Genres служить для централізованого збереження інформації про категорії фільмів, полегшуючи роботу з фільтрацією, пошуком і організацією даних у застосунку.

Таблиця Movies відіграє роль основного сховища інформації про кінопродукцію, яка використовується у десктопному застосунку. У контексті реляційної моделі бази даних ця таблиця є однією з головних сутностей, що зберігає набір атрибутів, що описують окремий об’єкт — фільм.

Реляційна структура таблиці організована таким чином, щоб уникнути надмірності даних і забезпечити простоту оновлення та підтримки інформації. Кожен запис у таблиці відповідає конкретному фільму і має унікальний ідентифікатор (Id), який виступає в ролі первинного ключа. Завдяки цьому забезпечується цілісність даних, тобто виключається можливість дублювання записів.

У базі даних реалізовано механізм зовнішніх ключів, що пов’язує таблицю Movies із таблицею Genres через поле GenreId. Такий зв’язок дозволяє класифікувати фільми за жанрами без дублювання інформації про сам жанр у кожному записі. Це важливо для підтримки нормалізації бази

даних, зокрема другої нормальної форми, яка передбачає усунення часткових залежностей.

Крім того, таблиця містить поля з різними типами даних, що відповідають специфіці інформації. Текстові поля (Title, Overview, PhotoUrl) зберігають символьні рядки — назву, опис та посилання на зображення відповідно. Дата релізу збережена у форматі тексту (ReleaseDate), що зручно для різних форматів дати, а також дозволяє гнучко її обробляти на стороні застосунку. Поле VoteAverage має тип REAL, що дозволяє зберігати дробові числові значення, які відображають середню оцінку фільму користувачами або критиками.

Зберігання зображення фільму у вигляді URL (посилання на ресурс) забезпечує оптимальний підхід, оскільки самі великі файли не зберігаються в базі даних, що зменшує її розмір і підвищує швидкість роботи. Це відповідає принципам розподіленої архітектури, де медіа-дані зберігаються окремо.

Загалом, таблиця Movies є важливою частиною інформаційної системи, що відповідає за збереження і структуризацію даних про фільми. Її організація сприяє ефективній реалізації функцій пошуку, фільтрації, відображення інформації та побудови рекомендацій у десктопному застосунку. Правильне проєктування цієї таблиці дозволяє знизити навантаження на систему, підвищити продуктивність запитів. Опис таблиці Movies можна побачити в табл. 3.3

Таблиця 3.3

Опис таблиці «Movies»

Назва поля	**Тип даних**	**IS NULL**	**Короткий опис**
1	2	3	4
Id	INTEGER	NOT NULL	Унікальний ідентифікатор фільму (первинний ключ)
Title	TEXT	NOT NULL	Назва фільму

Продовження таблиці 3.3

1	2	3	4
PhotoUrl	TEXT	NULL	URL зображення або постера фільму
GenreId	INTEGER	NULL	Посилання на жанр (зовнішній ключ з таблиці Genres)
Overview	TEXT	NULL	Короткий опис сюжету фільму
ReleaseDate	TEXT	NULL	Дата виходу фільму (зберігається у форматі тексту)
VoteAverage	REAL	NULL	Середня оцінка фільму (може бути дробовим числом)

Таким чином, таблиця Movies забезпечує структурований підхід до зберігання інформації, підтримуючи логічні зв’язки з іншими сутностями бази даних і сприяючи ефективному використанню даних у застосунку. Цей підхід відповідає кращим практикам реляційного моделювання і дозволяє легко адаптуватися до змін бізнес-вимог без суттєвих змін у логіці програми.

Таблиця MovieLists виконує функцію організації колекцій фільмів, які належать конкретним користувачам системи. Вона дозволяє створювати персоналізовані списки, наприклад, улюблених або переглянутих фільмів, що підвищує зручність роботи з контентом у десктопному застосунку.

В контексті реляційної бази даних ця таблиця представляє собою сутність, яка зв’язує користувачів (Users) з наборами фільмів. Для цього у таблиці присутнє поле UserId — зовнішній ключ, який посилається на унікальний ідентифікатор користувача в таблиці Users. Такий зв’язок реалізує відношення "один-до-багатьох": один користувач може мати багато списків фільмів, але кожен список належить лише одному користувачу.

Використання окремої таблиці для списків замість додавання списків безпосередньо у таблицю користувачів підвищує гнучкість та розширюваність системи. Це дозволяє користувачам створювати будь-яку кількість списків з різними назвами та призначенням, а також полегшує управління даними та їх оновлення. Такий підхід відповідає принципам нормалізації бази даних, де кожна таблиця відповідає окремій сутності або відношенню.

Поле Id у таблиці MovieLists — це унікальний ідентифікатор списку, який забезпечує його унікальність та дає змогу посилатися на нього з інших таблиць, зокрема MovieInList, де зберігаються зв’язки між списками і фільмами. Це дозволяє ефективно реалізувати структуру "багато-до-багатьох" між користувачами, списками та фільмами.

Таким чином, таблиця MovieLists є проміжною ланкою, що забезпечує логічну організацію і персоналізацію даних про фільми, створюючи зручну та масштабовану систему для збереження вподобань користувачів. Опис таблиці MovieLists можна побачити в табл. 3.4

Таблиця 3.4

Опис таблиці «MovieLists»

Назва поля	**Тип даних**	**IS NULL**	**Короткий опис**
Id	INTEGER	NOT NULL	Унікальний ідентифікатор списку (первинний ключ)
UserId	INTEGER	NOT NULL	Ідентифікатор користувача, власника списку (зовнішній ключ з таблиці Users)

Використання такої структури допомагає ефективно управляти великим обсягом персоналізованих даних, зберігаючи зв’язки між користувачами і їхніми колекціями фільмів. Цей підхід також спрощує подальшу інтеграцію функціоналу, наприклад, спільного доступу до списків або їх експорту,

оскільки кожен список має власний унікальний ідентифікатор і чітко пов’язаний з користувачем.

Таблиця MovieInList реалізує зв’язок між списками фільмів (MovieLists) та конкретними фільмами (Movies). Вона є класичним прикладом таблиці для організації відношення "багато-до-багатьох" у реляційній базі даних.

У моделі даних багато фільмів можуть входити у багато різних списків, створених різними користувачами. Щоб ефективно відобразити таку структуру, необхідна окрема таблиця, яка зберігатиме пари (список, фільм). Вона дозволяє уникнути дублювання інформації і підтримує гнучкість у керуванні колекціями фільмів.

Це дозволяє створювати персоналізовані набори фільмів, які можуть бути різними за змістом та призначенням: наприклад, "Хочу подивитись", "Улюблені", "Переглянуті" тощо. Кожен запис у таблиці MovieInList вказує, що певний фільм входить до конкретного списку.

Поля MovieListId і MovieId утворюють складений первинний ключ таблиці, що гарантує унікальність кожної пари (список-фільм). Крім того, ці поля є зовнішніми ключами, які посилаються на відповідні таблиці MovieLists і Movies. Це підтримує цілісність даних, запобігаючи появі некоректних або непов’язаних записів.

Використання такої проміжної таблиці є стандартною практикою при моделюванні складних зв’язків у реляційних базах даних, оскільки вона дозволяє чітко і прозоро описати взаємозв’язки між сутностями.

Така архітектура дає змогу ефективно додавати чи видаляти фільми зі списків без дублювання даних. Вона також полегшує запити для отримання всіх фільмів у конкретному списку або, навпаки, всіх списків, до яких належить певний фільм. Завдяки цьому реалізується висока гнучкість у роботі з колекціями контенту. Опис таблиці MovieInList зображено в табл. 3.5

Таблиця 3.5

Опис таблиці «MovieInList»

Назва поля	**Тип даних**	**IS NULL**	**Короткий опис**
MovieListId	INTEGER	NOT NULL	Ідентифікатор списку фільмів (зовнішній ключ, посилання на MovieLists.Id)
MovieId	INTEGER	NOT NULL	Ідентифікатор фільму (зовнішній ключ, посилання на Movies.Id)

Це були усі таблиці бази даних, необхідних для створення десктопного застосунку.

Висновки до розділу 3

У цьому розділі було детально описано структуру бази даних, що використовується для керування інформацією у застосунку з добору фільмів. Розглянуто кожну таблицю поетапно — від загальних концепцій до конкретних полів і їх призначення. Вся база даних спроектована за реляційними принципами та нормалізована до третьої нормальної форми, що сприяє уникненню дублювань, підвищенню точності даних і забезпеченню ефективної роботи з інформацією.

У ході опису наведено детальний аналіз структури кожної таблиці, включаючи типи даних, обмеження щодо порожніх значень та функціональне значення полів. Запропонована модель передбачає взаємодію між базою даних і клієнтською частиною застосунку, що реалізована на C# для десктопних пристроїв. Такий підхід гарантує гнучкість, зручність у підтримці й можливість подальшого розширення функціоналу.

РОЗДІЛ 4
РОЗРОБКА ЗАСТОСУНКІВ

4.1. Розробка віндовс-застосунку

Одним із перших кроків при взаємодії користувача з десктопним застосунком є створення персонального облікового запису, через окрему форму, яка відповідає за фіксацію основних ідентифікаційних параметрів та збереження їх у базі даних. У вікні реєстрації реалізовано фіксовану композицію елементів, орієнтовану на заповнення мінімального набору персональних даних (Рис. 4.1.). Просторове розміщення компонентів дозволяє одразу охопити весь набір полів для заповнення без потреби прокручування чи переходів між вкладками.

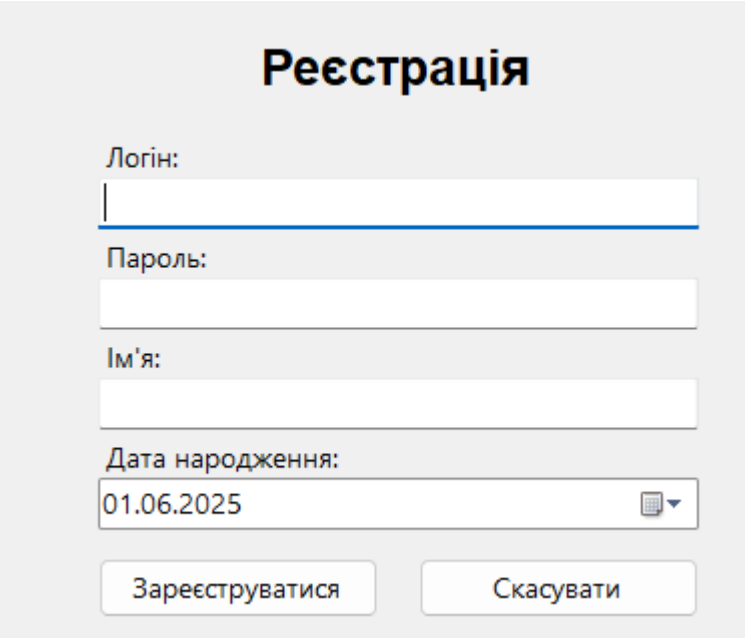

Рис. 4.1. Форма реєстрації

Усі поля, пов’язані з логіном, паролем, іменем та датою народження, розміщені в послідовному вертикальному порядку. У якості заголовка використано заголовок. Для текстового введення логіна, пароля та імені передбачено відповідні поля з обмеженим розміром, достатнім для даних, при

поле паролю має приховане відображення символів з метою захисту конфіденційності.

Реакція на дії користувача реалізована через два інтерактивні елементи — кнопки для підтвердження або скасування процесу. Одна з них ініціює перевірку введених даних, а інша завершує роботу без змін. У разі активації основної кнопки виконується послідовна валідація вмісту кожного поля: перевіряється наявність символів, їх допустимість, а також відповідність формату. Якщо знаходиться порожнє поле, система зупиняє подальше виконання та відображає повідомлення про помилку.

Коли всі умови дотримано, створюється об'єкт, що містить сукупність полів, заповнених користувачем, після чого ініціюється звернення до методу, відповідального за взаємодію з базою даних. У разі успішного збереження нового запису виводиться підтвердження завершення реєстрації. Якщо ж система фіксує конфлікт, виконується відображення відповідного повідомлення без завершення роботи форми.

Форма авторизації відкривається першою при запуску програми (Рис. 4.2.). Її роль полягає у перевірці існування користувача у базі даних, і дозволити або відмовити в доступі до основного функціоналу. Вікно має фіксований розмір, не змінюється вручну та розміщується по центру екрана.

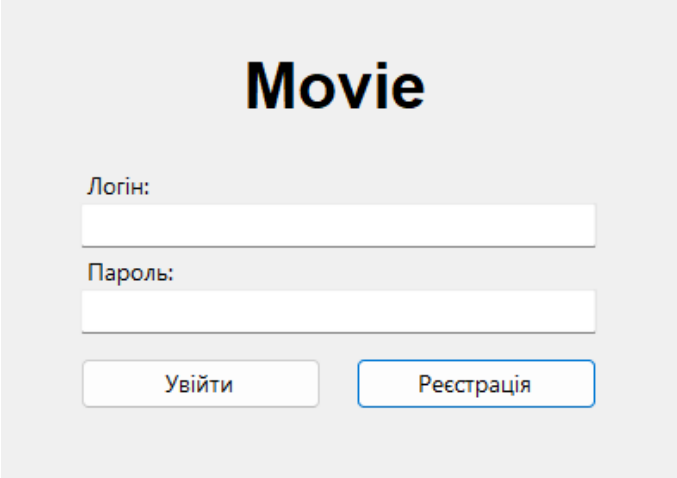

Рис. 4.2. Форма входу

У верхній частині виводиться назва застосунку великими літерами, стилізована як заголовок. Нижче розміщені два підписи до полів, для логіна й пароля, і самі поля введення. Вони однакової ширини, розташовані вертикально одне під одним. Поле для пароля приховує символи, щоб не було видно, що саме вводиться.

У нижній частині вікна — дві кнопки. Перша запускає перевірку логіна й пароля. Якщо хоча б одне з полів порожнє, з’являється попередження. Якщо обидва заповнені, дані передаються функції, яка звіряє їх з тими, що вже збережені. Якщо користувача з такими даними знаходить, відкривається основне вікно, а форма логіну зникає. Якщо нічого не збігається — з’являється повідомлення про помилку.

Друга кнопка відкриває вікно реєстрації, якщо користувача ще немає. Реєстрація відкривається поверх авторизації, а після її завершення або скасування користувач повертається до початкової форми.

Після успішного входу користувача в систему відкривається головне вікно, яке містить основні можливості десктоп-застосунку. Центральним об'єктом тут є візуальна модель куба, на який проєктуються зображення обраних фільмів (Рис. 4.3.). Куб служить механізмом випадкового вибору, а його інтерактивність реалізується через окремий компонент рендерингу.

Верхня частина вікна відведена під персональне привітання, яке змінюється, в залежності від імені користувача. Нижче розміщені дві основні кнопки — одна відкриває форму для вибору фільмів, інша активується лише після того, як користувач обрав рівно шість фільмів. Це обмеження пов’язане з кількістю граней куба, на які можна накласти зображення.

Зліва формується панель зі списком обраних фільмів (Рис. 4.4.). Для кожного елемента виводиться назва та жанр. Структура цієї панелі дозволяє прокручувати вміст, тому навіть при повторному відкритті програми дані, які вже були збережені, підтягуються автоматично. Вибір не треба повторювати вручну, якщо він був збережений раніше.

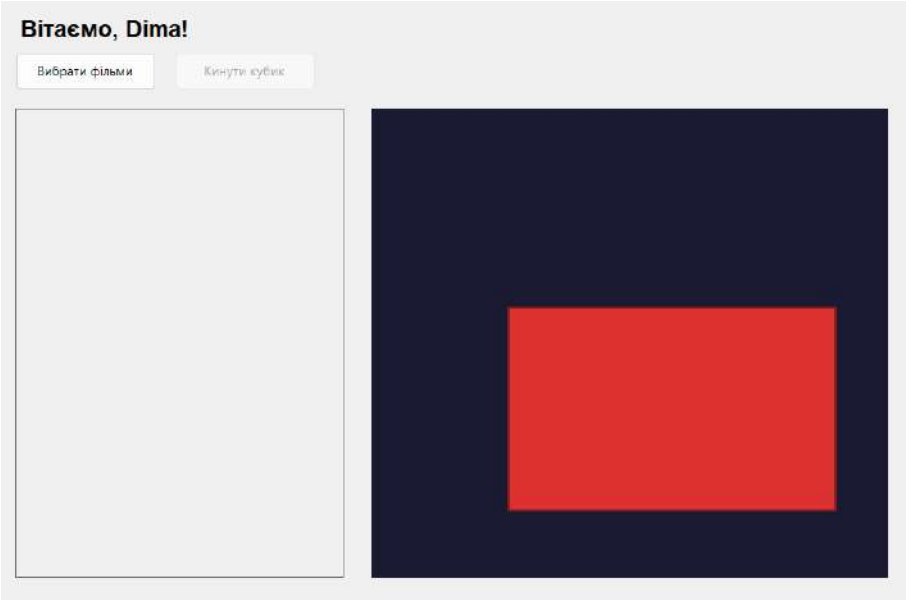

Рис. 4.3. Кабінет користувача

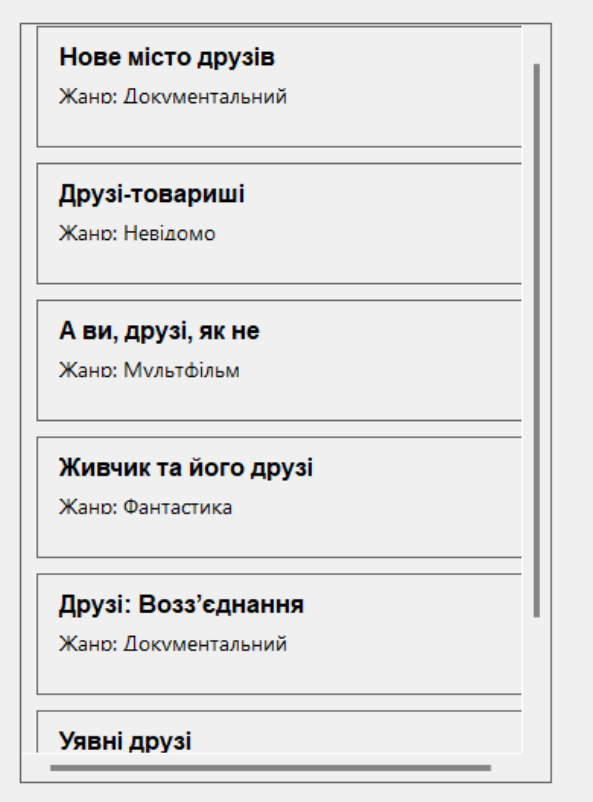

Рис. 4.4. Список обраних фільмів

Кнопка з вибором фільмів відкриває окреме вікно, у якому можна додавати фільми до списку (Рис. 4.5.). Вікно відкривається великого розміру, весь простір розподілено між пошуком і ручним відбором. Ліворуч — потік

результатів, праворуч — список уже вибраного. Спочатку список порожній, з відображенням лічильника. Максимум — шість позицій, і ні на одну більше: як тільки обрано всі, решта кнопок стають неактивними.

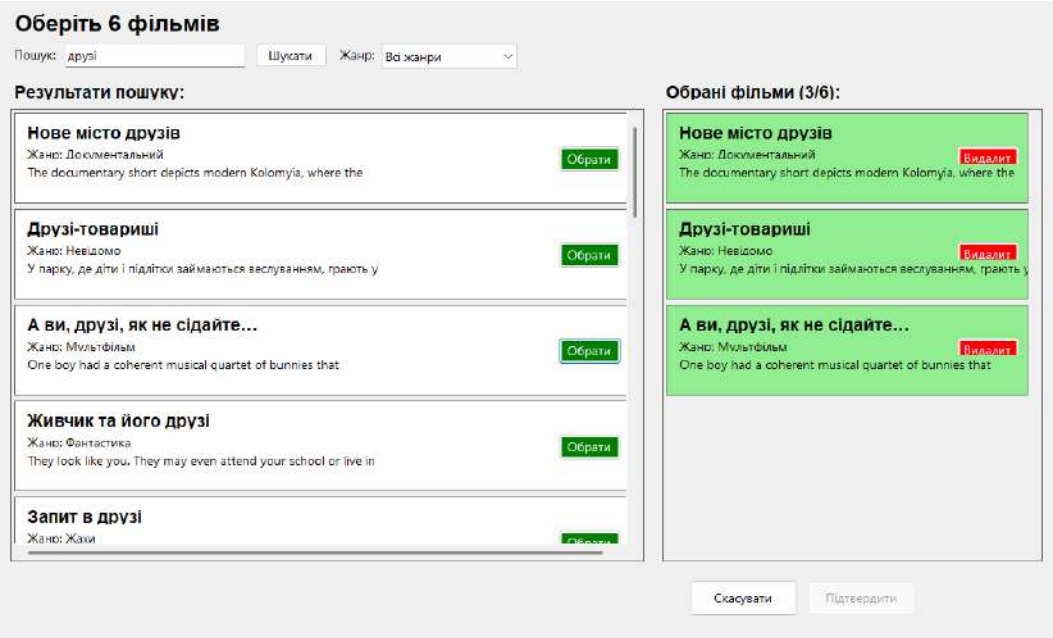

Рис. 4.5. Вікно вибору фільмів

Пошук працює разом із фільтром жанрів. Якщо обрати жанр, але не писати нічого в рядок, нічого не зламається, але результат буде порожнім. Введення запиту не перериває потік, тільки Enter або кнопка «Шукати» запускає реальну дію. Вивід фільмів завжди вертикальний, з фіксованою шириною панелей, обрізаними описами, і не змінюється залежно від кількості. У кожному блоці є кнопка, яка або додає, або видаляє фільм зі списку. Вибір в обидві сторони не потребує перезавантаження чи підтвердження.

Список обраного дублює той самий вигляд елементів, що й у результатах, тільки з іншим кольором фону та червоною кнопкою. Коли список готовий, з’являється можливість натиснути «Підтвердити». Якщо фільмів менше шести — кнопка неактивна.

Після завершення цього етапу відображення у головному вікні оновлюється автоматично, оновлюючі список обраних фільмів. Якщо кількість обраних фільмів дорівнює шести, вмикається кнопка для запуску

куба. Після цього користувач може кинути кубик, і система візуалізує результат на екрані.

Візуалізація куба у застосунку реалізована як окремий візуальний блок, який розміщується у правій частині вікна програми та займає значну площу інтерфейсу [Додаток А]. Він не масштабується вручну й автоматично оновлюється щоразу після завантаження списку фільмів. Кожна з шести граней куба відображає постер одного фільму зі списку. Це може бути як справжня афіша, отримана за вказаним URL, так і згенероване зображення з текстовою назвою фільму на однотонному фоні, якщо справжній постер недоступний.

Куб відображає афіші шести фільмів, які перед тим було додано у список. Кожна грань відповідає одному фільму (Рис. 4.6.). Текстури це завантажені постери за вказаними URL, або, якщо зображення недоступне, згенероване зображення з назвою фільму на однотонному тлі.

Формування куба відбувається динамічно за допомогою засобів OpenGL. У процесі ініціалізації створюється 3D-модель куба, кожній з шести граней якого присвоюється унікальна текстура. Спочатку програма завантажує або генерує зображення, після чого виконує їх обробку — масштабування до стандартного розміру та конвертацію у формат, сумісний із OpenGL. Кожна текстура завантажується у відеопам’ять за допомогою GL.TexImage2D, після чого прив’язується до відповідної грані куба. Границі визначаються через координати вершин та координати накладення текстури.

Рис. 4.6. Кубик з текстурами

Рендеринг виконується під час кожного циклу оновлення візуального інтерфейсу. Відбувається очищення буфера зображення, встановлення камери й перспективної проєкції, а також обчислення трансформацій — обертання навколо осей X, Y та Z. Сам куб малюється через побудову шести прямокутників (дві трикутні сітки на кожну грань), на які накладаються текстури. Це забезпечує тривимірний вигляд та дозволяє створити ілюзію реального фізичного об’єкта, що обертається.

Механізм анімації куба реалізовано через таймер. Під час натискання на кнопку запуску, куб починає обертатися навколо трьох осей одночасно. Для кожної осі випадково задається кількість обертів у межах від 2 до 4 повних обертів (720–1440°), щоб забезпечити непередбачуваність результату. Швидкість обертання плавно зменшується згідно з функцією ease-out, що створює ефект фізичного згасання руху.

Після визначення сторони користувач бачить повідомлення з назвою обраного фільму (Рис. 4.7.). Уся візуалізація відбувається в OpenGL: сцена

очищується, задається перспектива, куб позиціонується в просторі, і тоді вже малюється з відповідними обертами та текстурами.

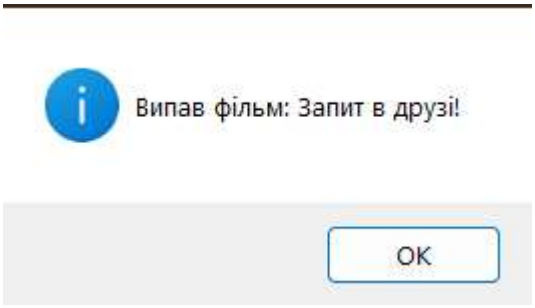

Рис. 4.7. Результат, який випав

Програма автоматично підвантажує попередньо збережені дані користувача: якщо раніше було зібрано список, він завантажується і відображається на панелі. Кожен перезапуск програми потребує авторизації, і список фільмів очищається, якщо програму було закрито.

4.2 Розробка бекенду сайту

Серверна частина реалізована на основі фреймворка Apollo Server, який надає можливість обробляти GraphQL-запити від клієнтського застосунку. Основною метою сервера є отримання інформації про фільми з зовнішнього сервісу The Movie Database (TMDb) через REST API, трансформація отриманих даних і повернення їх клієнту у зручному для подальшої обробки форматі.

У центральному файлі index.js створюється екземпляр Apollo Server, де вказуються визначення схеми та резолверів. Схема описана у форматі GraphQL SDL у файлі schema.graphql. Вона містить типи даних, такі як Movie та Movies, а також описує запити, які клієнт може робити: наприклад, запит movies приймає параметри сторінки пагінації та мови інтерфейсу і повертає об’єкт типу Movies з масивом фільмів і додатковою інформацією.

Резолвери реалізовані у файлі Query.js. Кожна функція-резолвер отримує аргументи від клієнта, формує HTTP-запит до TMDb через axios, а потім обробляє отриману відповідь. Наприклад, функція для запиту списку

фільмів виконує axios.get за URL типу https://api.themoviedb.org/3/movie/popular, передаючи параметри ключа API, номер сторінки та мови. Отримані сирі дані, що містять масив фільмів, передаються до конструктора класу Movies, який знаходиться у папці entities.

Клас Movies приймає відповідь від TMDb і виконує перетворення кожного елемента масиву у об’єкт типу Movie. Конструктор класу Movie приймає окремий фільм у вигляді сирого JSON-об'єкту і нормалізує поля. Наприклад, для формування URL постера він використовує базову адресу, яка зберігається у конфігурації, та шлях до зображення, отриманий від TMDb. Якщо постер відсутній, генерується заглушка — зображення з назвою фільму на однотонному фоні. Для дати виходу фільму виконується форматування у вигляді DD.MM.YYYY з урахуванням локалізації. Жанри фільму передаються як масив рядків, отриманих через співставлення ID жанрів із внутрішньою таблицею.

Завдяки такій інкапсуляції логіки в класах Movie і Movies клієнт отримує однорідні об’єкти без необхідності самостійно розбирати складну структуру відповіді TMDb. У результаті, кожен фільм містить поля: id, title, overview, posterPath, releaseDate, genres тощо.

Усі запити підтримують параметр language, який передається до TMDb API, що дозволяє отримувати локалізовані назви і описи фільмів. Це важливо для інтернаціоналізації інтерфейсу.

Ключ API зберігається у файлі конфігурації config/index.js. Цей файл також містить налаштування базового URL для завантаження постерів, параметри кешування та інші константи. Таким чином, зміна налаштувань не вимагає редагування логіки обробки запитів.

Загальний цикл обробки клієнтського запиту виглядає так: клієнт через GraphQL надсилає запит, сервер запускає відповідний резолвер, який робить HTTP-запит до TMDb API, отримує сирі дані, формує з них об'єкти класів Movies та Movie, а потім повертає клієнту оброблену структуру. Вся ця

взаємодія прозора для користувача — клієнт отримує готові до відображення дані.

Завдяки використанню Apollo Server і модульній архітектурі резолверів, реалізація легко масштабується: можна додавати нові типи запитів, додаткову бізнес-логіку, обробку помилок, а також інтегрувати кешування або механізми авторизації.

Нижче наведено фрагмент резолвера, який обробляє запит популярних фільмів:

```
const axios = require('axios');
const { Movies } = require('./entities/Movies');
const config = require('./config');

async function movies(_, args) {
  const response = await axios.get(`${config.tmdbApiBaseUrl}/movie/popular`, {
    params: {
      api_key: config.apiKey,
      page: args.page || 1,
      language: args.language || 'en-US'
    }
  });
  return new Movies(response.data);
}
```

У цьому прикладі видно, що через axios.get відбувається звернення до зовнішнього API з параметрами, отриманими із запиту клієнта. Результат передається у конструктор Movies, який готує дані для подальшої віддачі клієнту.

Таким чином, серверна частина забезпечує стабільну та гнучку роботу з даними TMDb, адаптуючи їх під потреби клієнтської частини застосунку.

У межах серверної частини, окрім запиту популярних фільмів, передбачена також реалізація функціоналу пошуку за назвою або фільтрування за жанрами. Для цього створено окремий резолвер searchMovies, який отримує параметри пошукового запиту. Його логіка аналогічна до запиту популярних фільмів, проте URL звернення змінюється на /search/movie, а параметри включають текст пошуку. Приклад коду резолвера:

```
async function searchMovies(_, { query, page, language }) {
  if (!query || query.trim() === '') {
    return new Movies({ results: [], page: 1, total_pages: 0, total_results: 0 });
  }

  const response = await axios.get(`${config.tmdbApiBaseUrl}/search/movie`, {
    params: {
      api_key: config.apiKey,
      query: query,
      page: page || 1,
      language: language || 'en-US'
    }
  });
  return new Movies(response.data);
}
```

У цьому фрагменті передбачено, що якщо пошуковий запит порожній, сервер поверне пустий результат без звернення до зовнішнього API, що дозволяє уникнути непотрібних викликів і покращує продуктивність.

Серверна частина реалізована з урахуванням принципів модульності, розділення відповідальності і масштабованості. Вона надає клієнтському застосунку простий і зрозумілий інтерфейс через GraphQL, ховаючи за собою складність роботи з зовнішнім API TMDb. Використання класів-сутностей дозволяє централізовано управляти трансформацією даних і підтримувати їх цілісність. Асинхронна обробка HTTP-запитів разом із механізмами обробки помилок забезпечує надійність та стабільність роботи. Завдяки цьому сервер можна легко розширювати, додаючи нові можливості або підтримку додаткових типів запитів без значних змін у архітектурі. Така організація коду сприяє швидкому розвитку проекту та підтримці його в довгостроковій перспективі.

4.3 Розробка фронтенду сайту

Клієнтська частина сайту-каталогу фільмів реалізована як односторінковий застосунок на базі бібліотеки React. Основна мета фронтенду — забезпечити інтерактивний і локалізований інтерфейс для пошуку та вибору фільмів, а також формування персоналізованих списків для подальших

рекомендацій. Уся логіка рендерингу побудована навколо компонента App [Додаток Б], у якому React-дерево обгортається в ApolloProvider — для виконання GraphQL-запитів — та BrowserRouter — для маршрутизації між сторінками. Додатково застосунок використовує глобальний контекст (context/appContext), який відповідає за збереження поточної мови інтерфейсу та передає її до заголовків GraphQL-запитів. Це дозволяє серверу на льоту повертати локалізовані дані, залежно від обраної користувачем мови (Рис. 4.8 та Рис. 4.9).

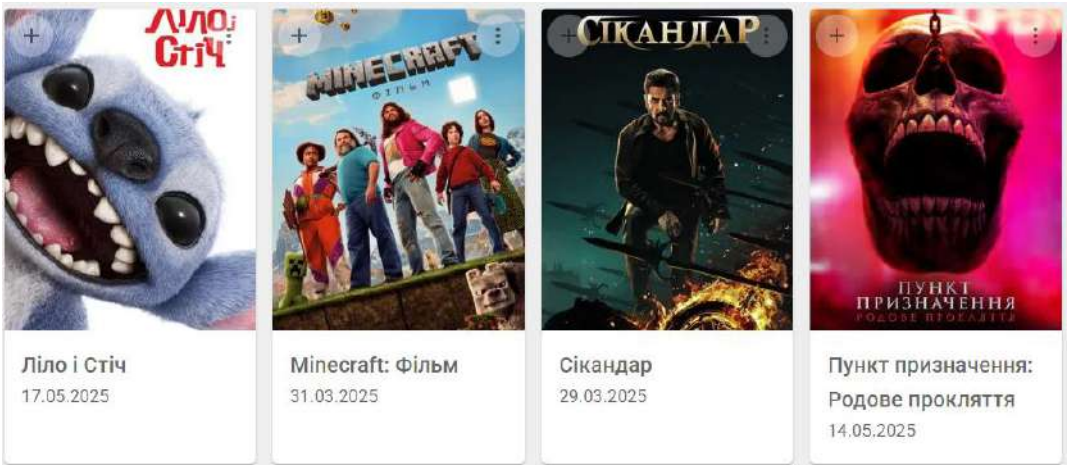

Рис. 4.8. Локалізвані фільми українською

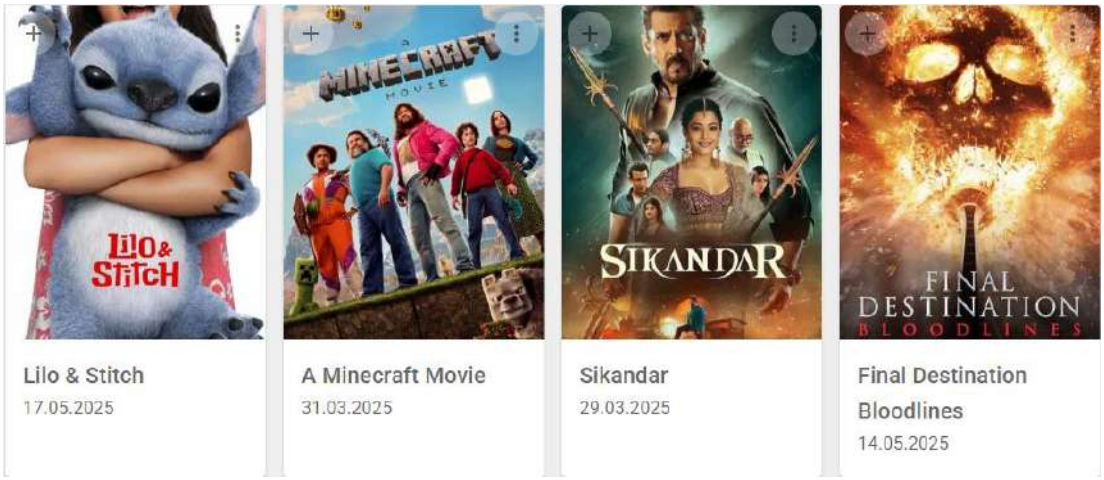

Рис. 4.9. Локалізвані фільми англійською

Візуальне оформлення інтерфейсу побудоване на основі бібліотеки Material UI. Вона забезпечує адаптивність до розміру екрана, а також уніфікований стиль компонентів. Структурно сторінка складається з фіксованої навігаційної панелі, яка розташована у файлі

components/navigation/index.js, і основного блоку вмісту. Перемикання між сторінками Home і Recommend відбувається за допомогою компонента Routes.

Головна сторінка Home є центральним елементом взаємодії з каталогом фільмів. Вона структурована з окремих частин, розміщених у папці Home/parts. Один із ключових блоків — FilterSection.js, який реалізує текстовий пошук фільмів за назвою. Пошуковий запит активується з невеликою затримкою (debounce), після чого виконується GraphQL-запит SEARCH_MOVIES, результат якого зберігається у стані компонента. У разі наявності результатів активується режим пошуку, і вміст сторінки перемикається на відповідні картки фільмів.

Поряд з пошуком, основну частину екрана займає MoviesSection.js — компонент, який відповідає за відображення сітки фільмів. У разі звичайного режиму, коли пошук неактивний, завантаження відбувається через запит MOVIES_QUERY з підтримкою пагінації. При переході в режим пошуку фільми оновлюються залежно від введеного тексту. Усі елементи фільмів виводяться у вигляді компонентів MovieCard [Додаток В], кожен з яких містить постер, назву, дату релізу, рейтинг та список жанрів. Користувач може додавати фільми до списку вибраних натисканням на картку. Після додавання картка змінює стан і вигляд, що зображено на Рис. 4.10.

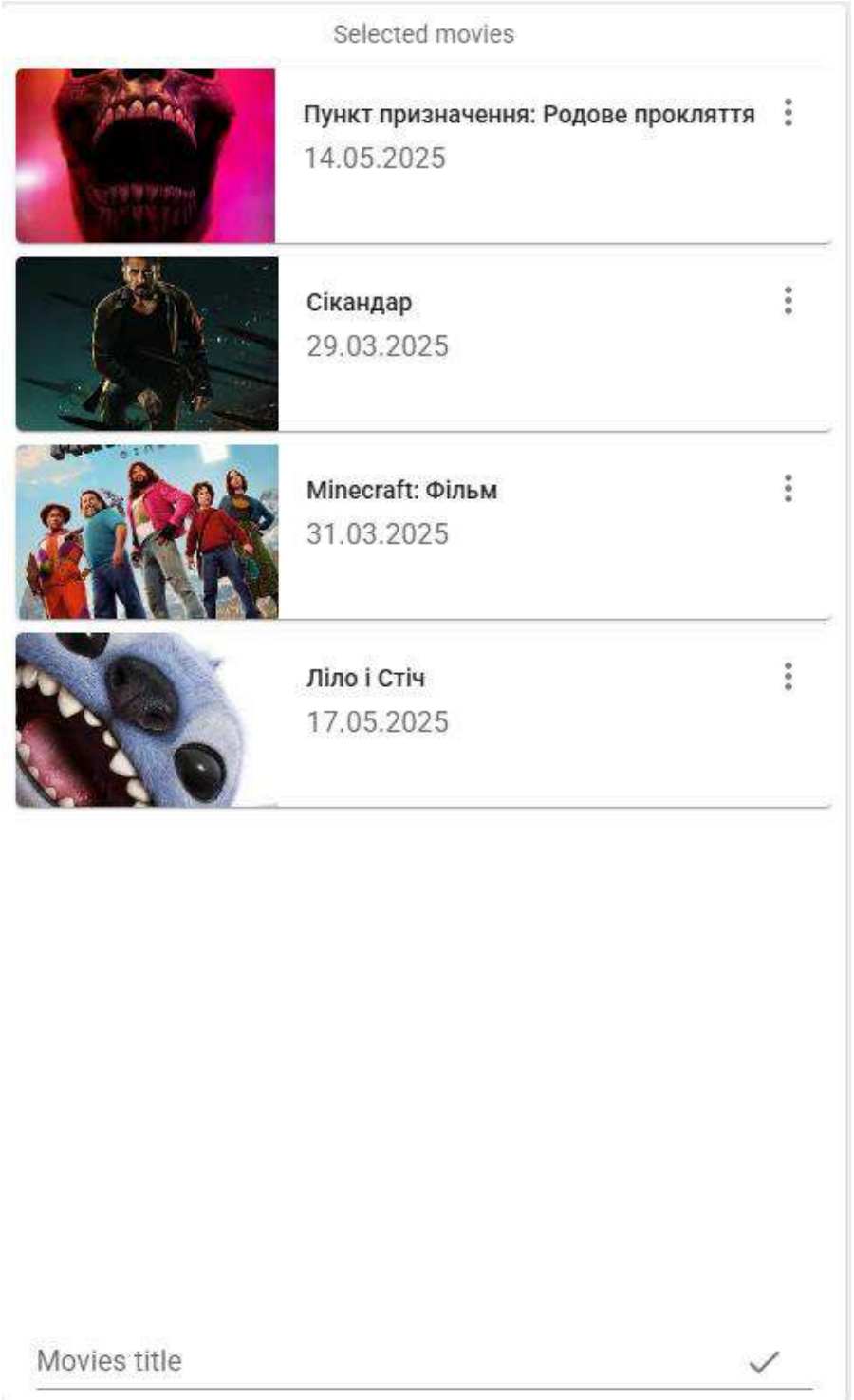

Рис. 4.10. Вигляд карток, доданих в список

Бічна панель, реалізована у файлі Sidebar.js, відображає обрані фільми у вигляді розширених карток MovieCardSelected [Додаток Г]. Кожна така картка містить повну назву фільму, рік виходу та кнопку видалення зі списку. Нижче розміщується форма для задання назви списку, після чого формується спеціальне посилання для сторінки рекомендацій, у якому передаються ідентифікатори фільмів і назва списку як параметри URL. Посилання

генерується автоматично після сабміту форми і відображається у модальному вікні з можливістю скопіювати його для подальшого використання або надсилання. Приклад такого модального вікна наведено на Рис. 4.11.

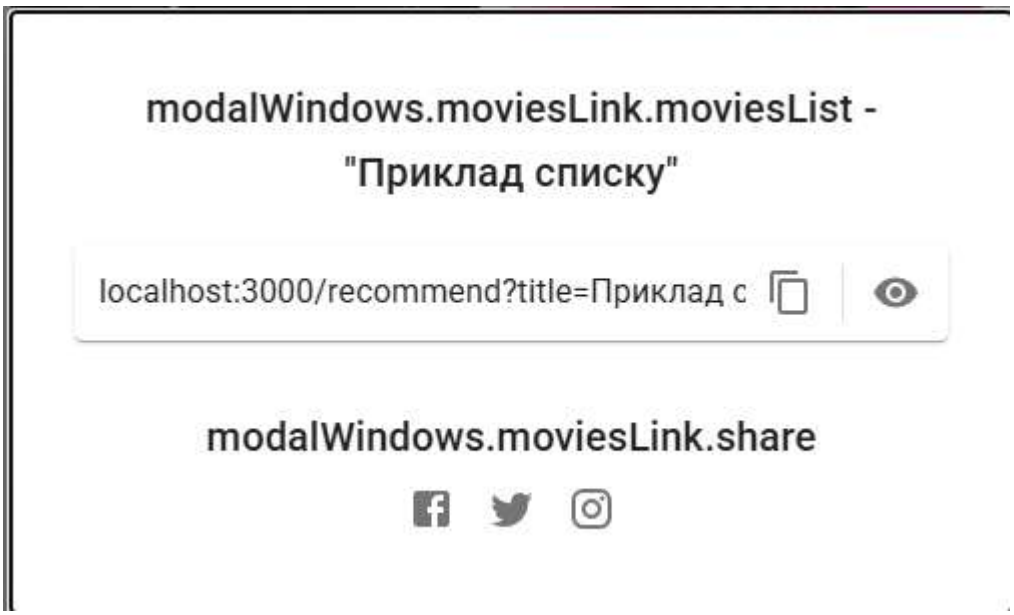

Рис. 4.11. Приклад модального вікна з посиланням

Сторінка Recommend обробляє URL-параметри, зокрема ids та title, і виконує відповідний запит до GraphQL API для отримання списку фільмів за заданими ідентифікаторами. Сторінка спрощена за структурою, не містить фільтрів або додаткових панелей, проте використовує ті самі компоненти MovieListCard для виведення списку рекомендованих фільмів у звичному стилі. У результаті, користувач отримує зручний спосіб ділитися сформованими добірками — як вручну, так і автоматично, на основі попередніх вподобань. Сторінку рекомендацій зображено на Рис. 4.12.

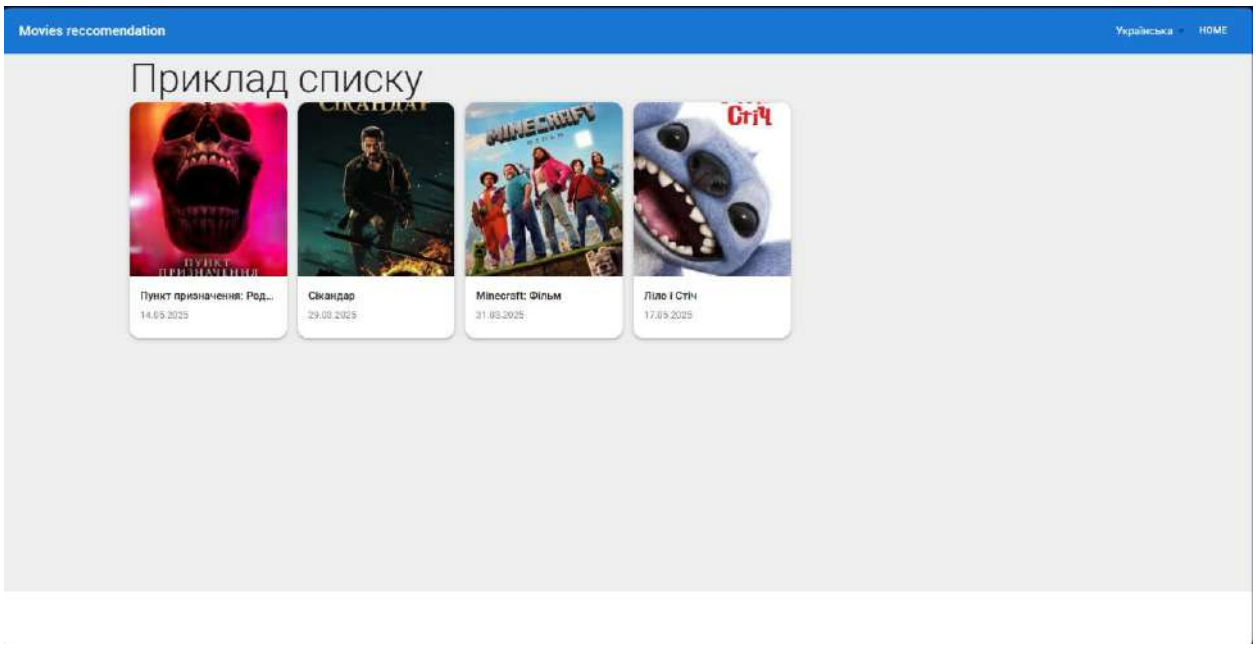

Рис. 4.12. Сторінка рекомендацій

Підтримка багатомовності реалізована через бібліотеку i18next. Локалізовані файли з текстами інтерфейсу зберігаються у форматі JSON за шляхами public/locales/uk/translation.json та public/locales/en/translation.json . Завантаження мовних ресурсів відбувається динамічно через плагін i18next-http-backend, а визначення мови користувача автоматизоване завдяки i18next-browser-languagedetector. Усі текстові елементи в інтерфейсі виводяться через хук useTranslation(), що забезпечує централізовану підтримку локалізації і спрощує подальше масштабування застосунку для інших мов.

Таким чином, фронтенд сайту побудований з урахуванням вимог до зручності користувача, підтримки локалізації, розширюваності й адаптивності. Завдяки використанню React, Apollo Client та Material UI, а також модульній структурі з поділом на сторінки, компоненти і хуки, застосунок залишається простим для розробки, підтримки та масштабування. Його архітектура дозволяє легко адаптувати клієнтську частину під інші API, змінювати структуру взаємодії або додавати нові функції без порушення існуючої логіки.

Висновки до розділу 4

У результаті розробки серверної частини було створено масштабований бекенд на базі Apollo Server з використанням GraphQL та Axios для інтеграції з API themoviedb, що забезпечує швидкий і гнучкий доступ до актуальної інформації про фільми. Серверна логіка підтримує авторизацію, роботу з профілями користувачів, вибір списків фільмів і зберігання історії переглядів, що підвищує персоналізацію сервісу.

Клієнтська частина сайту-каталогу фільмів була реалізована як сучасний веб-застосунок на React із використанням Apollo Client, Material UI та i18next для локалізації. Застосунок має інтуїтивний і мінімалістичний інтерфейс, що знижує когнітивне навантаження користувачів і забезпечує зручний пошук, перегляд карток фільмів, навігацію та роботу з формами. Логіка розподілена на компоненти, що підвищує гнучкість і підтримуваність коду. Застосунок також містить систему авторизації та роботу з профілем користувача.

Окрім цього, у рамках розробки десктопного застосунку реалізовано тривимірну сцену з кубом на OpenGL для вибору фільму, що надає інтерактивний і візуально привабливий спосіб взаємодії з каталогом. Застосунок підтримує офлайн-режим, зберігає історію виборів та має мінімалістичний інтерфейс із фокусом на простоту й зручність використання.

Загалом, комплексна реалізація серверної, клієнтської та десктопної частин забезпечила повноцінний і ефективний сервіс для пошуку, перегляду та вибору фільмів із сучасним дизайном і зручною взаємодією користувача.

ВИСНОВКИ

В результаті виконання даної роботи було досягнуто поставленої мети - створено два взаємопов’язані програмні продукти для зручного управління особистими кінодобірками: веб-застосунок для створення, пошуку та упорядкування списків фільмів, а також настільну програму з інтерактивним механізмом вибору фільму у вигляді гри. Для реалізації було використано API The Movie Database (TMDb), що забезпечило доступ до актуальної бази даних фільмів.

У ході роботи виконано такі основні завдання:

1. Розроблено веб-застосунок із функціоналом пошуку фільмів, створення списків і додавання до них контенту, що дозволяє користувачам зручно структурувати власні добірки.

2. Створено десктоп-додаток із підтримкою авторизації та реєстрації користувачів, що забезпечує персоналізацію роботи і збереження індивідуальних налаштувань.

3. Впроваджено унікальний інтерактивний механізм вибору фільму на основі випадковості — «кістку», що додає елемент гри та допомагає користувачам легко визначатися з переглядом.

Отже, розроблене програмне забезпечення забезпечує зручний та сучасний спосіб роботи з особистими кінодобірками у веб- і десктоп-форматах, підвищуючи комфорт і мотивацію користувачів до вибору фільмів, а також відкриває нові можливості для інтерактивної взаємодії з контентом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Troelsen A., Japikse P. Pro C# 9 with .NET 5: Foundational Principles and Practices in Programming. Apress, 2021.
2. Sharp J. Microsoft Visual C# Step by Step (9th Edition). Microsoft Press, 2021.
3. Petzold C. Programming Windows with C#. Microsoft Press, 2013.
4. Götz H., Haemmerle M. OpenGL Development Cookbook. Packt Publishing, 2013.
5. Angel E., Shreiner D. Interactive Computer Graphics: A Top-Down Approach with WebGL. Pearson, 2020.
6. OpenGL Architecture Review Board. The OpenGL Programming Guide (Red Book), 9th Edition. Addison-Wesley, 2017.
7. Freeman E., Robson E. Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software. O’Reilly Media, 2021.
8. Microsoft Docs. ADO.NET Overview [Електронний ресурс] – https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/
9. W3Schools. HTML, CSS, JavaScript Tutorials [Електронний ресурс] – https://www.w3schools.com
10. UX Planet. The Psychology of Choice: How to Design Better Interfaces [Електронний ресурс] – https://uxplanet.org/the-psychology-of-choice-in-ux-design
11. IMDb API Documentation [Електронний ресурс] – https://developer.imdb.com/
12. Open Movie Database API (OMDb) [Електронний ресурс] – https://www.omdbapi.com/

ДОДАТКИ

Додаток А

Рендер кубику

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Threading.Tasks;
using System.Windows.Forms;
using OpenTK;
using OpenTK.Graphics;
using OpenTK.Graphics.OpenGL;
using System.Drawing.Imaging;
using System.Net.Http;
using System.IO;
using OpenTK.GLControl;
using OpenTK.Windowing.Common;
using Timer = System.Windows.Forms.Timer;
using OpenTK.Mathematics;

namespace MovieSelection
{
    public partial class CubeRenderer : GLControl
    {
        private float rotationX = 0;
        private float rotationY = 0;
        private float rotationZ = 0;
        private float targetRotationX = 0;
        private float targetRotationY = 0;
        private float targetRotationZ = 0;
        private float rotationSpeedX = 0;
        private float rotationSpeedY = 0;
        private float rotationSpeedZ = 0;
        private bool isAnimating = false;
        private Timer animationTimer;
        private Random random;
        private int[] textureIds;
        private List<Movie> currentMovies;
        private Movie selectedMovie;
        private bool isInitialized = false;
        private float animationTime = 0;
        private const float maxAnimationTime = 3.0f;
        private const float dampingFactor = 0.98f;
```

Продовження додатку А

```
        public CubeRenderer() : base()
        {
            this.API = ContextAPI.OpenGL;
            this.Flags = ContextFlags.Default;
            this.IsEventDriven = false;
            this.Profile = ContextProfile.Compatability;
            this.SharedContext = null;
            this.TabIndex = 0;

            random = new Random();
            textureIds = new int[6];
            currentMovies = new List<Movie>();

            animationTimer = new Timer();
            animationTimer.Interval = 16;
            animationTimer.Tick += AnimationTimer_Tick;

            this.Load += CubeRenderer_Load;
        }

        private void CubeRenderer_Load(object sender, EventArgs
e)
        {
            if (isInitialized) return;

            try
            {
                MakeCurrent();

                GL.ClearColor(0.1f, 0.1f, 0.2f, 1.0f);
                GL.Enable(EnableCap.DepthTest);
                GL.Enable(EnableCap.Texture2D);
                GL.Enable(EnableCap.CullFace);
                GL.CullFace(CullFaceMode.Back);

                SetupLighting();
                CreateDefaultTextures();
                SetupPerspective();

                isInitialized = true;
```

```
                }
                catch (Exception ex)
                {
                    MessageBox.Show($"Помилка ініціалізації OpenGL:
{ex.Message}");
                }
            }

            protected override void OnLoad(EventArgs e)
            {
                base.OnLoad(e);
            }

            private void SetupLighting()
            {
                GL.Enable(EnableCap.Lighting);
                GL.Enable(EnableCap.Light0);
                GL.Enable(EnableCap.ColorMaterial);
                GL.ColorMaterial(MaterialFace.FrontAndBack,
ColorMaterialParameter.AmbientAndDiffuse);

                float[] lightPos = { 2.0f, 2.0f, 3.0f, 1.0f };
                float[] lightAmbient = { 0.4f, 0.4f, 0.4f, 1.0f };
                float[] lightDiffuse = { 0.8f, 0.8f, 0.8f, 1.0f };
                float[] lightSpecular = { 0.5f, 0.5f, 0.5f, 1.0f };

                GL.Light(LightName.Light0, LightParameter.Position,
lightPos);
                GL.Light(LightName.Light0, LightParameter.Ambient,
lightAmbient);
                GL.Light(LightName.Light0, LightParameter.Diffuse,
lightDiffuse);
                GL.Light(LightName.Light0, LightParameter.Specular,
lightSpecular);
            }

            private void CreateDefaultTextures()
            {
                GL.GenTextures(6, textureIds);

                Color[] faceColors = {
                    Color.FromArgb(220, 50, 50)
                };

                for (int i = 0; i < 6; i++)
                {
```

```
                    CreateSolidColorTexture(textureIds[i],
faceColors[i], i + 1);
                }
            }

            private void CreateSolidColorTexture(int textureId, Color
color, int sideNumber)
            {
                GL.BindTexture(TextureTarget.Texture2D, textureId);

                var bitmap = new Bitmap(512, 512);
                using (var g = Graphics.FromImage(bitmap))
                {
                    g.Clear(color);

                    using (var borderPen = new
Pen(Color.FromArgb(100, 0, 0, 0), 4))
                    {
                        g.DrawRectangle(borderPen, 2, 2, 508, 508);
                    }
                }

                LoadTextureFromBitmap(textureId, bitmap);
                bitmap.Dispose();
            }

            public async Task LoadMovieTextures(List<Movie> movies)
            {
                if (movies.Count != 6 || !isInitialized) return;

                currentMovies = movies;

                try
                {
                    MakeCurrent();

                    for (int i = 0; i < 6; i++)
                    {
                        await LoadMovieTexture(i, movies[i]);
                    }

                    if (InvokeRequired)
                    {
```

```
                        Invoke(new Action(() => {
                            if (!IsDisposed && IsHandleCreated)
                            {
                                Invalidate();
                            }
                        }));
                    }
                    else
                    {
                        if (!IsDisposed && IsHandleCreated)
                        {
                            Invalidate();
                        }
                    }
                }
                catch (Exception ex)
                {
                    MessageBox.Show($"Помилка завантаження текстур: 
{ex.Message}");
                }
            }

            private async Task LoadMovieTexture(int faceIndex, Movie 
movie)
            {
                try
                {
                    if (string.IsNullOrEmpty(movie.PhotoUrl))
                    {
                        CreateMovieTextTexture(textureIds[faceIndex], 
movie.Title);
                        return;
                    }

                    using (var httpClient = new HttpClient())
                    {
                        httpClient.Timeout = 
TimeSpan.FromSeconds(10);
                        var imageBytes = await 
httpClient.GetByteArrayAsync(movie.PhotoUrl);
                        using (var stream = new 
MemoryStream(imageBytes))
                        {
                            var bitmap = new Bitmap(stream);

LoadTextureFromBitmap(textureIds[faceIndex], bitmap);
                            bitmap.Dispose();
```

```
                        }
                    }
                }
                catch
                {
                    CreateMovieTextTexture(textureIds[faceIndex],
movie.Title);
                }
            }

            private void CreateMovieTextTexture(int textureId, string
movieTitle)
            {
                GL.BindTexture(TextureTarget.Texture2D, textureId);

                var bitmap = new Bitmap(512, 512);
                using (var g = Graphics.FromImage(bitmap))
                {
                    g.Clear(Color.FromArgb(20, 25, 40));

                    using (var borderPen = new Pen(Color.FromArgb(80,
100, 120), 6))
                    {
                        g.DrawRectangle(borderPen, 3, 3, 506, 506);
                    }

                    var rect = new RectangleF(20, 180, 472, 152);
                    var format = new StringFormat
                    {
                        Alignment = StringAlignment.Center,
                        LineAlignment = StringAlignment.Center,
                        FormatFlags = StringFormatFlags.LineLimit
                    };

                    using (var font = new Font("Arial", 18,
FontStyle.Bold))
                    using (var brush = new SolidBrush(Color.White))
                    {
                        g.DrawString(movieTitle, font, brush, rect,
format);
                    }
                }

                LoadTextureFromBitmap(textureId, bitmap);
                bitmap.Dispose();
```

```
            }

            private void LoadTextureFromBitmap(int textureId, Bitmap
bitmap)
            {
                GL.BindTexture(TextureTarget.Texture2D, textureId);

                var resized = new Bitmap(bitmap, 512, 512);
                var data = resized.LockBits(new Rectangle(0, 0, 512,
512),
                    ImageLockMode.ReadOnly,
System.Drawing.Imaging.PixelFormat.Format32bppArgb);

                GL.TexImage2D(TextureTarget.Texture2D, 0,
PixelInternalFormat.Rgba, 512, 512, 0,
                    OpenTK.Graphics.OpenGL.PixelFormat.Bgra,
PixelType.UnsignedByte, data.Scan0);

                resized.UnlockBits(data);
                resized.Dispose();

        GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureMinFilter, (int)TextureMinFilter.Linear);
                GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureMagFilter, (int)TextureMagFilter.Linear);
                GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureWrapS, (int)TextureWrapMode.ClampToEdge);
                GL.TexParameter(TextureTarget.Texture2D,
TextureParameterName.TextureWrapT, (int)TextureWrapMode.ClampToEdge);
            }

            protected override void OnPaint(PaintEventArgs e)
            {
                if (!isInitialized || IsDisposed || !IsHandleCreated)
                {
                    base.OnPaint(e);
                    return;
                }

                try
                {
                    MakeCurrent();

                    GL.Clear(ClearBufferMask.ColorBufferBit |
ClearBufferMask.DepthBufferBit);
```

```
                    SetupPerspective();

                    GL.LoadIdentity();
                    GL.Translate(0.0f, 0.0f, -6.0f);
                    GL.Rotate(rotationX, 1.0f, 0.0f, 0.0f);
                    GL.Rotate(rotationY, 0.0f, 1.0f, 0.0f);
                    GL.Rotate(rotationZ, 0.0f, 0.0f, 1.0f);

                    DrawCube();

                    SwapBuffers();
                }
                catch (Exception ex)
                {
                    Console.WriteLine($"Помилка малювання:
{ex.Message}");
                }

                base.OnPaint(e);
            }

            private void SetupPerspective()
            {
                GL.MatrixMode(MatrixMode.Projection);
                GL.LoadIdentity();

                if (Width > 0 && Height > 0)
                {
                    float aspect = (float)Width / Height;
                    float fovy = 45.0f;
                    float fH = (float)Math.Tan(fovy / 360.0f *
Math.PI) * 0.1f;
                    float fW = fH * aspect;
                    GL.Frustum(-fW, fW, -fH, fH, 0.1f, 100.0f);
                }

                GL.MatrixMode(MatrixMode.Modelview);
            }

            private void DrawCube()
            {
```

```
                float size = 1.2f;

                GL.BindTexture(TextureTarget.Texture2D,
textureIds[0]);
                GL.Begin(PrimitiveType.Quads);
                GL.Normal3(0.0f, 0.0f, 1.0f);
                GL.TexCoord2(0.0f, 0.0f); GL.Vertex3(-size, -size,
size);
                GL.TexCoord2(1.0f, 0.0f); GL.Vertex3(size, -size,
size);
                GL.TexCoord2(1.0f, 1.0f); GL.Vertex3(size, size,
size);
                GL.TexCoord2(0.0f, 1.0f); GL.Vertex3(-size, size,
size);
                GL.End();

                GL.BindTexture(TextureTarget.Texture2D,
textureIds[1]);
                GL.Begin(PrimitiveType.Quads);
                GL.Normal3(0.0f, 0.0f, -1.0f);
                GL.TexCoord2(1.0f, 0.0f); GL.Vertex3(-size, -size, -
size);
                GL.TexCoord2(1.0f, 1.0f); GL.Vertex3(-size, size, -
size);
                GL.TexCoord2(0.0f, 1.0f); GL.Vertex3(size, size, -
size);
                GL.TexCoord2(0.0f, 0.0f); GL.Vertex3(size, -size, -
size);
                GL.End();

                GL.BindTexture(TextureTarget.Texture2D,
textureIds[2]);
                GL.Begin(PrimitiveType.Quads);
                GL.Normal3(0.0f, 1.0f, 0.0f);
                GL.TexCoord2(0.0f, 1.0f); GL.Vertex3(-size, size, -
size);
                GL.TexCoord2(0.0f, 0.0f); GL.Vertex3(-size, size,
size);
                GL.TexCoord2(1.0f, 0.0f); GL.Vertex3(size, size,
size);
                GL.TexCoord2(1.0f, 1.0f); GL.Vertex3(size, size, -
size);
                GL.End();

                GL.BindTexture(TextureTarget.Texture2D,
textureIds[3]);
                GL.Begin(PrimitiveType.Quads);
                GL.Normal3(0.0f, -1.0f, 0.0f);
                GL.TexCoord2(1.0f, 1.0f); GL.Vertex3(-size, -size, -
size);
                GL.TexCoord2(0.0f, 1.0f); GL.Vertex3(size, -size, -
size);
                GL.TexCoord2(0.0f, 0.0f); GL.Vertex3(size, -size,
size);
                GL.TexCoord2(1.0f, 0.0f); GL.Vertex3(-size, -size,
size);
```

```
                GL.Normal3(1.0f, 0.0f, 0.0f);
                GL.TexCoord2(1.0f, 0.0f); GL.Vertex3(size, -size, -
size);
                GL.TexCoord2(1.0f, 1.0f); GL.Vertex3(size, size, -
size);
                GL.TexCoord2(0.0f, 1.0f); GL.Vertex3(size, size,
size);
                GL.TexCoord2(0.0f, 0.0f); GL.Vertex3(size, -size,
size);
                GL.End();

                GL.BindTexture(TextureTarget.Texture2D,
textureIds[5]);
                GL.Begin(PrimitiveType.Quads);
                GL.Normal3(-1.0f, 0.0f, 0.0f);
                GL.TexCoord2(0.0f, 0.0f); GL.Vertex3(-size, -size, -
size);
                GL.TexCoord2(1.0f, 0.0f); GL.Vertex3(-size, -size,
size);
                GL.TexCoord2(1.0f, 1.0f); GL.Vertex3(-size, size,
size);
                GL.TexCoord2(0.0f, 1.0f); GL.Vertex3(-size, size, -
size);
                GL.End();
            }

            public void ThrowCube()
            {
                if (isAnimating || currentMovies.Count != 6) return;

                isAnimating = true;
                animationTime = 0;

                rotationSpeedX = random.Next(8, 20);
                rotationSpeedY = random.Next(8, 20);
                rotationSpeedZ = random.Next(8, 20);

                targetRotationX = rotationX + random.Next(720, 1440);
                targetRotationY = rotationY + random.Next(720, 1440);
                targetRotationZ = rotationZ + random.Next(720, 1440);

                animationTimer.Start();
            }

            private void AnimationTimer_Tick(object sender, EventArgs
e)
            {
                animationTime += 0.016f;

                float progress = animationTime / maxAnimationTime;
```

```
                float easeOut = 1.0f - (float)Math.Pow(1.0f -
progress, 3);

                if (progress >= 1.0f)
                {
                    animationTimer.Stop();
                    isAnimating = false;

                    rotationX = targetRotationX % 360;
                    rotationY = targetRotationY % 360;
                    rotationZ = targetRotationZ % 360;

                    int selectedFace = GetVisibleFace();
                    selectedMovie = currentMovies[selectedFace];

                    MessageBox.Show($"Випав фільм:
{selectedMovie.Title}!",
                        "Результат", MessageBoxButtons.OK,
MessageBoxIcon.Information);
                }
                else
                {
                    rotationSpeedX *= dampingFactor;
                    rotationSpeedY *= dampingFactor;
                    rotationSpeedZ *= dampingFactor;

                    rotationX += rotationSpeedX * (1.0f - easeOut);
                    rotationY += rotationSpeedY * (1.0f - easeOut);
                    rotationZ += rotationSpeedZ * (1.0f - easeOut);
                }

                if (!IsDisposed && IsHandleCreated)
                {
                    Invalidate();
                }
            }

            protected override void OnResize(EventArgs e)
            {
                base.OnResize(e);

                if (Width > 0 && Height > 0 && isInitialized &&
IsHandleCreated)
                {
```

```
                    try
                    {
                        MakeCurrent();
                        GL.Viewport(0, 0, Width, Height);
                        SetupPerspective();
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine($"Помилка зміни розміру:
{ex.Message}");
                    }
                }
            }

            private int GetVisibleFace()
            {
                float normalizedRotX = ((rotationX % 360) + 360) %
360;
                float normalizedRotY = ((rotationY % 360) + 360) %
360;

                Vector3 cameraDir = new Vector3(0, 0, 1);

                Vector3[] faceNormals = {
                    new Vector3(0, 0, 1),
                    new Vector3(0, 0, -1),
                    new Vector3(0, 1, 0),
                    new Vector3(0, -1, 0),
                    new Vector3(1, 0, 0),
                    new Vector3(-1, 0, 0)
                };

                float maxDot = float.MinValue;
                int mostVisibleFace = 0;

                for (int i = 0; i < 6; i++)
                {
                    Vector3 rotatedNormal =
RotateVector(faceNormals[i], normalizedRotX, normalizedRotY,
rotationZ);
                    float dot = Vector3.Dot(rotatedNormal,
cameraDir);

                    if (dot > maxDot)
                    {
```

```
                        maxDot = dot;
                        mostVisibleFace = i;
                    }
                }

                return mostVisibleFace;
            }

            private Vector3 RotateVector(Vector3 vector, float rotX,
float rotY, float rotZ)
            {
                float radX = rotX * (float)Math.PI / 180.0f;
                float radY = rotY * (float)Math.PI / 180.0f;
                float radZ = rotZ * (float)Math.PI / 180.0f;

                float cosX = (float)Math.Cos(radX), sinX =
(float)Math.Sin(radX);
                float cosY = (float)Math.Cos(radY), sinY =
(float)Math.Sin(radY);
                float cosZ = (float)Math.Cos(radZ), sinZ =
(float)Math.Sin(radZ);

                Vector3 temp = new Vector3(
                    vector.X,
                    vector.Y * cosX - vector.Z * sinX,
                    vector.Y * sinX + vector.Z * cosX
                );

                temp = new Vector3(
                    temp.X * cosY + temp.Z * sinY,
                    temp.Y,
                    -temp.X * sinY + temp.Z * cosY
                );

                return new Vector3(
                    temp.X * cosZ - temp.Y * sinZ,
                    temp.X * sinZ + temp.Y * cosZ,
                    temp.Z
                );
            }

            protected override void Dispose(bool disposing)
            {
                if (disposing)
```

```
            {
                animationTimer?.Stop();
                animationTimer?.Dispose();

                if (textureIds != null && isInitialized)
                {
                    try
                    {
                        MakeCurrent();
                        GL.DeleteTextures(6, textureIds);
                    }
                    catch
                    {
                    }
                }
            }
            base.Dispose(disposing);
        }
    }
}
```

Додаток Б

Стартовий компонент клієнту

```
import { useContext } from 'react';
import {BrowserRouter, Routes, Route} from 'react-router-dom'
import {CssBaseline, Container, Box} from '@mui/material'

import {Navigation} from './components';
import {Home, Recommend} from './pages'

import {ApolloClient, InMemoryCache, ApolloProvider, HttpLink,
ApolloLink, from} from '@apollo/client'
import { AppContext } from './context/appContext';

function App() {
    const { state } = useContext(AppContext)
    const httpLink = new HttpLink({uri:
'http://localhost:4000/'})
    const localeMiddleware = new ApolloLink((operation,
forward)=>{
    const customHeaders =
operation.getContext().hasOwnProperty('headers') ?
operation.getContext().headers : {};

        operation.setContext({
            headers: {
                ...customHeaders,
                locale: state.locale
            }
        })
        return forward(operation)
    })

    const client = new ApolloClient({
        link: from([localeMiddleware, httpLink]),
        cache: new InMemoryCache()
    })

    return (
        <ApolloProvider client={client}>
            <BrowserRouter>
                <CssBaseline/>
                <Navigation/>
                <Box
                    sx={{
                    backgroundColor: (theme) =>
theme.palette.mode === 'light'
                        ? theme.palette.grey[200]
                        : theme.palette.grey[200],
                    flexGrow: 1,
                    minHeight: 'calc(100vh - 150px)'
                }}>
                    <Container maxWidth='xl'>
                        <Routes>
                            <Route path='home' element={< Home
/>}/>
                            <Route path='' element={< Home />}/>
```

Продовження додатку Б

```
                                    <Route path='recommend' element={<
Recommend />}/>
                                </Routes>
                            </Container>
                        </Box>
                    </BrowserRouter>
                </ApolloProvider>
            );
        }

        export default App;
```

Додаток В

Компонент картки на сайті

```
import * as React from 'react';
import {styled} from '@mui/material/styles';
import {
    Card,
    CardMedia,
    CardContent,
    Typography,
    IconButton
} from '@mui/material';
import AddIcon from '@mui/icons-material/Add';
import PropTypes from 'prop-types'
import {CardMenu} from '../';
import { useTranslation } from 'react-i18next';

const StyledCardContent = styled(CardContent)(({ theme })=>({
    "&:last-child":{
        paddingBottom: '16px',
        display:'flex',
        flexDirection: 'column',
        justifyContent: 'flex-end',
    }
}))

const StyledIconButton = styled(IconButton)(({theme})=>({
    position: 'absolute',
    top: '5px',
    left: '5px',
    backgroundColor: 'rgba(255,255,255, 0.3)',
    '&:hover':{
        backgroundColor: 'rgba(255,255,255, 0.1)',
    }
}))

const MovieCard = ({movie, onCardSelect}) => {

    const { t } = useTranslation()

    const options = [
        {
            title: t('buttons.add'),
            onclick: onCardSelect
        },
        {
            title:t('buttons.close'),
            onclick: ()=>{}
        }]
    return (
        <Card sx={{
            position: 'relative',
            height: '100%',
        }}>
            <CardMenu items={options}/>
            <CardMedia
                height='300px'
```

Продовження додатку В

```
                        component="img"
                        image={movie.image || movie.posterPath}
                        alt={movie.title}/>
                    <StyledCardContent>
                        <StyledIconButton onClick={onCardSelect}>
                            <AddIcon/>
                        </StyledIconButton>
                        <Typography variant="h6" color="text.secondary">
                            {movie.title}
                        </Typography>
                        <Typography variant="subtitle1"
color="text.secondary">

                            {new
Date(movie.releaseDate).toLocaleDateString()}
                        </Typography>
                    </StyledCardContent>
                </Card>
            );
        }

        MovieCard.propTypes  = {
            movie: PropTypes.shape({
                title: PropTypes.string.isRequired,
                releaseDate: PropTypes.string.isRequired,
                image: PropTypes.string
            }).isRequired,
            onCardSelect: PropTypes.func.isRequired
        }

        export default MovieCard
```

Додаток Г

Компонент картки в сайдбарі

```
     import React from 'react'
     import PropTypes from 'prop-types'
     import {Box, Card, CardContent, CardMedia, Typography} from
'@mui/material';
     import CardMenu from '../CardMenu';
     import { useTranslation } from 'react-i18next';

     function MovieCardSelected({movie, onCardDelete}) {

         const { t } = useTranslation()

         const options = [{
             title: t('buttons.remove'),
             onclick: onCardDelete
         }]

         return (
           <Card sx={{ display: 'flex', position: "relative", height:
100 }}>
             <CardMenu items={options} />
             <CardMedia
               component="img"
               sx={{ width: 151, height: 100 }}
               image={movie.image || movie.posterPath}
               alt={movie.title}
             />
             <Box sx={{ display: 'flex'}}>
                 <CardContent sx={{ flex: '1 0 auto', textAlign:
'start' }}>
                 <Typography component="div" variant="subtitle2">
                     {movie.title}
                 </Typography>
                 <Typography variant="subtitle1"
color="text.secondary" component="div">
                     {new
Date(movie.releaseDate).toLocaleDateString()}
                 </Typography>
                 </CardContent>
                 <CardContent sx={{ display: 'flex', flexDirection:
'column'}}>
                     {movie.genres?.length ? (
                         <Typography variant='subtitle2'
color="text.secondary" component="div">
                             {movie.genres[0].name}
                     </Typography>
                     ): null
                     }

                 </CardContent>
             </Box>

           </Card>

       );
```

```
}

MovieCardSelected.propTypes = {
    movie: PropTypes.shape({
        image: PropTypes.string.isRequired,
        title: PropTypes.string.isRequired,
        releaseDate: PropTypes.string,
        genres: PropTypes.arrayOf( PropTypes.shape({
            id: PropTypes.number.isRequired,
            name: PropTypes.string.isRequired
        })),
        runtime: PropTypes.number,
    }).isRequired,
    onCardDelete: PropTypes.func.isRequired,
}

export default MovieCardSelected
```