Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система таргетингу за інтересами для платформи самостійного видавництва художньої літератури

(тема)

Виконав:
студент 4 курсу, групи ПЗПІ-20-4

Онищенко М.Г.
(прізвище, ініціали)

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

Керівник доц. кафедри ПІ Ворочек О.Г.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри ____________ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна Інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________
(підпис)
«___» _________ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Онищенку Микиті Геннадійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Програмна система таргетингу за інтересами для платформи самостійного видавництва художньої літератури

Затверджена наказом по університету від 20.05.2024 р. № 471 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 17.05.2024

3. Вихідні дані до роботи розробити програмну систему для формування персоналізованих рекомендацій, читання книг по главам онлайн та самостійного видавництва книг авторизованими користувачами; реалізувати веб-API, використовуючи технології: ASP.NET Core, .NET 8 та мову програмування C#; реалізувати клієнтський веб-додаток, використовуючи технології: React.js, Redux, Material UI та мову програмування JS.

4. Перелік питань, що потрібно опрацювати в роботі

Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, огляд користувацького інтерфейсу, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	10.02.2024	*виконано*
2	Створення специфікації ПЗ	25.02.2024	*виконано*
3	Проектування ПЗ	30.02.2024	*виконано*
4	Розробка ПЗ	01.03.2024	*виконано*
5	Тестування ПЗ	30.04.2024	*виконано*
6	Оформлення пояснювальної записки	04.05.2024	*виконано*
7	Підготовка презентації та доповіді	05.05.2024	*виконано*
8	Попередній захист	08.06.2024	*виконано*
9	Нормоконтроль, рецензування	07.06.2024	*виконано*
10	Здача роботи у електронний архів	14.06.2024	*виконано*
11	Допуск до захисту у зав. кафедри	17.06.2024	*виконано*

Дата видачі завдання 20 грудня 2023р.

Студент ____________________ Онищенко М.Г.
(підпис)

Керівник роботи ________________ доц. кафедри ПІ Ворочек О.Г.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи бакалавра: 70 с., 18 рис., 6 дод., 8 джерел.

РЕКОМЕНДАЦІЙНА СИСТЕМА, JAVASCRIPT, REACT, САМОСТЙНЕ ВИДАВНИЦТВО, МАШИННЕ НАВЧАННЯ, ML.NET, API, ASP.NET, КНИГИ, ЧИТАННЯ, ВЕБ-ДОДАТОК.

Об’єкт розробки – програмна система таргетингу за інтересами для платформи самостійного видавництва художньої літератури.

Об’єктами дослідження є персоналізовані рекомендації та читання і самостійне видавництво онлайн.

Методи вирішення завдання – аналіз та моделювання предметної області, концептуальне моделювання, розробка серверного додатку, реалізація рекомендаційної системи на основі машинного навчання, створення веб-додатку, застосування популярних серверних .NET технологій та популярних веб-технологій на основі JS.

У результаті було розроблено програмну система, що дозволяє публікувати книги по главам, а також читати їх та отримувати персоналізовані рекомендації. Дана система дозволяє формувати та підтримувати читацьку спільноту на основі втримання уваги читача. Вона створена з метою доповнити типовий видавницький та читацький функціонал, а також віднайти зиск із накопиченої історії взаємодії з платформою для покращення виконання нею своїх задач.

RECOMMENDATION SYSTEM, JAVASCRIPT, REACT, SELF-PUBLISHING, MACHINE LEARNING, ML.NET, API, ASP.NET, BOOKS, READING, WEB APPLICATION.

The object of development is the interest-targeting software system for self-publishing fiction platform.

The objects of research are personalized recommendations as well as reading and self-publishing online.

The methods of solving the problem are: analysis and modeling of the subject area, conceptual modeling, development of a server application, implementation of a recommendation system based on machine learning, creation of a web application, employment of popular server-side .NET technologies and popular JS-based web technologies.

As a result, a software system has been implemented to allow publishing books by chapters, as well as reading them and receiving personalized recommendations. This system helps forming and maintaining a reader community based on keeping reader's attention. It was created to complement a typical publishing and reading functionality, as well as to benefit from the accumulated history of interaction with the platform to improve its effectiveness.

Я, Онищенко Микита Геннадійович , студент гр. ПЗПІ-20-4, здобувач вищої освіти на першому (бакалаврському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Програмна система таргетингу за інтересами для платформи самостійного видавництва художньої літератури», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Усі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови до допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

Вступ 8
1 Аналіз предметної галузі 10
1.1 Аналіз предметної галузі 10
1.1.1 Дослідження самостійного видавництва 10
1.1.2 Дослідження аналогів на ринку 11
1.2 Виявлення проблем та актуалізація рішень 14
1.3 Постановка задач 15
2 Формування вимог до програмної системи 16
2.1 Постановка мети 16
2.2 Загальний опис системи 17
2.3 Основний функціонал системи 17
2.4 Загальні обмеження 18
2.5 Припущення та залежності 19
3 Архітектура та проєктування програмного забезпечення 20
3.1 Проєктування бази даних 20
3.2 Проєктування серверної частини програмної системи 21
3.3 Проєктування рекомендаційної системи 22
3.4 Проєктування клієнтської частини програмної системи 27
4 Опис прийнятих програмних рішень 29
4.1 Прийняті рішення серверної частини 29
4.1.1 Прийняті рішення доступу до даних 29
4.1.2 Прийняті рішення API 31
4.1.3 Прийняті рішення машинного навчання 33
4.2 Прийняті рішення клієнтської частини 36
5 Тестування програмного забезпечення 40
5.1 Модульне тестування 40
5.2 Інтеграційне тестування 42
6 Огляд користувацького інтерфейсу 46
6.1 Огляд сторінок читача 46
6.2 Огляд сторінок автора 49
Висновки 52

Перелік джерел посилання 53
Додаток А Звіт результатів перевірки на унікальність у базі ХНУРЕ 54
Додаток Б Слайди презентації 55
Додаток В Діаграма прецедентів 66
Додаток Г Діаграма пакетів.. 67
Додаток Д Діаграма послідовності.. 69
Додаток Е Діаграма розгортання 70

ВСТУП

У сучасному цифровому світі, позначеному безпрецедентним поширенням контенту, незамінність персоналізованих систем рекомендацій стала визначальним аспектом сучасного користувацького досвіду. Це яскраво видно на прикладі TikTok, гіганта соціальних мереж, який може похвалитися приголомшливою користувацькою базою, що налічує понад 1 мільярд активних користувачів. Оскільки понад 83% користувачів TikTok беруть активну участь у створенні контенту платформи, щоденне завантаження 34 мільйонів[1] відео не лише свідчить про динамічний характер платформи, але й посилює виклики, з якими стикаються користувачі при навігації в цьому величезному резервуарі різноманітного контенту. Ці виклики вирішують алгоритми TikTok.

Проте корисність систем персоналізованих рекомендацій виходить далеко за межі платформ соціальних мереж і пронизує всі аспекти послуг на основі контенту, дивуючи індустрії своїм трансформаційним впливом. Так, у сфері самостійного видавництва книг онлайн, де за увагу читачів змагається широкий спектр доволі тривалих історій, впровадження систем персоналізованих рекомендацій дозволяє ефективно аналізувати індивідуальні вподобання. Утворене в процесі тонке розуміння дозволяє платформі не лише підвищує задоволеність користувачів, відчутно звужуючи коло пошуків, а й сприяє відкриттю "прихованих перлин" і нових перспективних авторів.

Дана робота присвячується вивченню механізмів колаборативної фільтрації, притаманних сучасним персоналізованим рекомендаційним системам. Вивчення поширюватиметься на сферу самостійного видавництва, представляючи реальні приклади платформ-аналогів, які ефективно використовують колаборативну фільтрацію для оптимізації залучення користувачів та доповнення ядра свого типового функціоналу, що полягає в публікації та читанні.

Метою проєкту є створення програмної системи, яка організує роботу хабу поширення художньої літератури для зацікавленої аудиторії. Основні функції включають пошук та рекомендації читачам, читання книжних глав, а також дозволяють авторам додавати та публікувати свої твори. Перш за все, він надає

можливість ефективного підбору релевантного контенту, забезпечуючи читачам доступ до різноманітних творів в умовах інформаційного перенасичення. Для повноцінності демонстрації, програмний продукт включатиме функціонал для самостійної публікації художніх творів, обходячи потребу в традиційних видавництвах і сприяючи агрегації контенту та формуванню ком’юніті.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

1.1.1 Дослідження самостійного видавництва

Загальновідомо, що традиційне видавництво становить перепону для багатьох авторів через його складний та багатоступінчастий процес.

Від пошуку агента, який взявся б представляти їхні інтереси, до отримання видавничого контракту, цей шлях може бути довгим, невпевненим і непередбачуваним. У цій моделі видавець виступає як фільтр, відсіюючи та відхиляючи безліч потенційних авторів, що часто призводить до обмеження різноманітності літературного ринку та ігнорування новаторських творчих голосів [2].

Крім того, автори, які укладають контракт з видавцем, часто стикаються з обмеженнями в їхній творчій свободі, змушені боротися за право на власний творчий внесок у книгу, а також отримують низькі роялті, які не завжди відображають справжню цінність їхньої роботи [2].

У відповідь на зміни у тенденціях розвитку літературної індустрії, що характеризуються зсувом у бік цифрового споживання та більшої доступності, запропонована онлайн платформа для самостійного видавництва і рекомендаційна система виступають як ключові складові у демократизації видавничого процесу. Платформа, надаючи авторам-початківцям зручний та інтуїтивно зрозумілий інтерфейс для створення, форматування та публікації своїх літературних творів, дозволяє поділитися їхніми історіями безпосередньо з глобальним споживачем.

Відповідно налаштована рекомендаційна система допомагає не тільки прибрати з процесу, але й замінити суб'єктивний і часто непрозорий характер традиційної видавничої індустрії.

В той же час вона робить більш справедливим розподіл уваги між літературними творами, орієнтуючись на дані про читацькі вподобання та метрики залучення. Такий підхід не лише сприяє більш інклюзивному та різноманітному спектру авторського голосу, але й сприяє розвитку динамічної та інтерактивної літературної екосистеми.

Що ж до самостійного видавництва, то феномен незалежних творців контенту супроводжується насамперед збереженням індивідуальності. Наприклад, для створення відеоконтенту та споживання цифрових медіа [3], внутрішні мотивації, такі як задоволення та соціальна взаємодія, відіграють вирішальну роль у підтримці постійної залученості автора.

Водночас, поява складних алгоритмів на цифрових платформах полегшила пошук і поширення контенту, пристосованого до індивідуальних уподобань, ще більше стимулюючи зростання незалежних творців контенту, розширюючи їхню аудиторію та підвищуючи їхню затребуваність.

Таким чином, сучасні технології створення контенту підтримують як внутрішнє прагнення до творчого самовираження, так і зовнішнє прагнення до визнання та винагороди, пропонуючи динамічний простір для людей, які можуть орієнтуватися в своїх ролях творців і споживачів цифрового контенту.

1.1.2 Дослідження аналогів на ринку

Найбільш помітним представником самостійного видавництва на англомовному ринку виступає Wattpad. Загалом, це відносно проста платформа, що наслідує принципи соціальних мереж для побудови свого головного функціоналу.

Представляючи книги сторінками, а глави – користувацькими постами, дана платформа підлаштовується до читачів, звичних до щоденного проведення часу в інтернеті, зокрема із застосуванням смартфону. В основі такого підходу лежить мінімалізм, responsive дизайн, безперервність та скролінг (див. рисунок 1.1).

Проте нічого з переліченого не видається достатнім, якщо платформа не здатна надовго втримати увагу користувача. Користувачу необхідно постійно знаходити підходящий контент, аби продовжувати користуватися платформою. Для задоволення потреби Wattpad застосовує систему персоналізованих рекомендацій, що постійно покращуються на основі аналізу історії взаємодій.

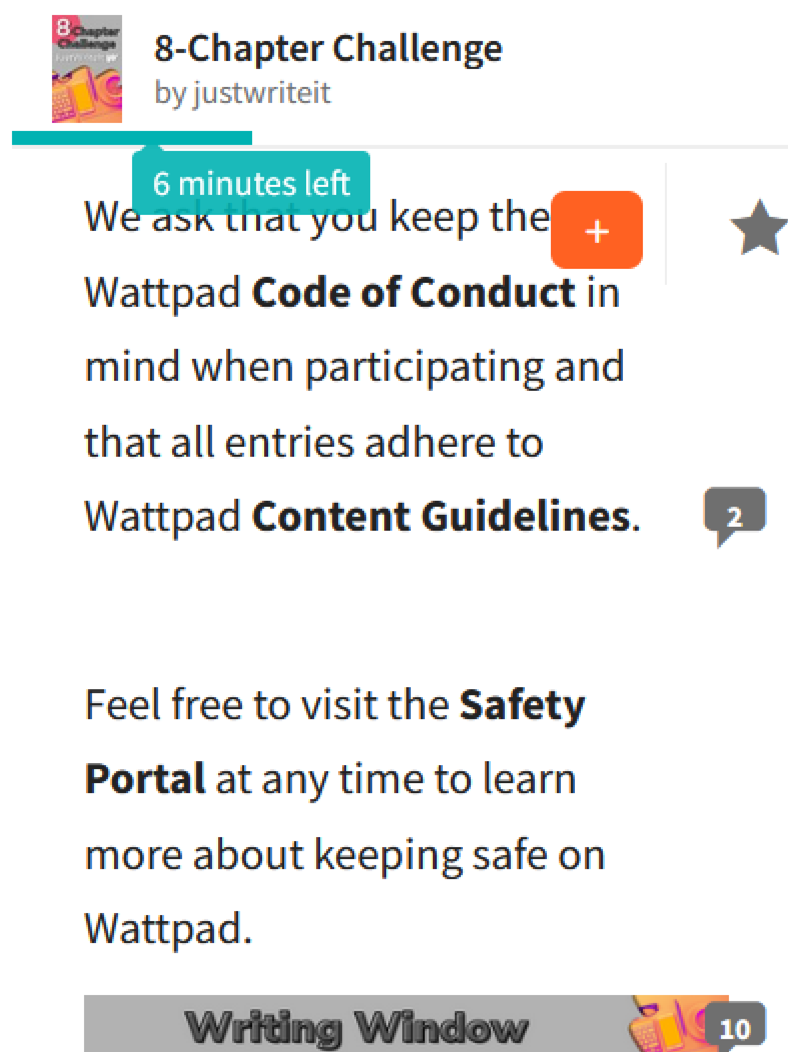


Рисунок 1.1 – Мобільний читацький інтерфейс Wattpad

Тоді слабкою стороною платформи можна вважати недостатнє охоплення рекомендаційною системою нових авторів, які тільки набирають популярність. Рекомендаційна система Wattpad фокусується на творах з уже сформованою читацькою базою, які мають багато переглядів та відгуків за весь час перебування на платформі. Тобто вона не бере до уваги тенденцій. В ній відсутній функціонал, який би формував нові тренди всередині спільноти.

Водночас, платформа містить усі необхідні для самостійного видавництва інструменти. Створивши обліковий запис, автор отримує доступ до створення сторінок своїх книг, оформлення та подальшої публікації глав. По ходу часу він також отримуватиме зворотній зв’язок у вигляді статистики та коментарів.

Другий релевантний аналог називається RoyalRoad. Попри менший розмір авдиторії, він мало поступається конкурентам активністю своєї спільноти. Це також проста платформа, але вона не наслідує сучасні тенденції. Натомість вона зберігає та практикує досвід автентичних, але застарілих агрегаторів, хоч і відійшла від форумного підходу.

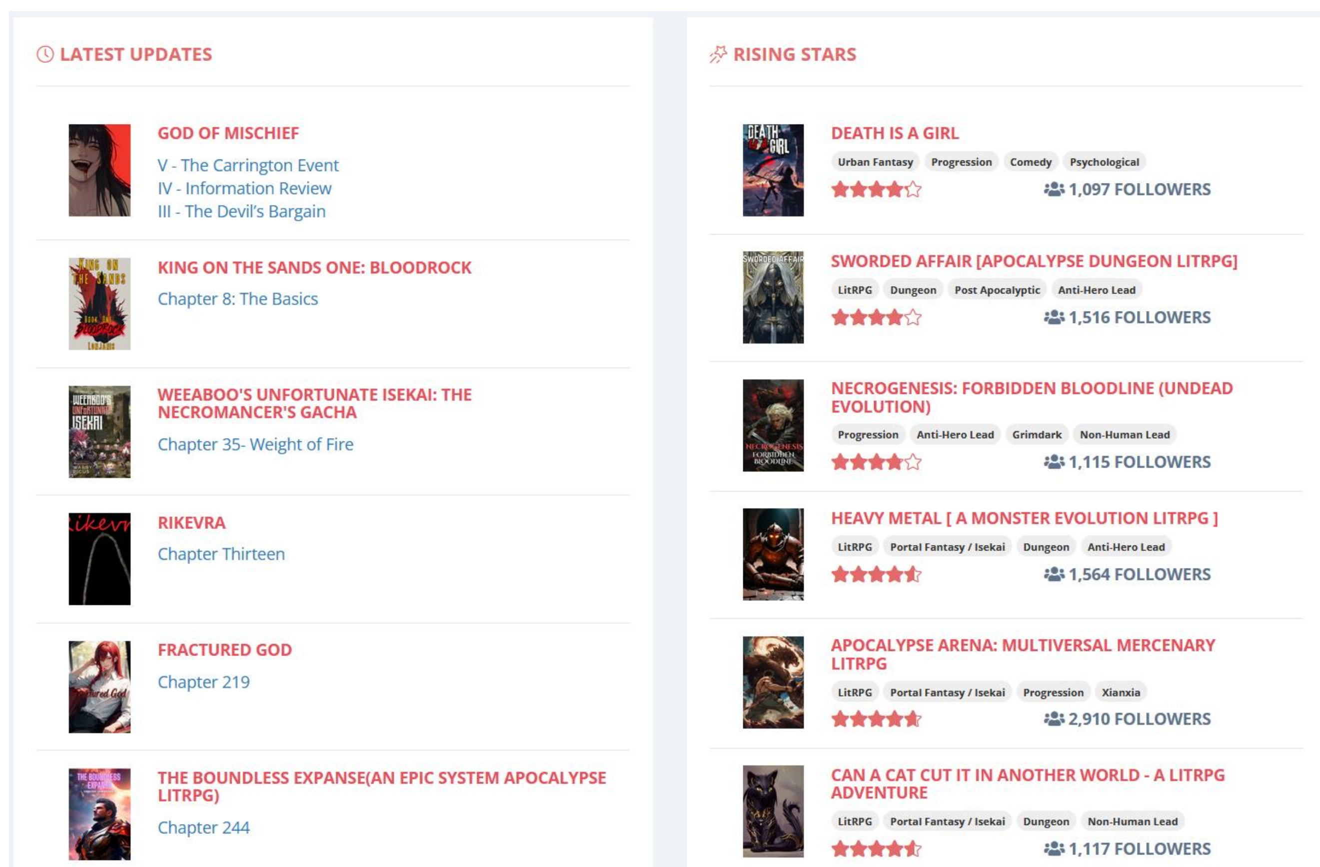


Рисунок 1.2 – Показ книг на RoyalRoad

Аналіз читацького досвіду з точки зору інтерфейсу та базової бізнес-логіки не видається актуальним, тож слід відразу перейти до огляду рекомендаційної системи. Вона теж намагається закрити потребу користувача в пошуку нового контенту. Для цього платформа насамперед формує динамічний список "Rising Stars" (див. рисунок 1.2), до якого входять "зірки, що сходять" – свіжі твори, які стрімко набирають популярність. Цей список постійно оновлюється, задаючи тренди спільноти. Таким чином нові автори отримують шанс прискорити формування читацької бази.

Тоді слабкою стороною платформи слід вважати повну відсутність персональних рекомендацій. На відміну від Wattpad, ця платформа не бере до уваги історію взаємодії конкретного користувача.

1.2 Виявлення проблем та актуалізація рішень

Платформи для самостійного видавництва художньої літератури онлайн – специфічні платформи, яким через універсальність свого призначення складно охопити весь спектр як простого, так і складного функціоналу для задоволення споживчих кінцевого користувача, а точніше – читача. Вони часто не здатні зібрати повний набір потужних (а тим паче – інноваційних) інструментів, реалізованих на платформах, що беруть на себе менше обов’язків. Мова йде про магазини електронних книг та книжні оглядові платформи.

Сучасна рекомендаційна система обов’язково має задовольняти погляд користувача на контент платформи з різних боків, будь то рекомендації контенту подібного до відкритого чи персональні рекомендації на основі історії взаємодії з платформою. До того ж, рекомендаційна система обов’язково має брати до уваги особливості предметної області. Як уже було згадано, у випадку платформи для читання та видавництва художньої літератури мова йде, наприклад, про просування нових творів, які стрімко набирають популярність. Причому всі згадані умови рекомендаційна система має задовольняти одночасно.

Водночас, розглянуті під час аналізу ринку приклади платформ стикаються з проблемою формування рекомендаційного списку для користувачів, які вперше заходять на платформу або використовують її нечасто. Ця проблема відома як "cold start" і виникає, коли користувачі мають обмежений контекст взаємодії з платформою. Важливість пошуку рішення особливо велика для колаборативної фільтрації, адже точність рекомендацій залежить від достатньої кількості даних [4].

Все це зумовлює актуальність спроби розробити MVP для платформи, яка б пропонувала не просто функціонал предметної області, а й доповнювала його персоніфікованою рекомендаційною системою, що базується на гнучкій моделі вподобань.

1.3 Постановка задач

Для реалізації повноцінної рекомендаційної системи спершу потрібно задати програмну систему, яку вона має обслуговувати. Іншими словами, необхідно публікувати та читати книжки, щоб до того ж їх рекомендувати.

Потреби публікації полягають у потребах авторів. Автор – це один із двох типів користувачів у даній програмній системі. Він очікує, що зможе створити індивідуальну сторінку для своєї книжки, а потім опублікувати її глави одна за одною по ходу написання. Саме таку логіку переслідує типове самостійне видавництво онлайн. Для наших задач її цілком вистачить. Дана робота не передбачає аналіз та розробку корисних для автора інструментів.

Опубліковані авторами книжки містять інтерес для читачів, що користуються даною програмною системою. Читачі – це другий тип користувачів. Перш за все вони бажають мати змогу переглянути повністю книжкову базу платформи, аби кожна опублікована книга була в їх безпосередньому доступі. Тоді вони зможуть відкрити потрібну книгу й почати читати їх глава за главою.

На даному етапі не вистачає лише одного компоненту програмної системи, без якої складно побудувати прогресивну рекомендаційну систему. Мова йде про читацький зворотній зв’язок – рецензії. Аби мати змогу оцінити вподобання спільноти загалом та авторизованого користувача зокрема, платформі необхідно збирати відповідні дані. Читачі зі свого боку бажають виразити свою думку про прочитане, а платформа потім має використати їх оцінки для втримання їх уваги та залученості.

Саме так з’являється можливість для подальшого проєктування та реалізації підходящої рекомендаційної системи. Вона має персоналізуватися відповідно до вподобань конкретного користувача, щоб надати індивідуальні пропозиції платформи. Проте ця система має бути достатньо гнучкою для задоволення суміжних запитів на кшталт "книги, схожі до вказаної", а також залишатися корисною для нових користувачів, даних щодо яких явно недостатньо. Іншими словами, рекомендаційна система має працювати завжди та за будь-який обставин.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Постановка мети

Основною метою роботи є розробка програмної системи для самостійного видавництва і читання та обов’язкове доповнення неї персоналізованою рекомендаційною системою.

Для серверної частини програмної системи було обрано стек технологій C#/.NET: ASP.NET, Entity Framework. Базою даною слугуватиме MySQL. В основу машинного навчання для рекомендаційної системи буде покладено бібліотеку ML.NET. Клієнтську частину програмної системи буде розроблено завдяки стеку технологій JS: React.js, Redux, Material UI.

Архітектура серверної частини програмної системи відповідатиме класичній трирівневій архітектурі [5]. Перший рівень цієї архітектури - це рівень доступу до даних (data access layer). Тут знаходяться компоненти, які забезпечують зв'язок з базою даних, виконують операції збереження та отримання даних. Другий рівень - бізнес-логіка (business logic layer). Це серце системи, де здійснюються всі операції обробки даних, валідації, бізнес-правил і логіки додатку. Нарешті, третій рівень - це рівень представлення (presentation layer). Тут знаходяться компоненти, які відповідають за обробку користувацьких запитів. Така архітектура дозволить забезпечити чітке розділення функціональності системи та полегшує розширення і підтримку програмного продукту.

Рекомендаційна система, зокрема навчена модель даних, буде частиною архітектури серверної частини програмної системи, зокрема рівня бізнес-логіки.

Архітектура клієнтської частини програмної системи полягатиме у вертикальному підході [6]. Це означає, що функціональність системи буде розділена на вертикальні структурні блоки або модулі, кожен з яких відповідає за конкретну функціональну частину програми. Такий підхід дозволяє створювати незалежні компоненти, які виконують обмежену кількість завдань і можуть бути легко масштабовані та підтримувані. Кожен модуль відповідає за свої власні операції і може бути розроблений, тестований і впроваджений незалежно від інших частин системи. Така архітектура сприяє швидкості розробки програмної системи.

2.2 Загальний опис системи

Головною частиною програмної системи виступатиме серверна частина, а саме – відокремлений API. Цей API є інтерфейсом, який надає зовнішній доступ до функціональності системи.

Клієнтська частина програми надсилатиме запити сюди, щоб скористатися реалізованою бізнес-логіку. Серед такого функціоналу API міститиме: автентифікація / авторизація, збереження книг, надсилання тексту глав, формування рекомендацій тощо.

Використання та навчання моделі даних, що лежатиме в основі рекомендаційної системи, відбуватиметься також усередині API. Рішення прийнято з міркувань економії ресурсів та збереження простоти конструкції.

У свою чергу API залежатиме від зовнішньої бази даних. Вона буде одна на систему, а її таблиці будуть повністю охоплювати визначену площу предметної області. Взаємодія з базою даних обмежуватиметься та регулюватиметься виключно через API.

Останньою частиною системи виступатиме клієнтський веб-додаток. Саме за рахунок нього користувачі системи у відповідних ролях зможуть взаємодіяти з необхідним їм функціоналом.

Ця частина міститиме графічний інтерфейс, що змінюватиметься відповідно до взаємодій, а також результатів обробки запитів на стороні API. Клієнтський веб-додаток не міститиме бізнес-логіки, не матиме доступу до бази даних без посередника, не зберігатиме сенситивних даних і не реалізовуватиме фактичних операцій над ними.

2.3 Основний функціонал системи

Увесь функціонал програмної системи розподіляється згідно двох типів (ролей) користувачів: читач та автор. Уодночас, рекомендаційна система пристосована до обслуговування саме споживача – тобто читача.

Загальний функціонал:

– реєстрація облікового запису користувача (за замовчуванням він отримуватиме обидві ролі);

– логін за допомогою пошти та паролю (надає доступ до функціоналу, поділеного за ролями та прихованого від анонімних користувачів).

Функціонал, що не потребує автентифікації та авторизації:

– перегляд повної бази книжок на платформі, поділеного пагінацією;

– перегляд індивідуальної сторінки книги;

– читання глав;

– перегляд рецензій до книги;

– рекомендації книг, схожих до відкритої.

Функціонал, що потребує авторизації читача:

– написання власної рецензії до книги;

– перегляд персоналізованих рекомендацій.

Функціонал, що потребує авторизації автора:

– перегляд опублікованих книг;

– додавання опублікованих глав;

– оновлення опублікованих глав;

– видалення опублікованих книг;

– перегляд опублікованих глав;

– додавання опублікованих глав;

– оновлення опублікованих глав;

– видалення опублікованих глав.

2.4 Загальні обмеження

Програмна система матиме наступні обмеження:

– використання Material UI припускає, що підтримуваними браузерами виступають Edge (>= 91), Firefox (>= 78), Chrome (>= 90), Safari (>= 14), Internet Explorer (>= 11) згідно релізів, указаних у дужках [6].

2.5 Припущення та залежності

Список припущень:

– нові автори, охочі до використання подібної онлайн платформи, готові до публікації книг у форматі веб-новел (тобто глава за главою);

– нові читачі готові залишати рецензії до творів, які вони почали / завершили читати на новій платформі.

Список залежностей:

– існує початкова база книжок, користувачів і рецензій, достатня для навчання ефективної моделі даних;

– модель даних зможе обслуговувати запити нових користувачів, історія взаємодії яких недостатня, та враховувати нові публікації, що з’явилися після попереднього навчання;

– читачі матимуть можливість отримати рекомендації на самому початку взаємодії з платформою, якість яких покращуватиметься з часом.

3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Проєктування бази даних

Розглянемо запропоновану схему базу даних, представлену за допомогою ER-діаграми. Вона містить усі необхідні сутності предметної області, взаємозв'язки між ними та потенційні поля. Всього для даної програмної системи та її рекомендаційної системи вистачить наступних сутностей: користувач, роль, книга, глава, рецензія. Доповнюючи основний функціонал, ми можемо розширити схему додатковими сутностями, проте для демонстрації працездатної рекомендаційної системи достатньо перерахованого.

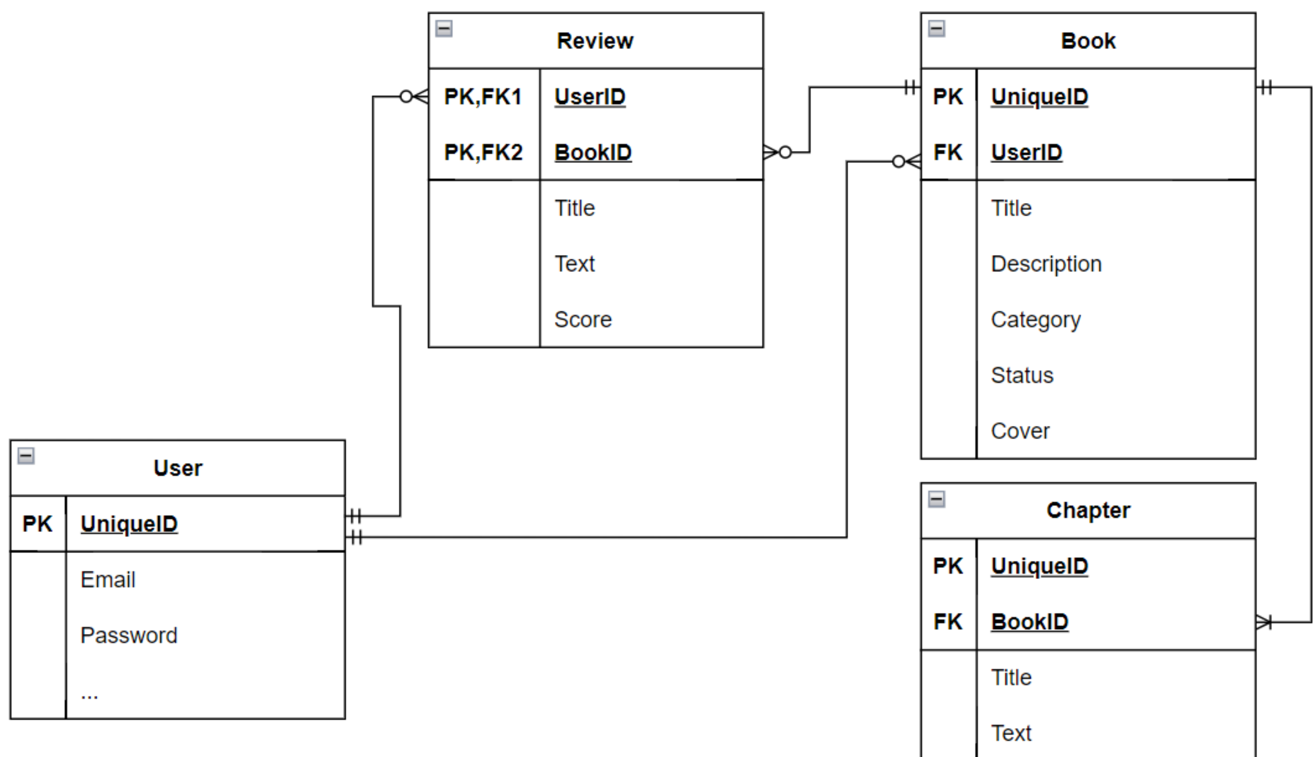


Рисунок 3.1 – ER-діаграма бази даних

Сутність Користувач має зв'язок багато-до-багатьох із сутністю Роль. Він потрібен для того, щоб користувачам можна було призначати декілька ролей без їх дуплікації. Також сутність Користувач має зв'язок один-до-багатьох із сутністю Книга. Це реалізація взаємозв'язку автора з написаною ним книгою, а точніше – власника з його книгою. У реаліях розглянутої програмної системи це має більше сенсу.

Сутність Книга має зв’язок один-до-багатьох із сутністю Глава. Таким чином буде представлено прив’язку тексту до відповідної книги, без ускладнення залежностей сутністю Користувача, який матиме зв’язок один-до-багатьох із сутністю Книга.

Сутність Рецензія – це реалізація зв’язку багато-до-багатьох між сутностями Користувач та Книга. У звичайному випадку вона б виступала проміжною сутністю, яка б лише поєднувала сутності, проте програмній системі необхідно зберігати асоційовані дані предметної області (рейтинг), тому ми розглядаємо її явно.

3.2 Проєктування серверної частини програмної системи

Серверною частино програмної системою слугуватиме API. Він прийматиме запити на обробку бізнес-логіки через точки доступу, а також контролюватиме авторизацію взаємодій і решту пов’язаних задач на кшталт логування.

Конфігурація API загалом та точок доступу зокрема будуть інкапсульовані на рівні презентації разом із реалізацією власне точок доступу. Цей рівень є рівнем інтерфейсу в трирівневій архітектурі, що лежатиме в основі проєктування серверної частини програної системи.

У додатку А на рисунках А.1 та А.2 можна знайти діаграму прецедентів, що допоможе розібрати акторів програмної системи та варіанти їх взаємодій. Усього буде розглянуто дві ролі: Читач та Автор. Уочевидь, доступ до бізнес-логіки регулюватиметься відповідно до призначених ролей. Проте слід відмітити, що типовий зареєстрований користувач матиме ролі Читача й Автора, що забезпечуватиме доступ до більшості функціоналу. Також частину функціоналу відкрито до анонімного використання.

Можливості, пов’язані з автентифікацією та авторизацією:

- реєстрація облікового запису за допомогою пошти та паролю;
- отримання токену доступу за допомогою пошти та паролю;
- оновлення токену доступу за допомогою токену оновлення.

Можливості анонімного користувача:

– операції зчитування книжок;

– операції зчитування глав;

– операції зчитування рецензій;

– отримання рекомендацій книг, схожих до наданої.

Можливості авторизованого Читача:

– CRUD-операції над власними рецензіями;

– отримання персоналізованих рекомендацій.

Можливості авторизованого Автора:

– CRUD-операції над власними книжками;

– CRUD-операції над власними главами.

Власне бізнес-логіка не буде реалізована на рівні презентації. Точки доступу застосовуватимуть сервіси, що реалізовані на рівні бізнес-логіки (див. рисунок Г.1 додатку Г). Ці сервіси не залежатимуть від рівня презентації та міститимуть виключно функціонал предметної області та моделі даних для обміну між рівнями. У свою чергу цей рівень залежить від рівня доступу до даних, аби проводити за його рахунок взаємодії з базою даних.

За наведеними можливостями щодо автентифікації та авторизації можна констатувати, що програмна система застосовуватиме bearer-токен підхід. Тобто доступ регулюватиметься за допомогою токенів доступу, що міститимуть необхідні для системи метадані на кшталт електронної пошти та ролі. Також тривалість дієздатності токенів доступу підтримуватиметься токенами оновлення, що дозволяють отримати новий токен доступу без повторної автентифікації.

3.3 Проєктування рекомендаційної системи

Розробка та впровадження необхідної проєкту рекомендаційної системи має на меті видобути вигоду з історичних даних взаємодії користувачів із програмною системою, а точніше – з накопичених у базі даних рецензій. Враховуючи приховані залежності, алгоритм зможе навчитися формувати персоналізовані рекомендації за запитом кожного користувача.

Для ефективного виконання подібного завдання сучасні рекомендаційні системи застосовують методи колаборативної фільтрації [7]. Колаборативна фільтрація полягає у виявленні подібностей у минулій поведінці користувача та прогнозуванні для нього на основі схожої поведінки з іншими користувачами. Потім ця модель використовується для прогнозування продуктів (або оцінок продуктів), які можуть зацікавити користувача. Наприклад, Netflix пропонує вам наступний серіал для перегляду на основі серіалів, які ви вже дивилися раніше, але також на основі серіалів, які дивилися користувачі, і яким сподобався той самий контент, що й вам. А в нашому випадку продуктами виступатимуть опубліковані книжки.

Існує декілька методів колаборативної фільтрації. У цій роботі застосовуватиметься алгоритм факторизації матриць. Матрична факторизація - це спосіб генерування прихованих ознак при перемножуванні двох різних типів об'єктів.

Тоді колаборативна фільтрація - це застосування матричної факторизації для виявлення зв'язку між сутностями товарів і користувачів [8]. Маючи дані про оцінки користувачів щодо книг у програмній системі, ми хотіли б передбачити, як користувачі оцінили б решту книг, щоб користувачі могли отримати рекомендації на основі прогнозу .

Для того, щоб розглянути математичний концепт методу матричної факторизації, слід задати наступні дані: множину користувачів (U), книг (B). Матриця $R = |U| \times |B|$ містить усі оцінки, виставлені користувачами. Мета полягає у виявленні k прихованих ознак (див. формулу 3.1). Якщо на вхід подати дві матриці $P(|U| \times k)$ та $P(|B| \times k)$, то вона згенерує результат їх множення $\hat{R}$.

$$R \approx P \times Q = \hat{R}, \qquad 3.1)$$

де R – початкова матриця оцінок книг користувачами;

$\hat{R}$ – матриця, доповнена передбаченими оцінками.

Матриця P відображає зв'язок між користувачем та ознаками, а матриця Q відображає зв'язок між книгою та ознаками. Ми можемо отримати прогноз рейтингу товару шляхом обчислення точкового добутку двох векторів u_i та b_j (див. формулу 3.2).

$$\hat{r}_{ij} = p_i q_j = \sum_{k=1}^{k} p_{ik} q_{jk}, \quad 3.2)$$

де $\hat{r}_{ij}$ – передбачення рейтингу.

Щоб отримати дві сутності P і Q, нам потрібно ініціалізувати обидві матриці і обчислити різницю добутку. Далі ми мінімізуємо цю різницю за допомогою ітерацій. Метод називається градієнтним спуском і спрямований на пошук локального мінімуму різниці (див. формулу 3.3).

$$e_{ij}^2 = \left(r_{ij} - \hat{r}_{ij}\right)^2 = \left(r_{ij} - \sum_{k=1}^{k} p_{ik} q_{jk}\right)^2, \quad 3.3)$$

де e_{ij}^2 – квадрат похибки передбаченої матриці відносно початкової.

Мінімізувати похибку здатен градієнт, і тому ми диференціюємо наведене вище рівняння відносно цих двох змінних окремо (див. формулу 3.4).

$$\begin{gathered} \frac{d}{dp_{ik}} e_{ij}^2 = \left(r_{ij} - \hat{r}_{ij}\right)\left(q_{jk}\right) = -2 e_{ij} q_{jk} \\ \frac{d}{dq_{ik}} e_{ij}^2 = \left(r_{ij} - \hat{r}_{ij}\right)(p_{ik}) = -2 e_{ij} p_{ik} \end{gathered} \quad 3.4)$$

Виходячи з градієнта, математична формула може бути оновлена як для p_{ik}, так і для q_{jk}. Тоді α - це крок для досягнення мінімуму під час обчислення градієнта, і зазвичай α задається невеликим числом. У формулі 3.5 можна оцінити результат.

$$p'_{ik} = p_{ik} + \alpha \frac{d}{dp_{ik}} e_{ij}^2 = p_{ik} + 2\alpha e_{ij} q_{jk}$$
$$q'_{jk} = q_{jk} + \alpha \frac{d}{dq_{jk}} e_{ij}^2 = q_{jk} + 2\alpha e_{ij} p_{ik} \quad 3.5)$$

З наведеного вище рівняння, p'_{ik} та q'_{jk} можна оновлювати за допомогою ітерацій, поки похибка не збіжиться до мінімуму (див. формулу 3.6).

$$E = \sum_{(u_i, b_j, r_{ij})} e_{ij} = \sum_{(u_i, b_j, r_{ij})} \left(r_{ij} - \sum_{k=1}^{k} p_{ik} q_{jk} \right), \quad 3.6)$$

де E – мінімальна похибка передбаченої матриці відносно початкової;

(u_i, b_j, r_{ij}) – взаємодія користувача u_i з книгою b_j за рейтингом r_{ij}.

Приховані ознаки, тобто асоціації між матрицями користувачів та книг, визначаються рекомендаційною системою, щоб знайти схожість і зробити прогноз на основі записів як книг, так і користувачів. Ось чому нам знадобляться накопичені в базі даних рецензії, адже вони містять зв’язки між книгами та користувачами, доповнені якісною властивістю – рейтингом, проставленим користувачем.

Уодночас, попри доведену ефективність, модель даних, навчена за допомогою факторизації матриць, стикається з проблемами під час прогнозування для користувачів, щодо яких мало даних, та нових користувачів, щодо яких дані повністю відсутні.

Ця проблема носить назву "холодного старту" [4]. Система рекомендацій може запропонувати необхідні товари лише після того, як користувач вибере їх за власною ініціативою. Такі користувачі вже не є "холодними", і інформація про них може бути використана для підтримки споживчого вибору. Холодний старт є важливою проблемою в рекомендаційній системі для епізодичних користувачів, які відвідують відповідний сайт періодично, з великими інтервалами, наприклад, для купівлі побутової техніки, книг, подорожей на вихідні.

У такому випадку необхідно вжити заходів для задання прихованих залежностей, які має виявити рекомендаційна система, певним іншим способом. У даній роботі для цього не застосовуватимуться додаткові алгоритми, які могли б доповнити навчання моделі даних або внести правки в результати її роботи. Натомість буде розглянута спеціальна підготовка вхідних даних для донавчання моделі та запиту до вже готової моделі.

Модель даних, навчена за допомогою факторизації матриць, цілком здатна підготувати персоналізовані рекомендації для нового користувача. Як уже було сказано, для цього їй не вистачає лише персональної інформації – рецензій нового користувача. Тому для вирішення проблеми пропонується підготувати заміну. Для початку необхідно відібрати випадкові книжки з топу найкращих книг за середнім рейтингом. Потім слід знайти користувачів із найбільшою кількістю рецензій, які водночас високо оцінили книги, відібрані на попередньому кроці. Ці рецензії зі зміненим походженням до нового користувача можливо застосувати для донавчання моделі даних. Інакше слід обрати випадкового користувача серед тих, що були відібрані на попередньому кроці та використати погляд з його боку для надання рекомендацій. Повторювати подібний сценарій прийдеться кожен раз, поки новий користувач не набере певну кількість власних рецензій.

Також колаборативну фільтрацію можна використати використати для задачі, яка більше підходить простішим підходам до рекомендацій: підбір книг, схожих до відкритої. Аби досягти подібного результату, модель даних можливо донавчити на основі набору вхідних даних, пов’язаного з відкритою книгою. Для початку достатньо подати запис взаємозв’язку пустого (неіснуючого) користувача з відкритою книгою та подати моделі на донавчання. Після вона зможе надати рекомендації для даного користувача безпосередньо з врахуванням відкритої книги.

Усі процеси, пов’язані з рекомендаційною системою, зокрема навчання моделі та обробка запитів до неї, відбуватимуться в API (див. рисунок Д.1 додатку Д). Тоді серверна частина програми застосовуватиме модель даних як внутрішній модуль.

3.4 Проєктування клієнтської частини програмної системи

Клієнтською частиною слугуватиме веб-додаток, що розгорнутий окремо від серверної частини. Цей додаток буде працювати через веб-браузер без необхідності встановлення спеціального програмного забезпечення на бік користувача. Веб-додаток буде взаємодіяти з серверною частиною за допомогою мережевих запитів, обмінюючись даними із сервером для обробки різних запитів та оновлення інформації на стороні клієнта. Таким чином буде завершено реалізацію клієнт-серверної архітектури програмної системи (див. рисунок Е.1 додатку Е). Така архітектура дозволить забезпечити масштабованість, безпеку і доступність системи для користувачів з будь-яких пристроїв з доступом до Інтернету.

Також це означає, що з клієнтської частини повністю знімається відповідальність за бізнес-логіку програмної системи (див. рисунки В.1 та В.2 додатку В) та фактичне управління даними. Натомість веб-додаток слугуватиме як зручний та зрозумілий інтерфейс користувача. Внутрішня логіка додатку регулюватиме взаємодію з API на основі даних, які API повертатиме. Наприклад, додаток зберігатиме токени доступу та токени оновлення. Тобто збереження даних, у тому числі в якості локального кешування, та їх модифікація відповідатимуть лише вимогам побудови клієнтського додатку.

Тоді вертикальна архітектура забезпечуватиме модульність при розробці додатку. Вертикальна архітектура в контексті окремого веб-додатку означає організацію коду та функціональності додатку вздовж функціональних функцій або функціональних блоків. Замість того, щоб групувати код за рівнями або компонентами, функціонально пов'язані частини програми знаходяться поруч, утворюючи вертикальні структури (див. рисунок Б.2 додатку Б).

У цій архітектурі кожен модуль або компонент відповідає за виконання певного видимого для користувача функціоналу. Наприклад, може бути окремий компонент для реєстрації користувача, компонент для управління профілем користувача, компонент для читання книг тощо.

Вертикальна архітектура спрощує розуміння структури додатку, оскільки функціонально пов'язані частини розташовані поруч одна з одною. Кожен

компонент може бути незалежно розвинутим та тестованим, що полегшує роботу з окремими частинами додатку.

Такий підхід дозволить розробникам фокусуватися на конкретній функціональності додатку без зайвого перекриття між компонентами. Вертикальна архітектура також сприяє простоті розвитку нових функцій або модулів, оскільки вони можуть бути додані як нові вертикальні блоки, не впливаючи на існуючі компоненти.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Прийняті рішення серверної частини

4.1.1 Прийняті рішення доступу до даних

Вибір MySQL як системи управління реляційними базами даних (СКБД) для даної програмної системи ґрунтується на її перевіреній надійності, продуктивності та широкому впровадженні в індустрії. MySQL, розроблена та підтримувана корпорацією Oracle, є однією з найпопулярніших баз даних з відкритим вихідним кодом у світі, що робить її добре підтримуваним та надійним вибором для управління базою даних проєкту.

Використання умовно безкоштовної вбудованої бази даних MySQL для Azure Web App - це зручне й економічно вигідне рішення для розробників, яким потрібен швидкий і простий спосіб інтегрувати базу даних MySQL у свої веб-програми. Ця функція особливо корисна для розробки, тестування та невеликих додатків, де простота та економія коштів мають вирішальне значення.

Тобто це легка, повністю керована служба бази даних MySQL, вбудована в Azure Web App. Ця інтеграція дає змогу розробникам швидко створювати базу даних MySQL без необхідності окремої підготовки, конфігурації або керування зовнішніми службами баз даних. Так було забезпечено можливість перейти до активної розробки із збереженням повноцінної функціональності без зайвих грошових затрат.

Для доступу до даних використовується Entity Framework Core (EF Core), що добре узгоджується з нашим вибором MySQL. EF Core - це легка, розширювана та крос-платформна версія Entity Framework, популярного об'єктно-реляційного маппера (ORM) від Microsoft.

Він полегшує розробку коду доступу до даних, дозволяючи розробникам працювати з більш високим рівнем абстракції за рахунок зосередження на сутностях предметної області, а не таблицях бази даних.

Центральним елементом реалізації EF Core та його повноцінного застосування в програмній системі виступає клас так званого контексту:

```
public class WheelingfulDbContext : IdentityDbContext<AppUser>
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Chapter> Chapters { get; set; }
    public DbSet<Review> Reviews { get; set; }
    public
WheelingfulDbContext(DbContextOptions<WheelingfulDbContext> options)
        : base(options) { }
    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTrack
ing);
    }
    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        modelBuilder.BuildBook();
        modelBuilder.BuildChapter();
        modelBuilder.BuildReview();
        base.OnModelCreating(modelBuilder);
        modelBuilder.SeedRoles();
        modelBuilder.SeedUsers();
        modelBuilder.SeedAssignRoles();
    }
}
```

Використання DbContext, як показано у WheelingfulDbContext, має кілька ключових переваг. По-перше, він централізує конфігурацію та керування нашими сутностями бази даних. Визначивши DbSet<Book>, DbSet<Chapter> і DbSet<Review>, ми чітко окреслюємо таблиці, якими будемо керувати у нашій базі даних. Такий підхід називається Code-First. Визначивши сутності бази даних за допомогою класів, написаних мовою C#, EF Core дозволяє створювати міграції бази даних для збереження структури бази даних у коді та історії її оновлень.

Завдяки налаштуванню DbContextOptionsBuilder за допомогою optionsBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking) оптимізується доступ до бази даних для зчитування. Цей параметр гарантує, що контекст не відстежує зміни в отриманих сутностях, зменшуючи навантаження на пам'ять і покращуючи продуктивність для запитів, де відстеження не потрібне.

У методі OnModelCreating використовується ModelBuilder для налаштування зв'язків між сутностями та початкових даних. Методи BuildBook(), BuildChapter() та BuildReview() є методами, які налаштовують властивості сутностей, зв'язки та обмеження.

Заповнення початкових даних - це ще один важливий аспект, яким керує OnModelCreating. Методи SeedRoles(), SeedUsers() та SeedAssignRoles() ініціалізують базу даних з попередньо визначеними ролями та ключовими користувачами на кшталт адміністратора.

4.1.2 Прийняті рішення API

Фреймворк .NET, розроблений компанією Microsoft, - це надійна і зріла екосистема, яка надає повний набір інструментів і бібліотек. Це робить його чудовим вибором для створення масштабованих, підтримуваних і високопродуктивних додатків.

Однією з головних переваг стеку .NET є його потужна підтримка додатків корпоративного рівня. ASP.NET, будучи невід'ємною частиною цього стеку, пропонує потужний фреймворк для створення динамічних веб-додатків.

Він надає багатий набір функцій, які дозволяють розробникам ефективно створювати безпечні, масштабовані та високопродуктивні веб-додатки. Модульна архітектура фреймворку дозволяє легко інтегрувати різні компоненти, гарантуючи, що система може рости і розвиватися з часом без значних переробок або перебоїв у роботі.

Дана технологія відповідає насамперед за налаштування API. Розробникам надається можливість детально задати точки доступу, а потім довершити конфігурацію за допомогою так званих пайплайнів. Це дозволяє гнучко налаштовувати процес обробки запитів та відповідей, забезпечуючи ефективну інтеграцію з іншими сервісами та системами. Завдяки цьому, розробники можуть створювати надійні та масштабовані рішення, які легко адаптуються до змін вимог та умов експлуатації.

Тобто згідно потреб проєкту процес попередньої обробки запиту можна наситити різноманітними додатковими операціями на кшталт перевірки спільного використання ресурсів із різних джерел (CORS), автентифікації та авторизації користувача, валідації переданих даних та багато іншого.

Згідно конфігурації API в точці збірки запропонованої програмної системи можна оцінити, як було використано можливості ASP.NET та пов'язаних технологій зі стеку .NET:

```
builder.Services
    .AddAuthentication()
    .AddBearerToken();
builder.Services.AddAuthorizationBuilder()
  .AddPolicy(PolicyContants.AuthorizeAuthor, policy =>
        policy
             .RequireRole(nameof(UserRoleEnum.Admin),
nameof(UserRoleEnum.Author)))
  .AddPolicy(PolicyContants.AuthorizeReader, policy =>
        policy
             .RequireRole(nameof(UserRoleEnum.Admin),
nameof(UserRoleEnum.Reader)));
var app = builder.Build();
app.UseCors(PolicyContants.AllowClientOrigin);
app.UseAppExtension();
app.UseHttpsRedirection();
app.UseAuthentication();
app.UseAuthorization();
app.UseSerilogRequestLogging();
```

Даний фрагмент коду демонструє налаштування пайплайну попередньої обробки запиту. Можна побачити, що ASP.NET задає CORS, автентифікацію, авторизацію, логування.

Політики авторизації визначаються за допомогою builder.Services.AddAuthorizationBuilder().AddPolicy(...), визначаючи контроль доступу на основі ролей. Такі політики, як PolicyContants.AuthorizeAuthor та PolicyContants.AuthorizeReader гарантують, що лише користувачі з відповідними ролями можуть виконувати певні дії, підвищуючи рівень безпеки та відповідності.

Тоді app.UseCors(PolicyContants.AllowClientOrigin) налаштовує політики CORS, забезпечуючи безпеку перехресних запитів. app.UseAppExtension() і app.UseHttpsRedirection() налаштовують кастомне проміжне програмне

забезпечення і впроваджують HTTPS, підвищуючи безпеку і функціональність.

Автентифікація та авторизація застосовується за допомогою app.UseAuthentication() та app.UseAuthorization(), гарантуючи, що всі вхідні запити будуть належним чином автентифіковані та авторизовані. Це налаштування є критично важливим для захисту конфіденційних ресурсів і забезпечення дотримання політик безпеки.

Водночас, згідно принципів трирівненої архітектури API, реалізований за допомогою ASP.NET, виступає рівнем презентації, а тому не може містити бізнес-логіку. Кожна задана точка доступу може лише застосовувати власні конфігурації, а також сервіси, реалізовані на інших рівнях. Відповідно, типова точка доступу має наступний вигляд:

Ця структура чітко розділяє завдання обробки запитів, виконання бізнес-логіки та генерації відповідей. Кінцева точка призначена для обробки HTTP POST запитів на створення рецензій, пов'язаних з конкретною книгою, яка ідентифікується за допомогою {bookId}.

Параметр посередника [FromServices] IMediator підключає службу IMediator, дотримуючись шаблону посередника. Цей шаблон є ключовим компонентом трирівневої архітектури, що полегшує розділення завдань шляхом делегування обробки бізнес-логіки окремим класам обробників. Тоді використання await mediator.Send(request.To()) асинхронно надсилає запит до медіатора, який потім перенаправляє його до відповідного обробника з рівня бізнес-логіки. Це гарантує, що бізнес-логіка створення відгуку інкапсульована в обробнику, сприяючи чіткому розділенню між рівнями презентації та бізнес-логіки.

4.1.3 Прийняті рішення машинного навчання

Вибір ML.NET для матричної факторизації в програмній системі ґрунтується на його здатності забезпечити безперебійну та ефективну структуру машинного навчання, яка ідеально інтегрується з додатками .NET.

ML.NET спеціально розроблений для роботи в екосистемі .NET, пропонуючи вбудовану інтеграцію, яка дозволяє розробникам створювати, навчати і розгортати

моделі машинного навчання, не виходячи за межі звичного середовища. Ця інтеграція спрощує процес розробки, зменшує складність і підвищує продуктивність, оскільки розробники можуть використовувати наявний досвід роботи з .NET.

Завдяки можливостям цієї бібліотеки, код навчання рекомендаційної моделі, необхідної програмній системі, вміщається в один метод:

```
        var splitData = mlContext.Data.TrainTestSplit(
            data: mlContext.Data.LoadFromEnumerable(reviews),
            testFraction: 0.2
        );
        IEstimator<ITransformer> estimator =
mlContext.Transforms.Conversion
            .MapValueToKey(outputColumnName: "UserIdEncoded",
                inputColumnName: "UserId")
            .Append(mlContext.Transforms.Conversion
            .MapValueToKey(outputColumnName: "BookIdEncoded",
                inputColumnName: "BookId"));
        var options = new MatrixFactorizationTrainer.Options
        {
            MatrixColumnIndexColumnName = "UserIdEncoded",
            MatrixRowIndexColumnName = "BookIdEncoded",
            LabelColumnName = "ReviewScore",
            NumberOfIterations = 20,
            ApproximationRank = 100
        };
        var trainerEstimator = estimator
 .Append(mlContext.Recommendation().Trainers.MatrixFactorization(options));
        ITransformer model = trainerEstimator.Fit(splitData.TrainSet);
        var prediction = model.Transform(splitData.TestSet);
        var metrics = mlContext.Regression
            .Evaluate(prediction, labelColumnName: "ReviewScore",
           scoreColumnName: "Score");
```

Метод ініціалізує екземпляр MLContext, який слугує точкою входу для всіх операцій ML.NET. Цей контекст необхідний для управління життєвим циклом машинного навчання, включаючи завантаження даних, перетворення, навчання моделі та оцінювання.

Розділення даних виконується за допомогою mlContext.Data.TrainTestSplit, розділяючи дані на навчальні та тестові набори з тестовою часткою 20%.

Далі код визначає конвеєр IEstimator<ITransformer>, який починається з перетворень для зіставлення ідентифікаторів користувачів та книг з ключовими значеннями. Таке кодування необхідне, оскільки матрична факторизація вимагає

числового представлення категорій даних. Перетворення MapValueToKey перетворює відповідні значення на числові індекси, які потім використовуються алгоритмом матричної факторизації.

Тренер матричної факторизації налаштовується за допомогою певних параметрів, таких як LabelColumnName, NumberOfIterations та ApproximationRank. Ці параметри визначають вхідні стовпці, стовпець міток, кількість ітерацій навчання та ранг апроксимації матричної факторизації. Налаштувавши ці параметри, код гарантує, що модель буде пристосована до конкретних вимог програми та характеристик даних.

Підсумовуючи сказане, цей метод навчає рекомендаційну модель та оцінює його якість за допомогою оцінки відхилення від середнього квадратичного та кореню з нього.

Потім цей метод застосовується всередині іншого методу, який проводить формування персональних рекомендацій:

```
.Select(b =>
{
    var modelInput = new ModelInput
    {
        BookId = b.Id,
        UserId = currentUser.Id
    };
    var modelOutput = prediction.Predict(modelInput);
    return new { Book = b, PredictedScore = modelOutput.Score };
})
.Where(o => !currentUserReviews.Any(r => r.BookId == o.Book.Id)
          && o.PredictedScore >= MlConstants.ScoreMinThreshold
          && o.PredictedScore <= MlConstants.ScoreMaxThreshold)
.OrderByDescending(o => o.PredictedScore)
.Select(o => new FetchBookResponse
{
    Id = o.Book.Id,
    Title = o.Book.Title,
    Description = o.Book.Description,
    Category = o.Book.Category,
    Status = o.Book.Status,
    CoverUrl = bookCover.GetCoverUrl(o.Book.Id,
          o.Book.AuthorUserId, o.Book.CoverId),
    AuthorUserName = o.Book.AuthorUserName,
    Reviews = o.Book.Reviews,
})
.Take(PaginationConstants.DefaultPageSize);
```

Метод починається з оперування колекцією книг з метою створення персоналізованих рекомендацій для поточного користувача. Основна ідея полягає в тому, щоб передбачити, наскільки ймовірно, що користувач високо оцінить кожну книгу, а потім відфільтрувати і відсортувати ці книги на основі прогнозованих оцінок.

Для кожної книги в колекції створюється об'єкт ModelInput. Цей об'єкт включає BookId та UserId поточного користувача. Клас ModelInput слугує вхідною схемою для моделі прогнозування, гарантуючи, що для прогнозування надаються необхідні характеристики.

Потім метод використовує механізм передбачення, представлений prediction, щоб передбачити оцінку для кожної книги. Метод Predict механізму передбачення отримує об'єкт ModelInput і повертає об'єкт ModelOutput, який містить передбачену оцінку. На цьому кроці використовується навчена модель машинного навчання, щоб оцінити, наскільки сильно поточний користувач хотів би прочитати книжки.

Після отримання прогнозованих оцінок метод створює колекцію об'єктів, що містять книгу та її прогнозовану оцінку. Ця проміжна колекція необхідна для фільтрації та сортування книг на основі прогнозованих оцінок. На вихід із методу піде готова колекція з персональними рекомендаціями, які відповідна точка доступу поверне клієнтському додатку на його запит.

4.2 Прийняті рішення клієнтської частини

Опис окремого модуля у вертикальній архітектурі додатку передбачає розуміння того, як усі необхідні компоненти, пов'язані з певною функцією, інкапсульовані в одному цілісному блоці. У вертикальному додатку кожен модуль або фрагмент представляє окрему частину функціоналу, окресленої для програмної системи, а все необхідне для цього функціоналу організовано у відповідному каталозі та спеціалізованих каталогах усередині нього. В даній програмній системі типовий модуль представлений маршрутизатором, сховищем та компонентами сторінок.

Почнемо з прикладу маршрутизатора:

```
export default function AuthRouter() {
  return (
    <Routes>
      <Route path={AUTH_CONFIG.routes.login}
           element={<LoginPage />} />
      <Route path={AUTH_CONFIG.routes.register}
           element={<RegisterPage />} />
    </Routes>
  );
}
```

Компонент AuthRouter є базовою частиною модуля Auth у вертикальній архітектурі програми. Його основна роль полягає в управлінні логікою маршрутизації, специфічною для модуля Auth, гарантуючи, що шляхи, пов'язані з автентифікацією, є чітко визначеними і легкодоступними.

Функція AuthRouter використовує компонент Routes для створення структури маршрутизації для модуля Auth. Визначаючи конкретні шляхи і відповідні їм компоненти, вона організовує навігацію користувачів у модулі.

Наступним слід розглянути сторінковий компонент LoginPage, який інкапсулює в собі фактичну сторінку, її розмітку та поведінку:

```
export default function LoginPage() {
  const [signIn, results] = useSignInMutation();
  let error;
  if (results.isError) {
    error = <div>{results.error.data.detail}</div>;
  }
  return (
    <AuthComponent
      onSubmit={({ email, password }) => signIn({ email, password })}
      title="Please, enter your credentials."
      authTitle="Sign In"
      redirectTitle="Sign Up"
      redirectTo={
      `/${AUTH_CONFIG.routes.group}/${AUTH_CONFIG.routes.register}`}
      error={error}
      isLoading={results.isLoading}
      isSuccess={results.isSuccess}
      isError={results.isError}
    />
  );
}
```

В основі LoginPage лежить хук useSignInMutation, який полегшує процес входу в систему, керуючи станом і логікою, необхідною для входу. Цей хук повертає функцію signIn, яка використовується для ініціювання процесу входу в систему, разом з результатами - об'єктом, який відстежує стан і результат спроби входу в систему. Така інтеграція бізнес-логіки безпосередньо в компонент ілюструє модульний характер вертикальної архітектури, де кожен компонент є самодостатнім і відповідає за свою специфічну функціональність.

Обробка процесу входу включає в себе управління різними станами, такими як завантаження, успіх і помилка. Компонент перевіряє наявність помилки при спробі входу через results.isError. Якщо помилку виявлено, він присвоює повідомлення про помилку змінній error, яка потім відображається у компоненті. Таке управління станом гарантує, що користувачі отримують негайний зворотній зв'язок про свої дії, покращуючи загальний користувацький досвід.

Компонент LoginPage рендерить AuthComponent, перевикористовуваний компонент, пристосований для завдань, пов'язаних з автентифікацією. Цей компонент отримує різні дані, щоб налаштувати його поведінку для процесу входу в систему. Серед таких даних передається функція onSubmit, яка запускає функцію signIn з електронною поштою та паролем користувача. Обробляючи логіку надсилання всередині компонента, LoginPage підтримує чіткий і прямий потік даних і дій.

Останньою важливою частиною, що повторюється в кожному модулі, виступає складова Redux-сховища, що охоплює складову загального стану даних запущеного в браузері додатку, а також певну логіку доступу до них, зокрема відправку запитів на зовнішнє API:

```
export const authApi = createApi({
  reducerPath: 'authApi',
  baseQuery: baseQuery,
  endpoints(builder) {
    return {
      signUp: builder.mutation({
        query: (credentials) => {
          return {
            url: '/auth/register',
            body: {
```

```
                email: credentials.email,
                password: credentials.password,
              },
              method: 'POST',
            };
          },
          onQueryStarted: onAuthQueryStarted,
        }),
        signIn: builder.mutation({
          query: (credentials) => {
            return {
              url: '/auth/login',
              body: {
                email: credentials.email,
                password: credentials.password,
              },
              method: 'POST',
            };
          },
          onQueryStarted: onAuthQueryStarted,
        }),
      };
    },
  });
```

В основі цього фрагмента лежить асинхронна функція onAuthQueryStarted, яка призначена для обробки побічних ефектів запитів на автентифікацію. Ця функція приймає два аргументи: аргумент arg, який містить аргументи, передані в запит, і об'єкт з властивостями dispatch і queryFulfilled. Функція dispatch використовується для надсилання дій до сховища Redux, а queryFulfilled - це обіцянка, яка виконується після завершення запиту.

AuthApi створюється за допомогою createApi з Redux Toolkit. Він визначає дві кінцеві точки: signUp і signIn, кожна з яких налаштована на обробку реєстрації та входу користувачів відповідно. Ці кінцеві точки є мутаціями, тобто вони змінюють дані, надсилаючи запити сторону сервера.

Конфігурація кожної кінцевої точки включає функцію запиту, яка визначає деталі запиту. Для signUp функція створює POST-запит на URL-адресу /auth/register з електронною поштою та паролем користувача в тілі запиту. Аналогічно, для signIn запит надсилається на URL-адресу /auth/login з необхідними обліковими даними.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Модульне тестування

Модульне тестування API передбачає систематичну перевірку окремих компонентів, щоб переконатися, що вони функціонують правильно в ізоляції.

Цей процес гарантує, що обробка запитів, обробка відповідей та обробка помилок працюють за призначенням. Для ефективного модульного тестування API слід застосовувати кілька основних принципів і практик.

По-перше, основна увага при модульному тестуванні API приділяється перевірці поведінки окремих функцій або методів, які складають API. Сюди входять функції, які бізнес-логіку, перевірку даних і обробку помилок. Кожна функція повинна бути протестована з різними вхідними даними, щоб переконатися, що вона дає очікувані результати.

Наприклад, якщо функція обробляє запит на вхід користувача, тести повинні охоплювати дійсні облікові дані, недійсні облікові дані та граничні випадки, такі як порожній ввід або неправильні дані.

Імітація сервісів є важливою технікою в модульному тестуванні API. Оскільки модульні тести мають на меті ізолювати функціональність, що тестується, залежності, такі як бази даних, зовнішні API та інші сервіси, повинні бути замінені на макети об'єктів. Макети імітують поведінку реальних об'єктів і дозволяють тестувальникам зосередитися на коді, що тестується, без зовнішніх впливів.

Наприклад, при тестуванні функції, яка робить запит до бази даних, можна використовувати макет бази даних для повернення заздалегідь визначених даних, усуваючи необхідність підключення до реальної бази даних.

Аби врахувати всі перелічені умови, за основу рішення для модульного тестування було взято популярний фреймворк XUnit.

Розглянемо окремо один із юніт-тестів, аби зрозуміти підхід:

```
[Fact]
public void BookValidator_Validate_ReturnTitleEmpty()
{
    // Arrange
    var request = new CreateBookRequest
```

```
        {
            Title = string.Empty,
            Description = string.Empty,
            Category = default,
            Status = default,
            CoverBase64 = string.Empty,
        };
        var validator = new CreateBookValidator();
        // Act
        var result = validator.TestValidate(request);
        // Assert
        result.ShouldHaveValidationErrorFor(b => b.Title);
    }
```

Тест має чіткий і структурований формат, поділений на три основні частини: Arrange, Act, Assert. Цей формат, відомий як патерн AAA, покращує читабельність та організацію тесту, полегшуючи розробникам розуміння його мети та ходу виконання.

У розділі Arrange створюється об'єкт CreateBookRequest з певними властивостями.

Для параметрів Title, Description і CoverBase64 встановлюються порожні рядки, а для Category і Status - значення за замовчуванням. Таке налаштування готує необхідний контекст для тесту, гарантуючи, що вхідні дані запустять логіку перевірки, яка тестується. Крім того, створюється екземпляр CreateBookValidator, який представляє компонент, що тестується.

У секції Act викликається метод TestValidate CreateBookValidator з об'єктом запиту. Цей метод виконує логіку валідації наданих даних, повертаючи результат, який містить інформацію про будь-які помилки валідації, що виникли. Виконуючи крок перевірки в цій секції, тест ізолює дію, що тестується, даючи зрозуміти, яка поведінка перевіряється.

Нарешті, в секції Assert тест перевіряє, чи присутня в результаті очікувана помилка валідації.

Твердження ShouldHaveValidationErrorFor перевіряє, чи властивість Title об'єкта запиту викликала помилку перевірки. Це твердження перевіряє, чи CreateBookValidator поводиться очікувано, правильно ідентифікуючи порожній заголовок як некоректний.

Test	Duration
Wheelingful.UnitTests (12)	2.4 sec
Wheelingful.UnitTests (12)	2.4 sec
BookAuthorServiceTests (1)	1.1 sec
BookAuthorService_CreateBook_ReturnSuccess	1.1 sec
BookAuthorValidationTests (7)	61 ms
BookValidator_Validate_ReturnCategoryCorrect	< 1 ms
BookValidator_Validate_ReturnCategoryOutOfRa...	2 ms
BookValidator_Validate_ReturnImageCorrect	1 ms
BookValidator_Validate_ReturnImageCorrupted	< 1 ms
BookValidator_Validate_ReturnTitleCorrect	1 ms
BookValidator_Validate_ReturnTitleEmpty	1 ms
BookValidator_Validate_ReturnTitleTooLong	56 ms
BookCoverTests (2)	1 sec
BookCoverService_GetCoverUrl_ReturnDefaultUrl	48 ms
BookCoverService_GetCoverUrl_ReturnUniqueUrl	972 ms
RedisCacheServiceTests (2)	173 ms
RedisCacheService_GetAndSet_ReturnCachedData	7 ms
RedisCacheService_GetAndSet_ReturnFetchedData	166 ms

Рисунок 5.1 – Успішно пройдені модульні тести.

На рисунку 5.1 можна побачити підтвердження, що цей модульний тест та інші з набору були успішно пройдені. Слід навести їх повний список:

– створення книги успішно завершується;
– валідація всіх властивостей книги коректна;
– сервіс правильно формує посилання на обкладинку книги;
– сервіс правильно повертає книги з кешу.

Отже, функціональність частин відповідних сервісів API підтверджено.

5.2 Інтеграційне тестування

Інтеграційні тести відіграють важливу роль у забезпеченні надійності та функціональності програми. Ці тести виходять за рамки модульних тестів, перевіряючи взаємодію між декількома компонентами або системами, забезпечуючи комплексну перевірку всього робочого процесу від початку до кінця.

Інтеграційні тести для API призначені для перевірки взаємодії між кінцевими точками API та базовими сервісами, базами даних і зовнішніми системами, від яких вони залежать. Цей тип тестування гарантує, що API поводиться так, як очікується, в різних сценаріях, правильно обробляючи запити і відповіді та підтримуючи цілісність даних протягом усього процесу.

Основна мета інтеграційних тестів - перевірити, що різні частини системи працюють разом безперебійно. У контексті API це передбачає тестування обробки запитів, виконання бізнес-логіки та рівнів збереження даних. Таким чином, інтеграційні тести допомагають виявити проблеми, які можуть бути неочевидними при ізольованому тестуванні компонентів.

Аби організувати симуляцію відправки та обробки запитів, необхідно відповідно сконфігурувати середовище для тестування. Стандартні потужності ASP.NET на це здатні:

```
public async Task InitializeAsync()
    {
        await _mySqlContainer.StartAsync();
        await _redisContainer.StartAsync();
        Environment.SetEnvironmentVariable("MYSQLCONNSTR_localdb",
            _mySqlContainer.GetConnectionString());
        Factory = new WebApplicationFactory<Program>()
            .WithWebHostBuilder(b =>
            {
                b.UseEnvironment(ApiConstants.BaseEnvironment);
                b.ConfigureServices(s =>
                {
                    RemoveService<ICurrentUser>(s);
                    s.AddScoped<ICurrentUser, MockCurrentUser>();
                });
                b.UseSetting("ConnectionStrings:RedisConnection",
                     _redisContainer.GetConnectionString());
            });
        HttpClient = Factory.CreateClient(new
        WebApplicationFactoryClientOptions
        {
            BaseAddress = new Uri(ApiConstants.BaseUrl),
        });
        using var scope = Factory.Services.CreateScope();
        await using var db = scope.ServiceProvider
            .GetRequiredService<WheelingfulDbContext>();
        await db.Database.MigrateAsync();
    }
```

Метод InitializeAsync в абстрактному класі BaseTest є важливою частиною налаштування тестового середовища для інтеграційних тестів в контексті API. Цей метод збирає всі необхідні служби і залежності, забезпечуючи належним чином готовність до запуску тестів завдяки максимально точному відтворенню умов середовища виконання.

Процес починається з асинхронного запуску контейнерів MySQL і Redis. Ці контейнери забезпечують ізольовані екземпляри MySQL і Redis, гарантуючи, що тести мають доступ до тимчасових контрольованих середовищ. Запускаючи ці контейнери, метод гарантує, що база даних і системи кешування доступні і належним чином ініціалізовані, забезпечуючи послідовний і передбачуваний фон для тестів.

Далі метод встановлює змінну оточення для рядка з'єднання з MySQL. Використовуючи Environment.SetEnvironmentVariable, він динамічно налаштовує рядок з'єднання, дозволяючи програмі підключитися до контейнера MySQL. Цей крок гарантує, що програма, яка тестується, використовуватиме правильний екземпляр бази даних, відповідно до налаштувань конфігурації, що використовуються у реальному розгортанні.

Потім створюэться об'єкт Factory. Ця фабрика необхідна для створення тестового сервера, на якому розміщується додаток. Вона налаштовується за допомогою конструктора веб-хостингу, який встановлює середовище на попередньо визначене значення, вказане в ApiConstants.BaseEnvironment.

У конструкторі веб-хостингу метод ConfigureServices використовується для модифікації колекції сервісів. Зокрема, він видаляє існуючий сервіс ICurrentUser і замінює його імітацією MockCurrentUser.

Крім того, метод UseSetting встановлює рядок з'єднання для Redis, вказуючи на контейнер Redis. Така конфігурація гарантує, що рівень кешування програми взаємодіє з правильним екземпляром Redis, підтримуючи узгодженість з виробничим налаштуванням.

Об'єкт HttpClient створюється за допомогою Factory.CreateClient. Цей клієнт конфігурується з базовою адресою, отриманою з ApiConstants.BaseUrl,

забезпечуючи узгоджену точку входу для всіх запитів API, зроблених під час тестів. Використовуючи цей клієнт, тести можуть взаємодіяти з додатком так само, як і реальний клієнт, надсилаючи HTTP-запити та отримуючи відповіді.

Метод також включає блок scoped, який створює нову область видимості сервісів з фабричних сервісів. У цій області він отримує екземпляр WheelingfulDbContext, контекст бази даних програми. Метод Database.MigrateAsync викликається для застосування всіх очікуваних міграцій до бази даних. Цей крок гарантує, що схема бази даних є актуальною і відповідає поточним вимогам програми, забезпечуючи надійну основу для тестів.

Розглядати приклад інтеграційного тесту необов'язково. За виключенням майже абсолютної відсутності імітаційних сервісів та організації реальних запитів, такий тест відповідає структурі та результатам модульного тесту.

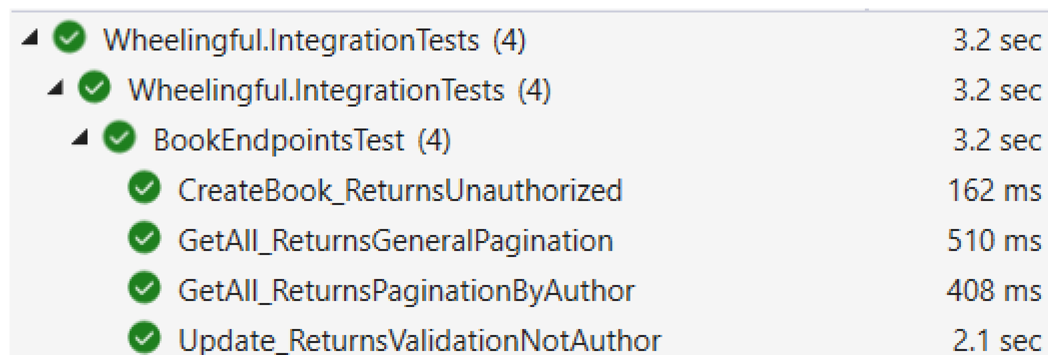


Рисунок 5.2 – Успішно пройдені інтеграційні тести

На рисунку 5.2 можна побачити підтвердження, що всі інтеграційні тести були успішно пройдені. Слід навести їх повний перелік:

– неавторизоване створення книги заблоковано;
– пустий запит списку книг повертає пагінацію за замовчуванням;
– запит списку книг за автором повертає книги автора з пагінацією;
– неавторизоване оновлення книги заблоковано.

Таким чином, функціональність відповідних точок доступу API підтверджено.

6 ОГЛЯД КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ

6.1 Огляд сторінок читача

Список сторінок читача:

– сторінка загального списку книг (див. рис. 6.1);

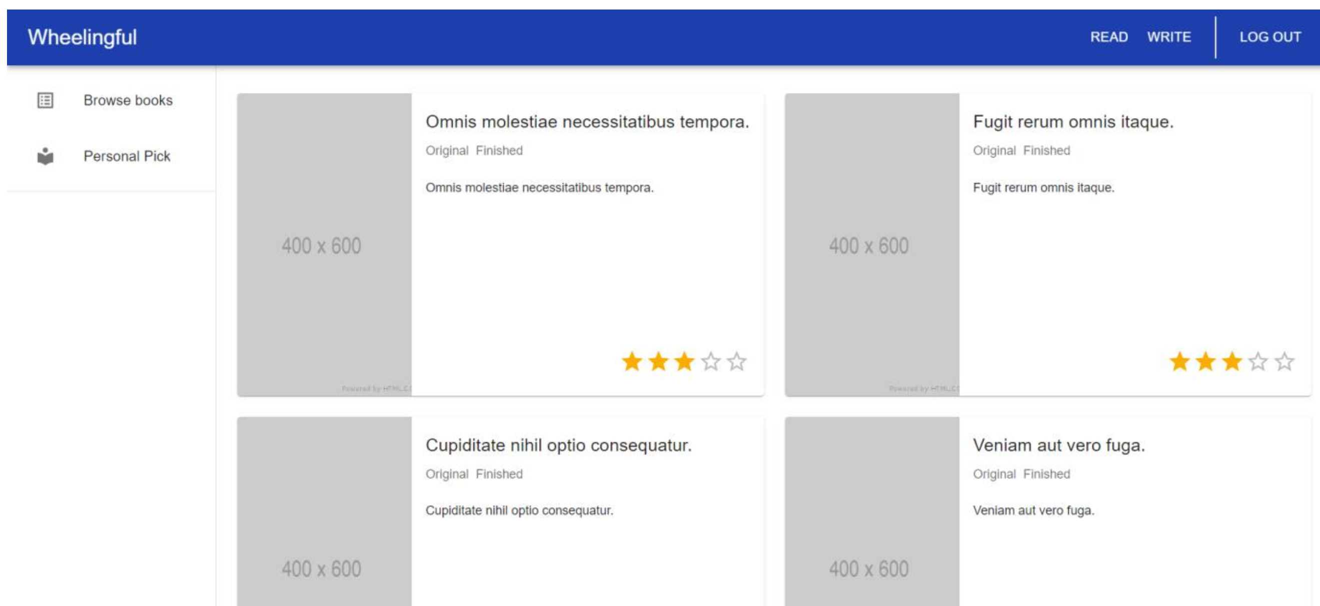


Рисунок 6.1 – Сторінка загального списку книг

– індивідуальна сторінка книги (див. рис. 6.2);

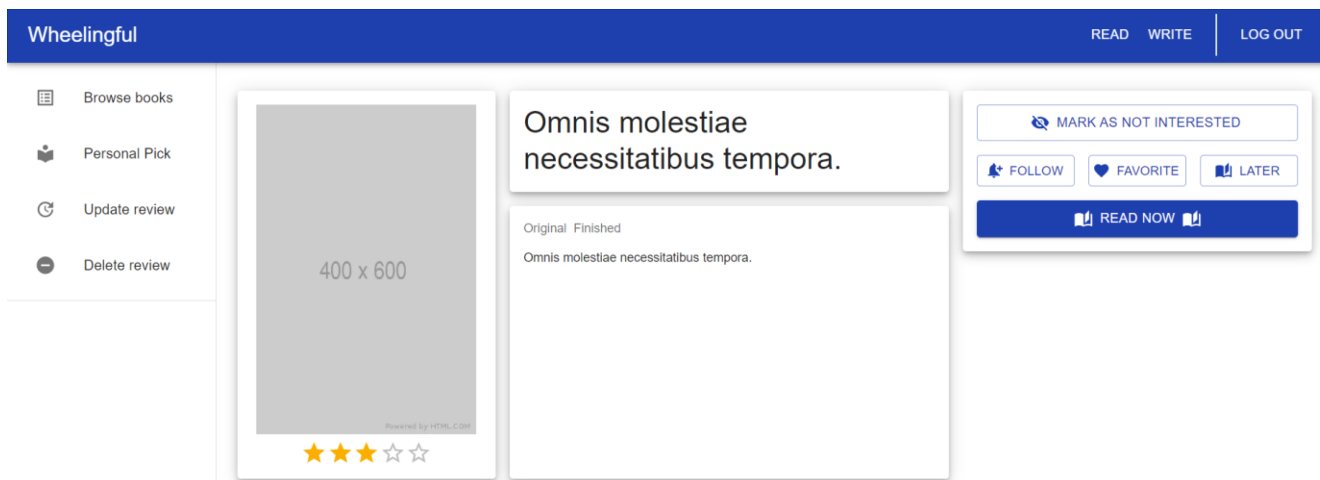


Рисунок 6.2 – Індивідуальна сторінка книги

– список глав на індивідуальній сторінці книги (див. рис. 6.3);

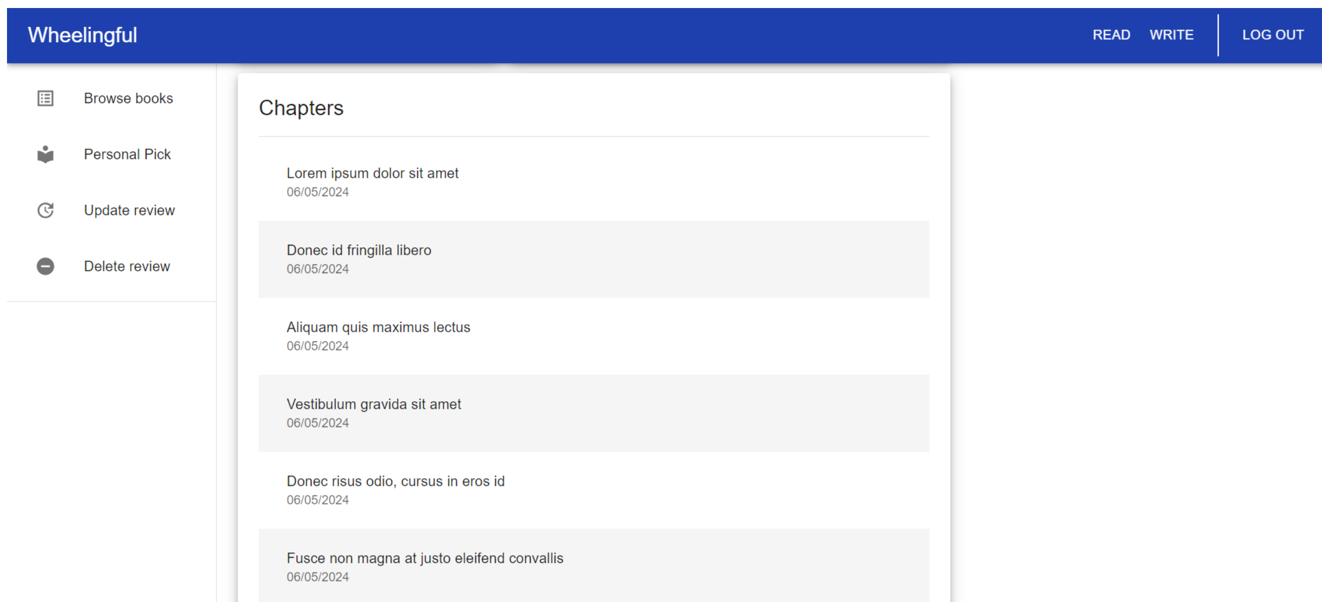


Рисунок 6.3 – Список глав на індивідуальній сторінці книги

– рекомендації книг, схожих до відкритої (див. рис. 6.4)

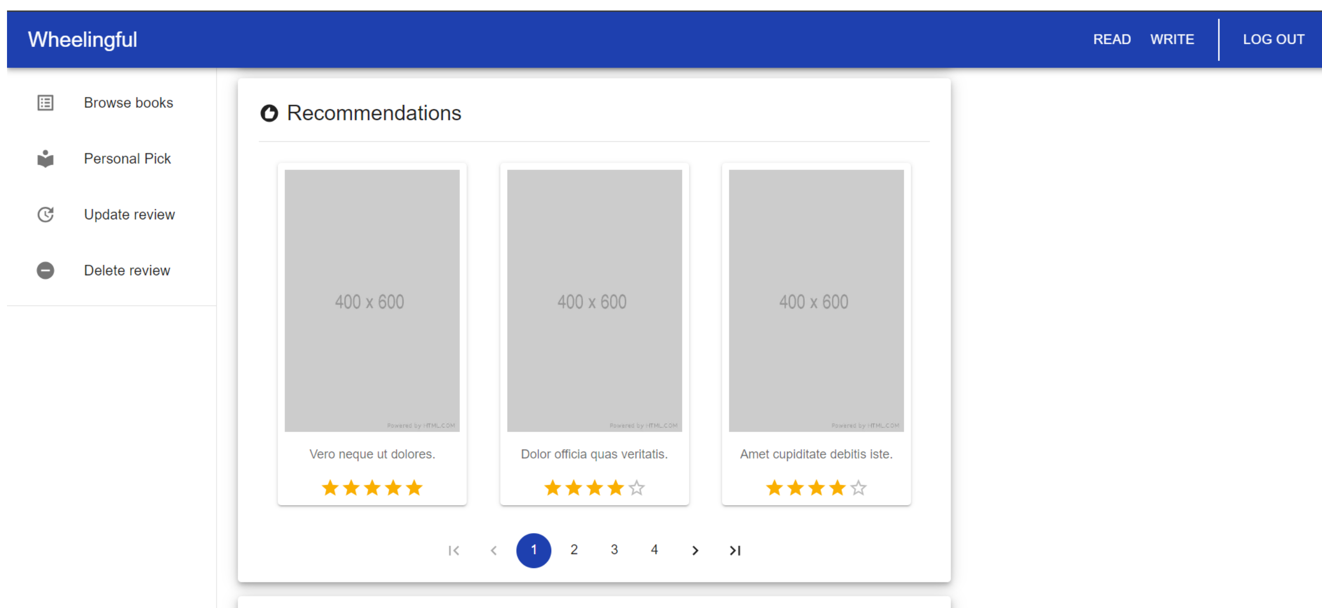


Рисунок 6.4 – Рекомендації книг, схожих до відритої

– секція рецензій користувачів на відкриту книгу (див. рис. 6.5);

– сторінка написання / оновлення власної рецензії (див. рис. 6.6);

– сторінка для читання відкритої глави книги (див. рис. 6.7);

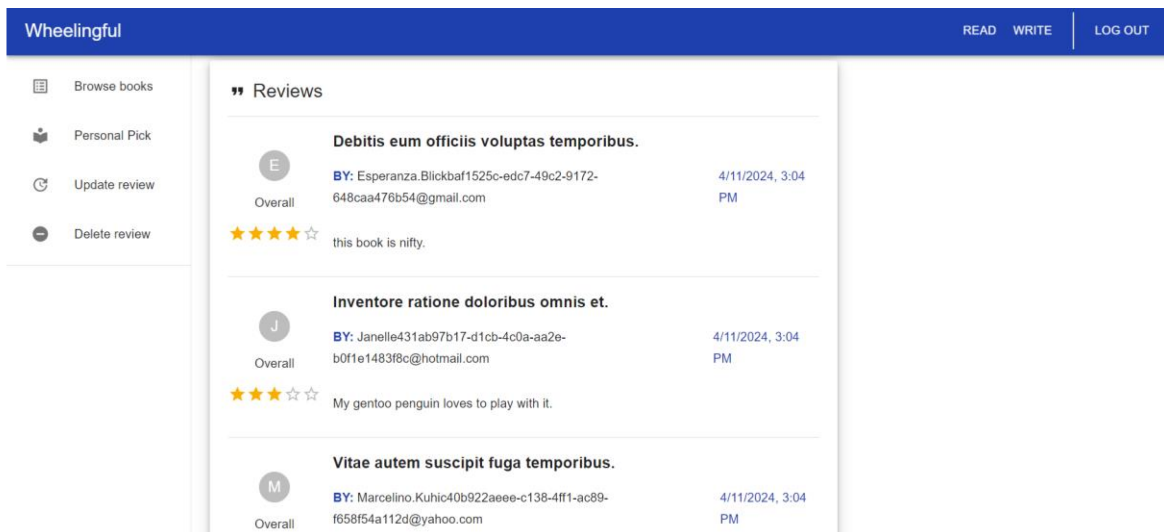


Рисунок 6.5 – Секція рецензій користувачів на відкриту книгу

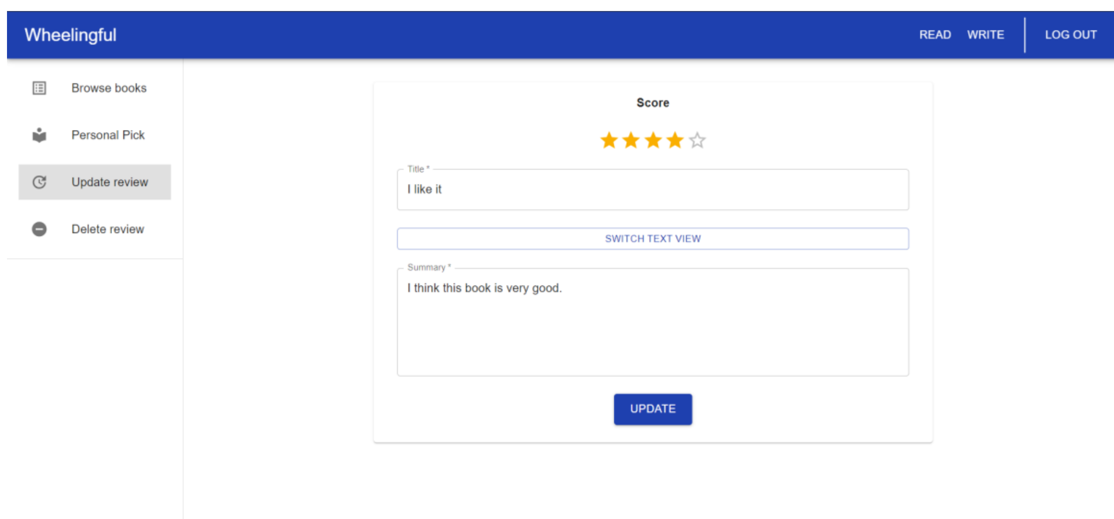


Рисунок 6.6 – Сторінка написання / оновлення власної рецензії

Рисунок 6.7 – Сторінка для читання відкритої глави книги

– сторінка персоналізованих рекомендацій книг (див. рис. 6.8).

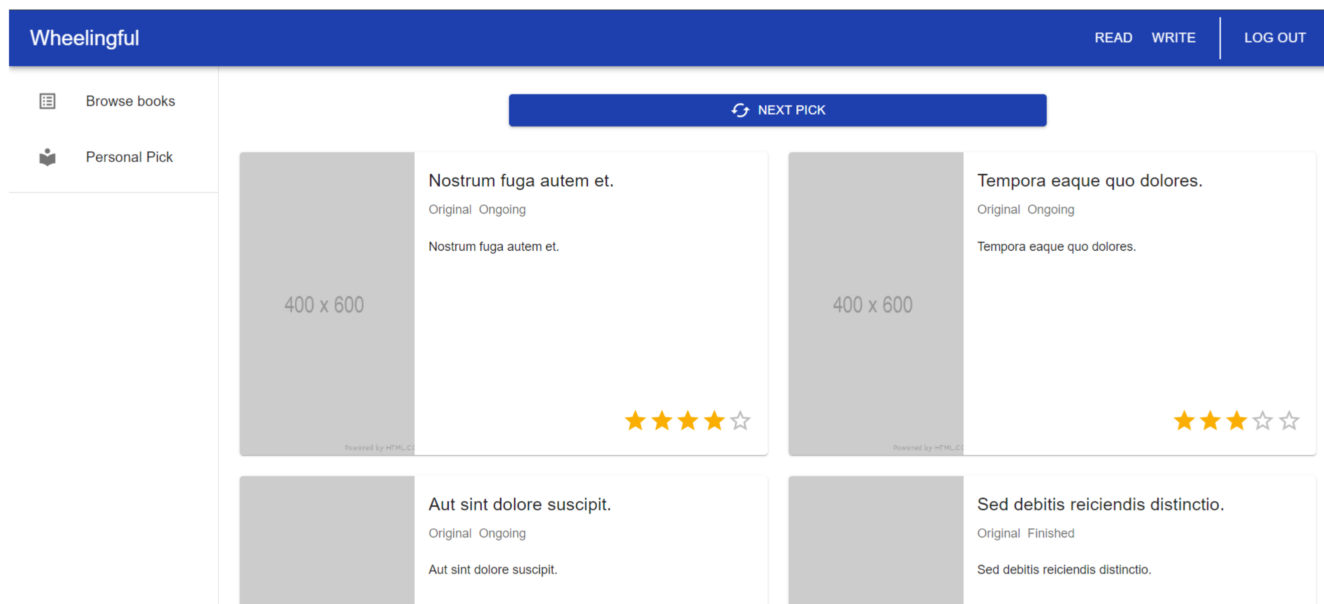


Рисунок 6.8 – Сторінка персоналізованих рекомендацій книг

Слід відзначити, що кожен продемонстрований список підтримує пагінацію, аналогічну до тої, що потрапила на рисунок 6.4.

Також слід відзначити кнопки справа на рисунку 6.2. Більшість із них нефункціональні, але були додані в якості перспективи. Саме через цю нефункціональність в огляді не було продемонстровано можливі пов'язані сторінки.

6.2 Огляд сторінок автора

Список сторінок автора:

– сторінка списку книг, опублікованих авторизованим користувачем (див. рис. 6.9);

– сторінка публікації / оновлення книги (див. рис. 6.10);

– секція глав на сторінці оновлення (не публікації) книги для внесення змін у текст (див. рис. 6.11);

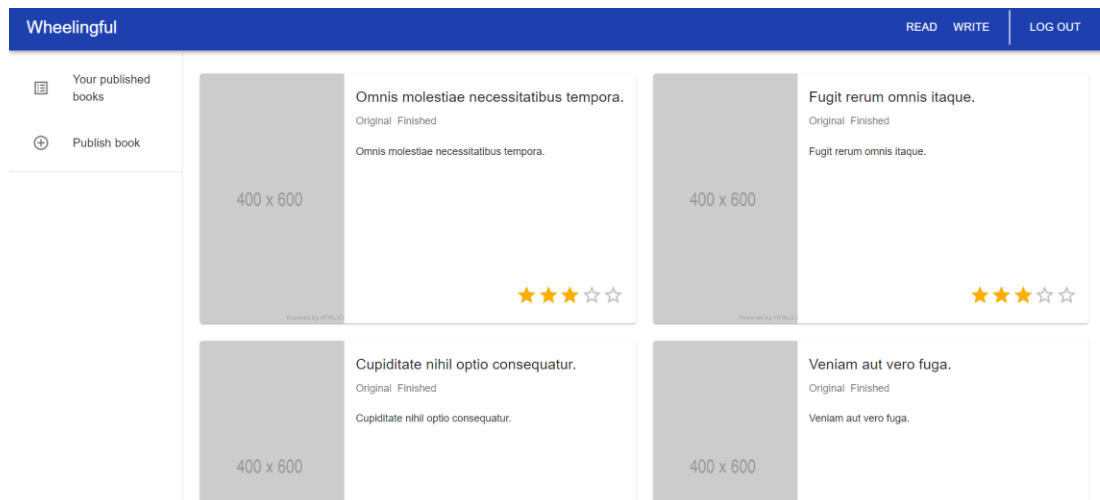


Рисунок 6.9 – сторінка списку книг, опублікованих авторизованим користувачем

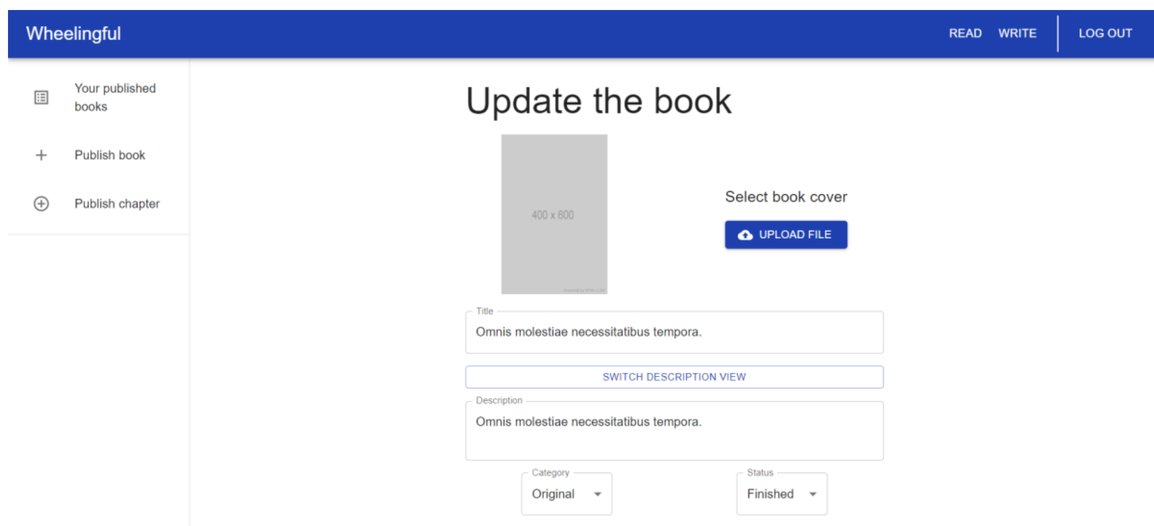


Рисунок 6.10 – Сторінка публікації / оновлення книги

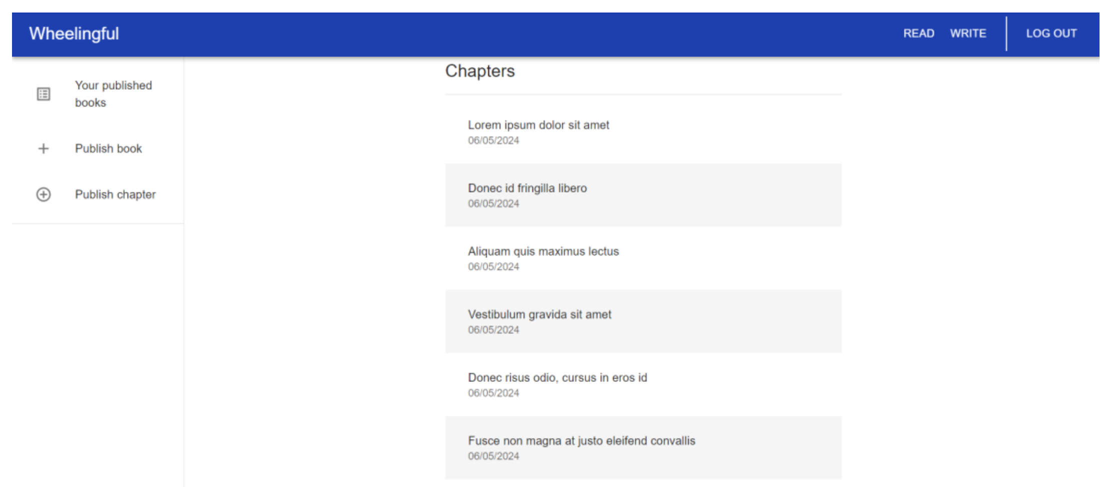


Рисунок 6.11 – Секція глав на сторінці оновлення (не публікації) книги

– сторінка публікації / оновлення глави з видимою розміткою (див. рис. 6.12);

Wheelingful READ WRITE LOG OUT

Your published books

Publish book

Publish chapter

Publish a new chapter

Title

Hello, World

SWITCH TEXT VIEW

Text

Hello

This is the beginning of the **new chapter** in *my life*.

It is going to be wonderful!

PUBLISH

Рисунок 6.12 – Сторінка публікації / оновлення глави з видимою розміткою

– сторінка публікації / оновлення глави з обробленою розміткою (див рис. 6.13).

Wheelingful READ WRITE LOG OUT

Your published books

Publish book

Publish chapter

Publish a new chapter

Title

Hello, World

SWITCH TEXT VIEW

Hello

This is the beginning of the **new chapter** in *my life*.

It is going to be wonderful!

PUBLISH

Рисунок 6.13 – Сторінка публікації / оновлення глави з обробленою розміткою

Слід відзначити, що сторінки створення та оновлення книг чи книжних глав ідентичні по своєму зовнішньому вигляду, тому до огляду було додано лише по одному такому прикладу, кожен із яких краще демонструє моменти, важливі для сприйняття розробленого програмного продукту.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було спроектовано та розроблено програмну систему таргетингу за інтересами для платформи самостійного видавництва та читання художньої літератури. Для неї застосовано клієнт-серверну архітектуру, тож вона складається з серверної частини, до складу якої входить рекомендаційна система та база даних, а також клієнтської частини, що представлена веб-додатком.

У процесі було проведено аналіз предметної області, зокрема конкурентів на ринку, а також визначено вимоги до програмної системи та архітектури програмної системи. Дослідження допомогло якомога точніше охарактеризувати очікування та бажаний стан проєкту як MVP.

Обраною архітектурою для серверної частини стане класична трирівнева архітектура, застосування якої охоплювало API та бізнес-логіку, що за ним стоїть, зокрема рекомендаційну систему. Тоді архітектурою клієнтської частини стане вертикальна архітектура, що задовольняло потреби розробки по ходу додавання точок доступу.

Розробка стане можливою за допомогою застосування такого стеку .NET-технологій, як ASP.NET, ML.NET, Entity Framework. Тоді для реалізації веб-додатку застосовувався стек JS-технологій, зокрема React-фреймворк та Redux.

Проєкт успішно реалізує мету створення програмної системи для організації роботи хабу поширення художньої літератури. Основні функції включають персоналізовані рекомендації книг читачам, читання книжних глав, а також можливість авторів додавати й публікувати свої твори. Система надає ефективний підбір релевантного контенту для забезпечення доступу читачів до різноманітних творів в умовах інформаційного перенасичення. Додатково, вона дозволяє авторам самостійно публікувати художні твори, уникнувши залежності від традиційних видавництв, а також сприяє агрегації контенту і формуванню ком’юніті.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. How many videos are uploaded to TikTok daily? [Електронний ресурс] – URL: https://techjury.net/blog/how-many-videos-are-uploaded-to-tiktok-daily/ (дата посилання: 06.05.2024).
2. Pros and Cons of Traditional Publishing: Is It Right for You? [Електронний ресурс] – URL: https://www.alyssamatesic.com/free-writing-resources/pros-and-cons-of-traditional-publishing (дата посилання: 06.05.2024).
3. Why people create video content? [Електронний ресурс] – URL: https://www.emerald.com/insight/content/doi/10.1108/INTR-06-2018-0270/full/html (дата посилання: 06.05.2024).
4. Method of forming recommendations using temporal constraints in a situation of cyclic cold start of the recommender system. [Електронний ресурс] – URL: https://openarchive.nure.ua/entities/publication/66431f52-f888-4f0e-8eb6-e272872ad158 (дата посилання: 06.05.2024).
5. Microsoft .NET documentation [Електронний ресурс] – URL: https://learn.microsoft.com/en-us/ (дата посилання: 06.05.2024).
6. React – the library for web and native user interfaces [Електронний ресурс] – URL: https://react.dev/ (дата посилання: 06.05.2024).
7. BigChaos solution to Netflix Grand Prize [Електронний ресурс] – URL: https://www.researchgate.net/publication/223460749_The_BigChaos_Solution_to_the_Netflix_Grand_Prize (дата посилання: 06.05.2024).
8. Matrix factorization for Netflix Grand Prize [Електронний ресурс] – URL: https://dl.acm.org/doi/10.1145/1454008.1454049 (дата посилання: 06.05.2024).

ДОДАТОК А

Звіт результатів перевірки на унікальність у базі ХНУРЕ

Ім'я користувача:
Олійник Олена Володимирівна каф. ПІ

ID перевірки:
1016335286

Дата перевірки:
08.06.2024 14:45:40 EEST

Тип перевірки:
Doc vs Library

Дата звіту:
08.06.2024 14:47:19 EEST

ID користувача:
100012353

Назва документа: 2024_Б_ПІ_ПЗПІ-20-4_Онищенко_М_Г_скорочений_.pdf

Кількість сторінок: 51 Кількість слів: 8949 Кількість символів: 75254 Розмір файлу: 1.37 MB ID файлу: 1016135926

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

3.82% Схожість

Найбільша схожість: 0.99% з джерелом з Бібліотеки (ID файлу: 1008296708)

Пошук збігів з Інтернетом не проводився

3.82% Джерела з Бібліотеки 243 .... Сторінка 53

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0% Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи 7

Підозріле форматування 16 сторінок

ДОДАТОК Б

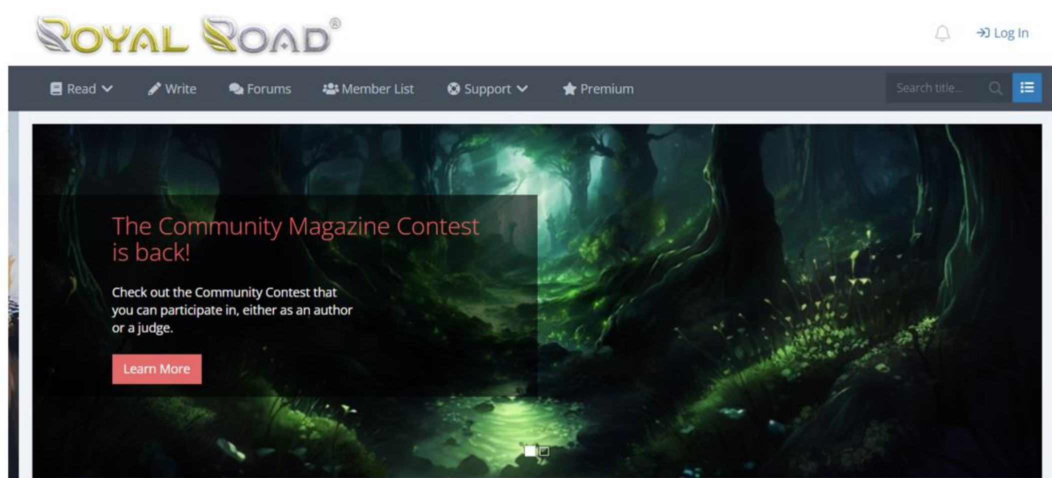
Слайди презентації

Програмна система таргетингу за інтересами для платформи самостійного видавництва та читання художньої літератури

ВИКОНАВ: СТ. ГР. ПЗПІ-20-4 ОНИЩЕНКО М.Г.
НАУКОВИЙ КЕРІВНИК: ДОЦ. КАФ. ВОРОЧЕК О.Г.

1

Аналіз конкурентів – Royal Road



2

Аналіз конкурентів – Royal Road

Переваги:

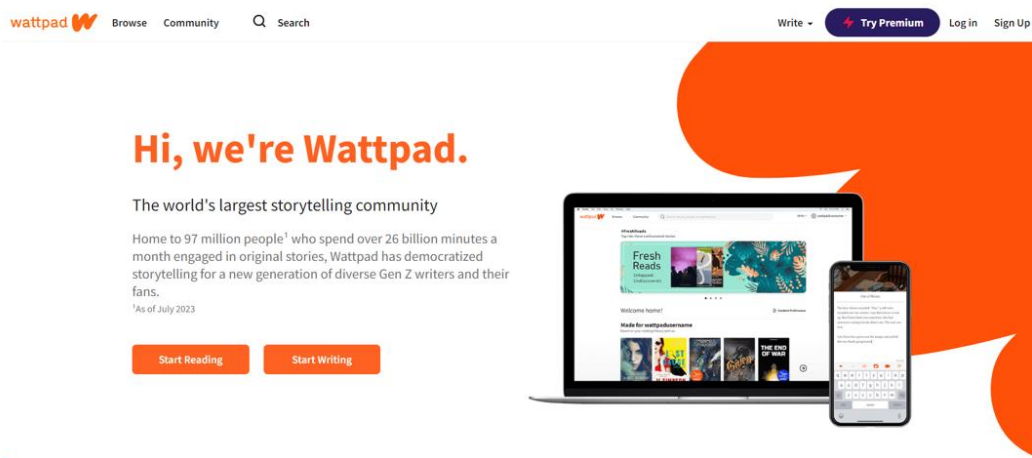
- актуальні рекомендації творів, що набирають популярність;
- широке проактивне ком'юніті.

Недоліки:

- повна відстуність персоналізованих рекомендацій;
- застарілий інтерфейс.

3

Аналіз конкурентів – Wattpad



4

Аналіз конкурентів – Wattpad

Переваги:

- персоналізовані рекомендації, засновані на історії взаємодій;

- сучасний інтерфейс.

Недоліки:

- відсутність актуального поширення свіжих творів;

5

Постановка задачі

Коротко про головне:

- підбирати релевантні книги згідно підходів персоналізації, використовуючи історію взаємодій користувачів із платформою;

- надавати базові інструменти читання книг за главами онлайн без необхідності завантаження;

- надавати базові інструменти самостійного видавництва книг онлайн глава за главою.

6

Типи користувачів

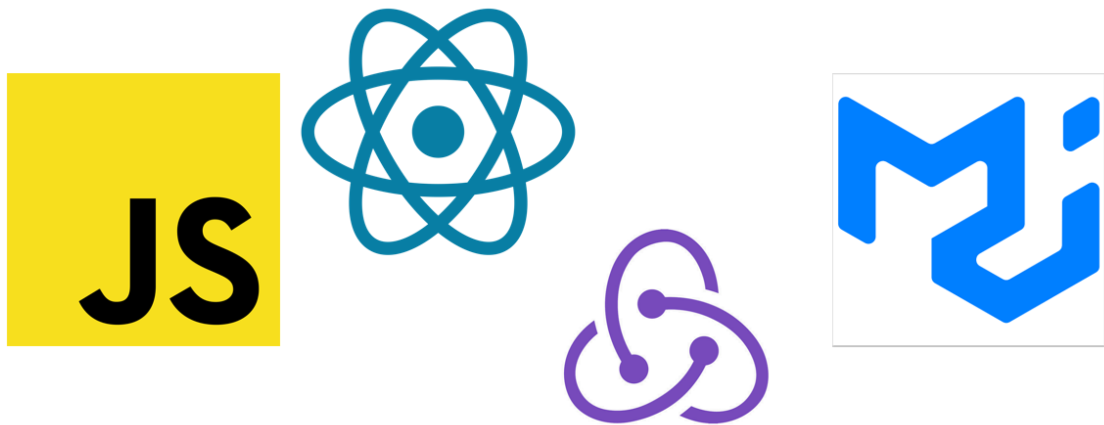
Читач

Автор

Attributions:
https://www.flaticon.com/free-icons/writer
https://www.flaticon.com/free-icons/reading

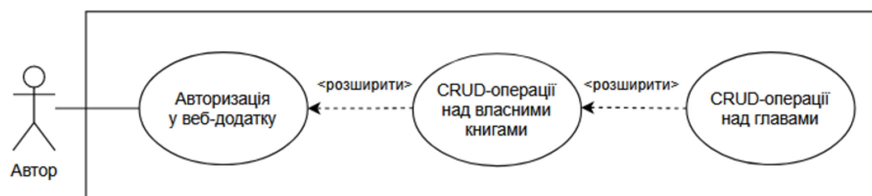
Обрані технології – серверна частина

Обрані технології – клієнтська частина



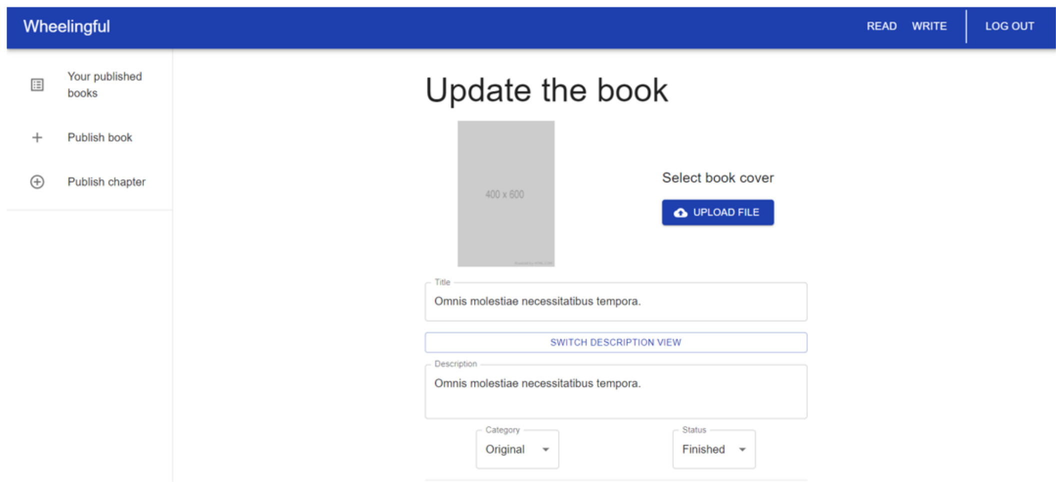
9

Архітектура та проєктування

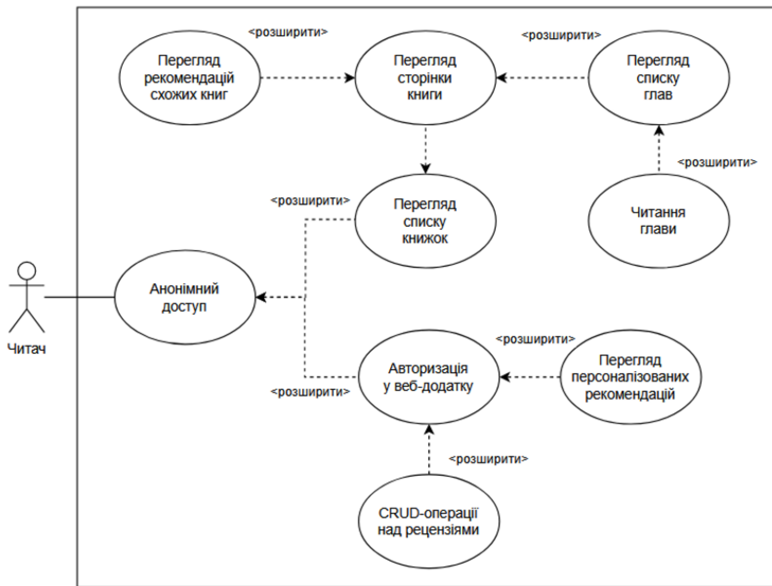


10

Огляд програмної системи



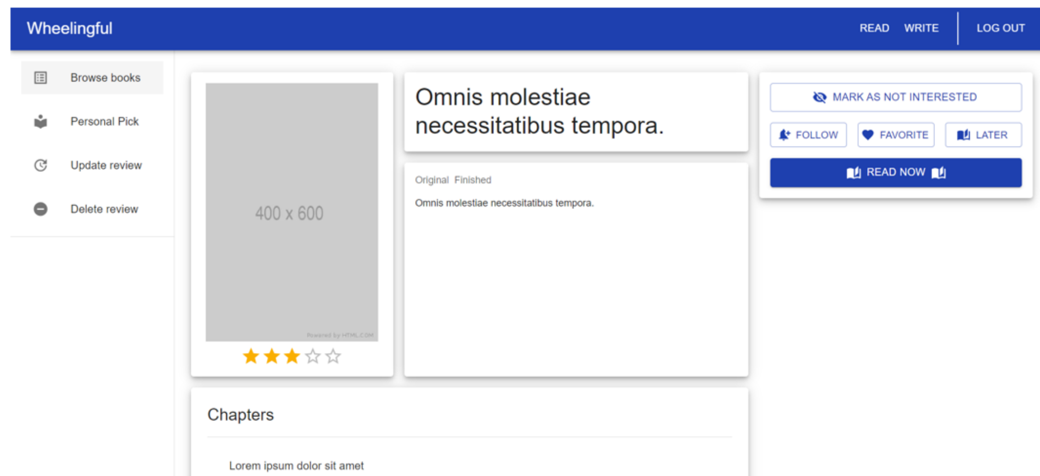
11



Архітектура та проєктування

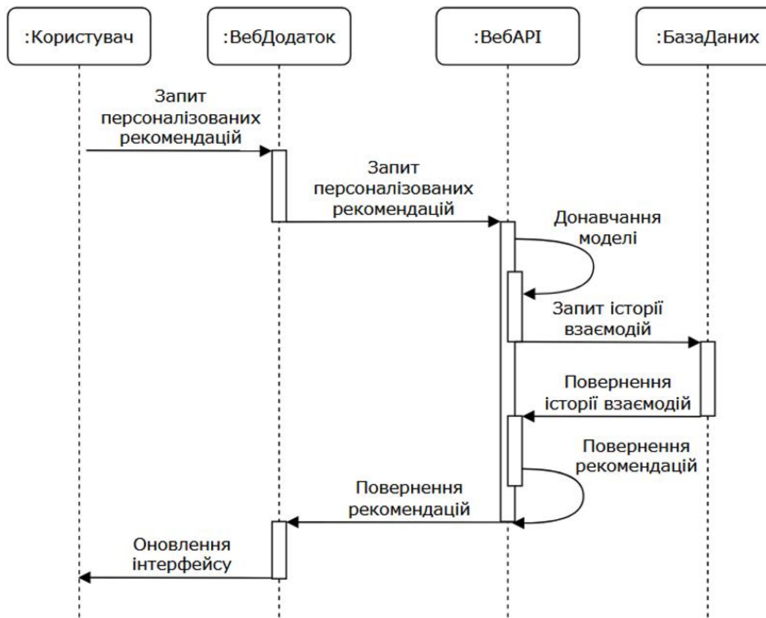
12

Огляд програмної системи



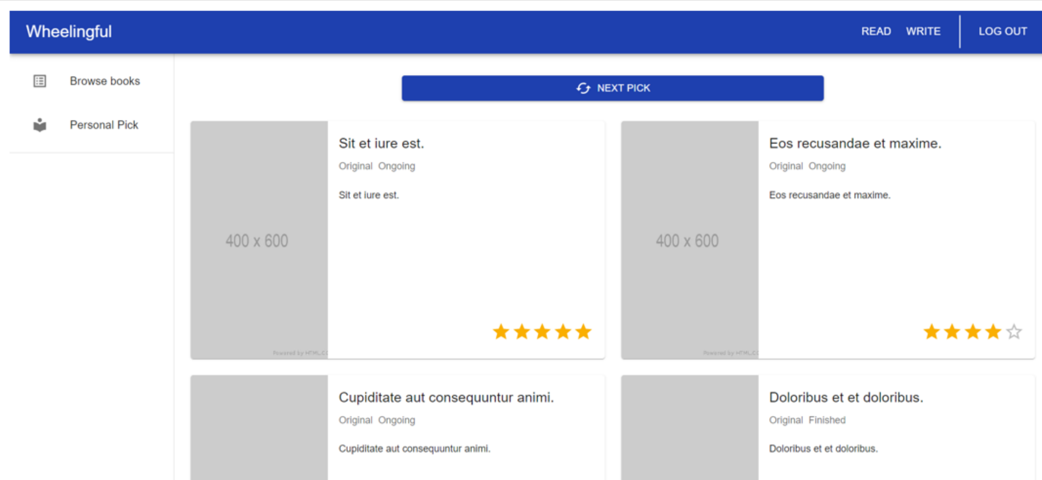
13

Архітектура та проєктування



14

Огляд програмної системи



15

Прийняті програмні рішення

```
var splitData = mlContext.Data.TrainTestSplit(
    data: mlContext.Data.LoadFromEnumerable(reviews),
    testFraction: 0.2
);

IEstimator<ITransformer> estimator = mlContext.Transforms.Conversion
    .MapValueToKey(outputColumnName: "UserIdEncoded", inputColumnName: "UserId")
    .Append(mlContext.Transforms.Conversion
    .MapValueToKey(outputColumnName: "BookIdEncoded", inputColumnName: "BookId"));

var options = new MatrixFactorizationTrainer.Options
{
    MatrixColumnIndexColumnName = "UserIdEncoded",
    MatrixRowIndexColumnName = "BookIdEncoded",
    LabelColumnName = "ReviewScore",
    NumberOfIterations = 20,
    ApproximationRank = 100
};

var trainerEstimator = estimator
    .Append(mlContext.Recommendation().Trainers.MatrixFactorization(options));

ITransformer model = trainerEstimator.Fit(splitData.TrainSet);

var prediction = model.Transform(splitData.TestSet);

var metrics = mlContext.Regression
    .Evaluate(prediction, labelColumnName: "ReviewScore", scoreColumnName: "Score");
```

16

Прийняті програмні рішення

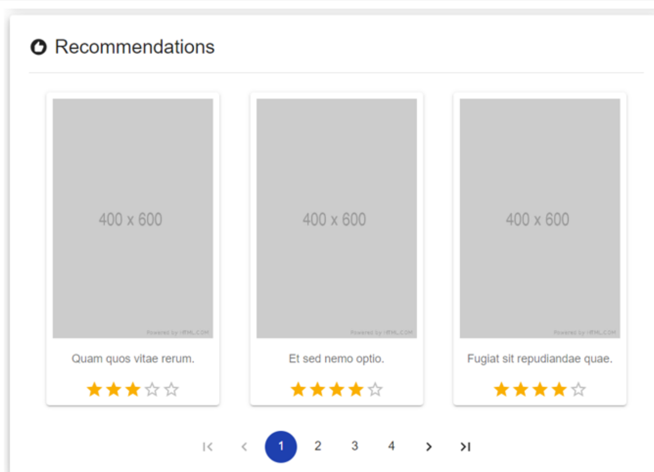
```
return books
    .Select(b =>
    {
        var modelInput = new ModelInput
        {
            BookId = b.Id,
            UserId = currentUser.Id
        };

        var modelOutput = prediction.Predict(modelInput);

        return new { Book = b, PredictedScore = modelOutput.Score };
    })
    .Where(o => !currentUserReviews.Any(r => r.BookId == o.Book.Id)
            && o.PredictedScore >= MlConstants.ScoreMinThreshold
            && o.PredictedScore <= MlConstants.ScoreMaxThreshold)
    .OrderByDescending(o => o.PredictedScore)
    .Select(o => new FetchBookResponse
    {
        Id = o.Book.Id,
        Title = o.Book.Title,
        Description = o.Book.Description,
        Category = o.Book.Category,
        Status = o.Book.Status,
        CoverUrl = bookCover.GetCoverUrl(o.Book.Id, o.Book.AuthorUserId, o.Book.CoverId),
        AuthorUserName = o.Book.AuthorUserName,
        Reviews = o.Book.Reviews,
    })
    .Take(PaginationConstants.DefaultPageSize);
```

17

Огляд програмної системи



18

Тестування програмного забезпечення

```
// Arrange

var imageMock = MockImageService();

var loggerMock = MockServicesHelper.MockLogger<BookCoverService>();

var optionsMock = MockServicesHelper.MockOptions(new BookCoverOptions...);

using var context = new TestDbContextFactory().CreateDbContext();

var book = new Book...;

context.Books.Add(book);

context.SaveChanges();

var bookCoverService = new BookCoverService(imageMock, loggerMock, optionsMock);

// Act

var expected = $"{BookCoverConstants.Endpoint}/{BookCoverConstants.Folder}/{BookCoverConstants.Def
var result = bookCoverService.GetCoverUrl(book.Id, DbConstants.AdminUserId, book.CoverId);

// Assert

Assert.NotNull(result);
Assert.NotEmpty(result);
Assert.Equal(expected, result);
```

19

Тестування програмного забезпечення

```
// Arrange

using var scope = Factory.Services.CreateScope();

await using var db = scope.ServiceProvider.GetRequiredService<WheelingfulDbContext>();

var newUserId = await AuthHelper.RegisterUser(scope, "new.user@gmail.com", "#123Admin");

var books = new List<Book>
{
    new Book...,
    new Book...,
};

db.AddRange(books);

await db.SaveChangesAsync();

// Act

var result = await HttpClient.GetFromJsonAsync<FetchPaginationResponse<FetchBookResponse>>("/books?

// Assert

Assert.NotNull(result);
Assert.Equal(2, result.Items.Count);
```

20

ДЯКУЮ ЗА УВАГУ ! ! !

ДОДАТОК В

Діаграма прецедентів

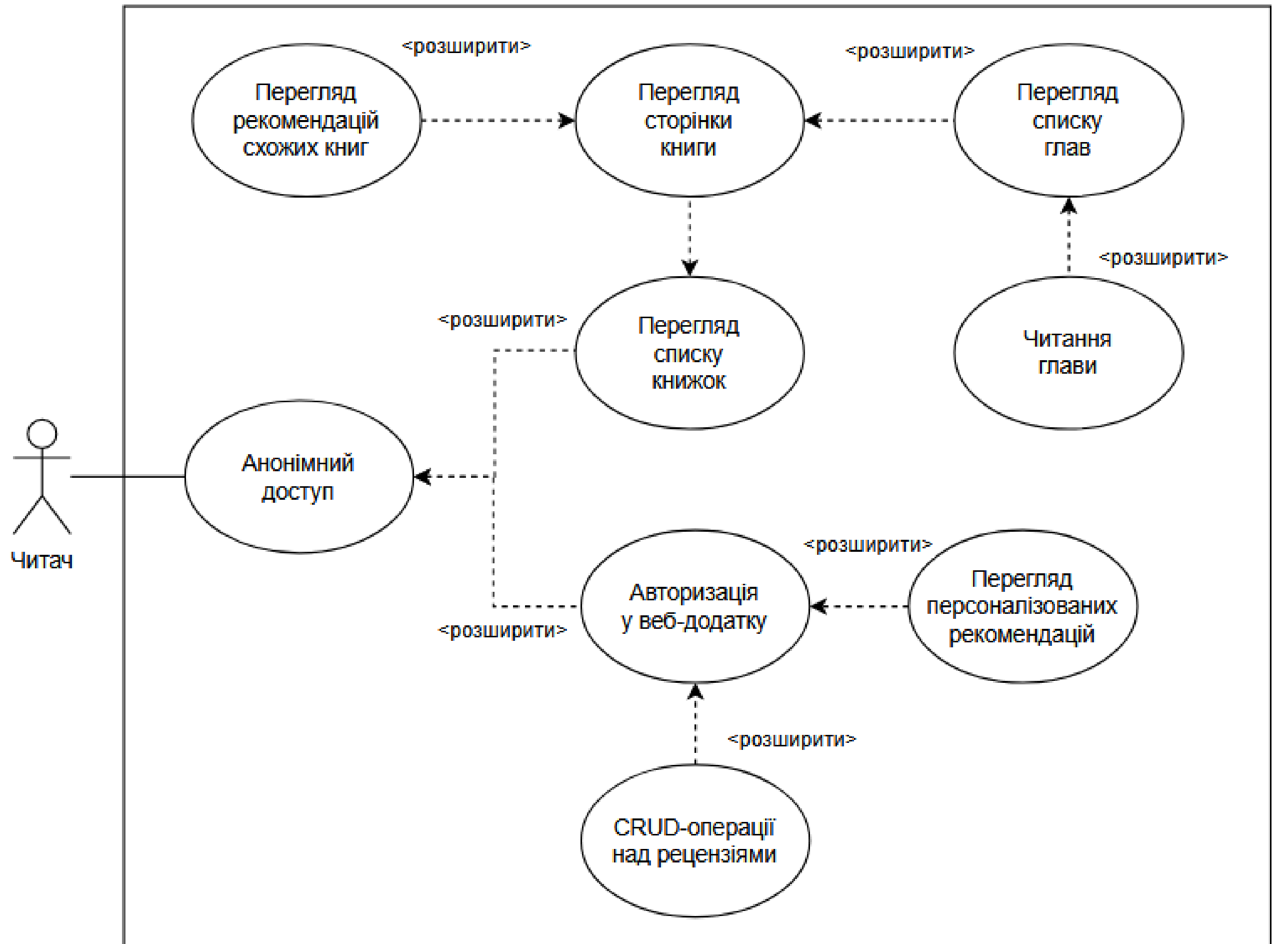


Рисунок В.1 – Use Case читача

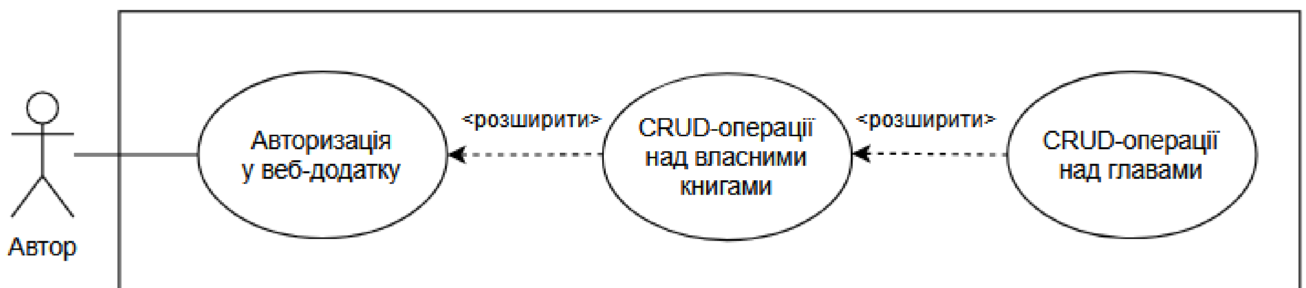


Рисунок В.2 – Use Case автора

ДОДАТОК Г

Діаграма пакетів

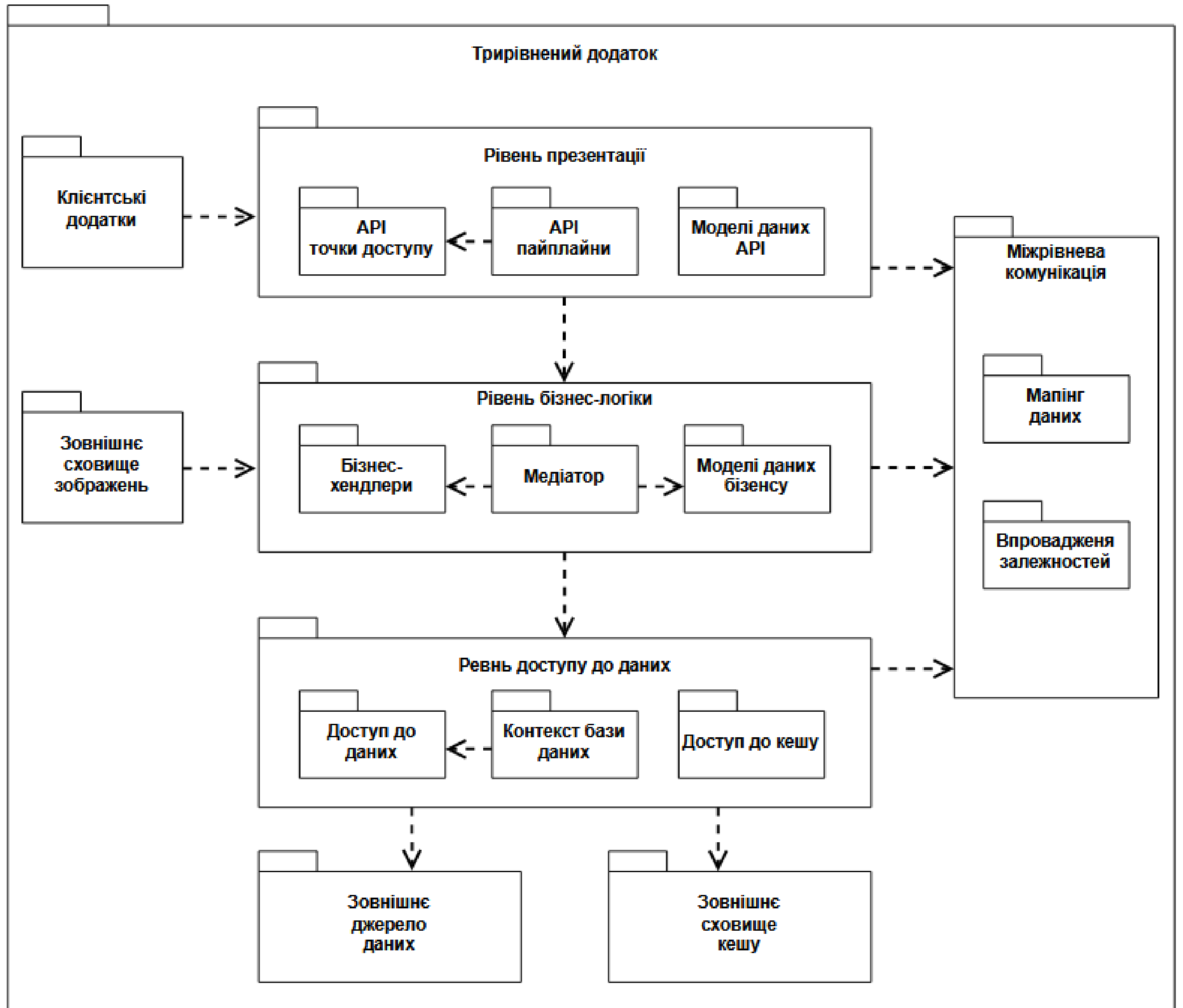


Рисунок Г.1 – Діаграма пакетів серверного додатку

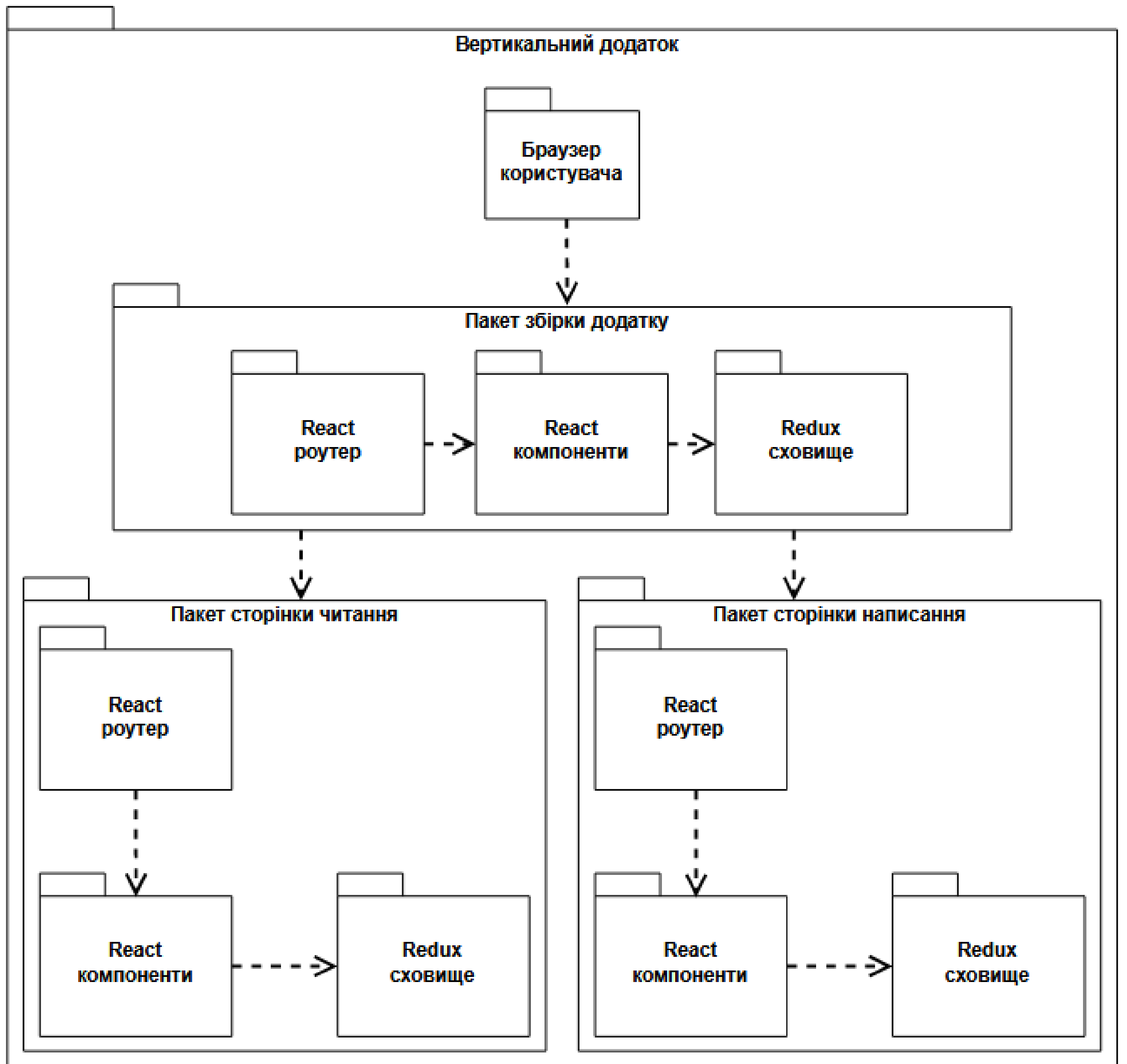


Рисунок Г.2 – Діаграма пакетів клієнтського додатку

ДОДАТОК Д

Діаграма послідовності

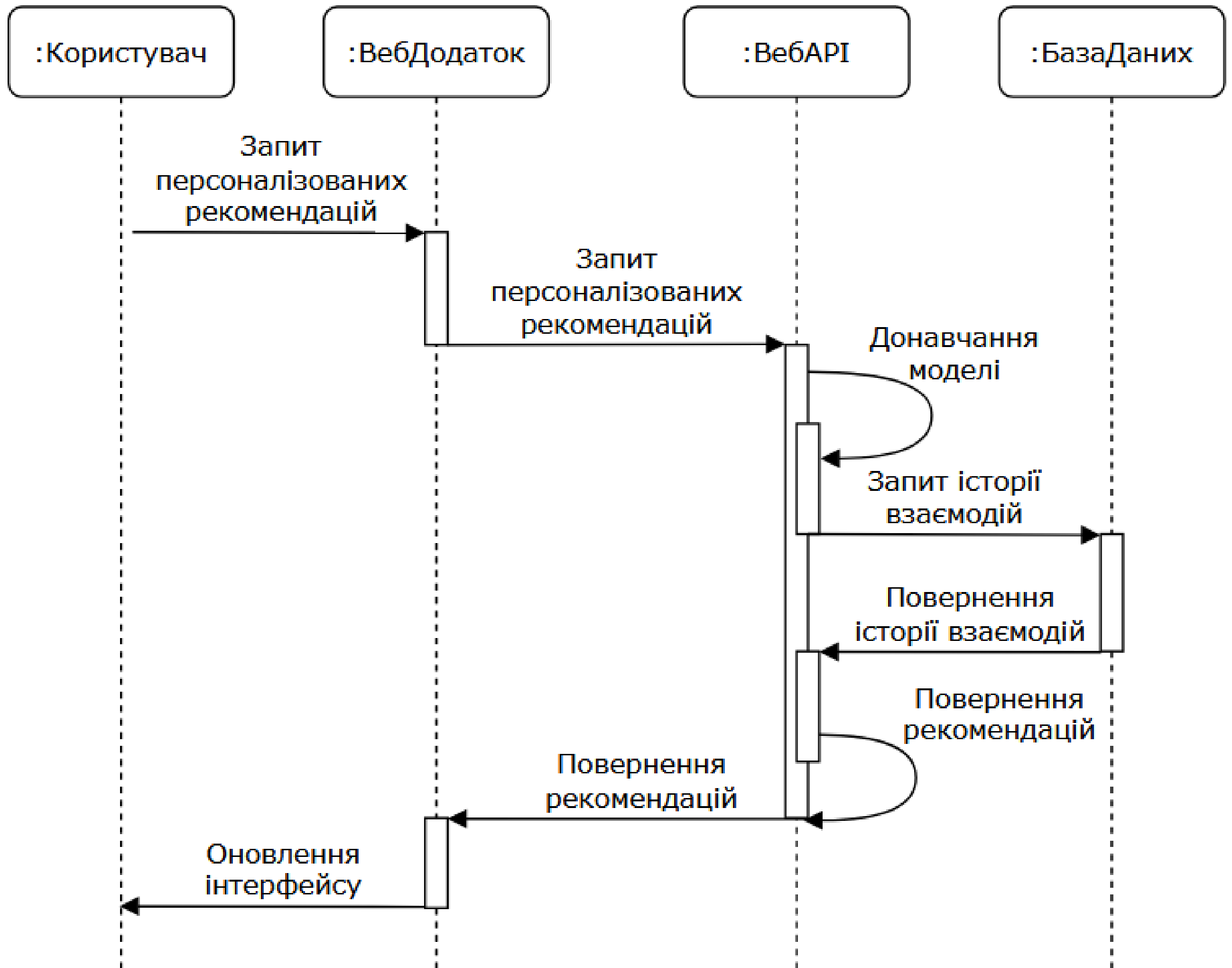


Рисунок Д.1 – Діаграма послідовності для персоналізованих рекомендацій

ДОДАТОК Е

Діаграма розгортання

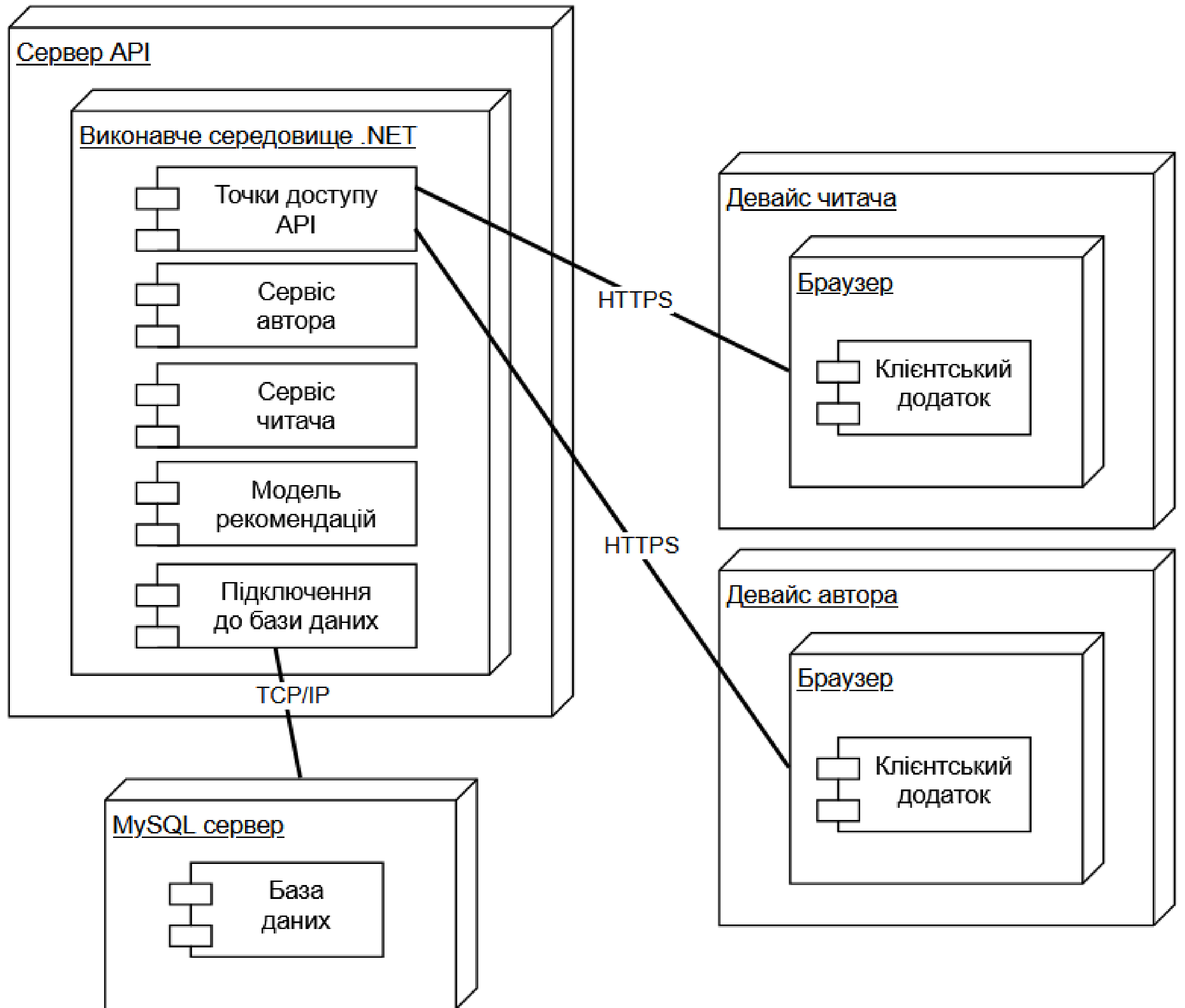


Рисунок Е.1 – Діаграма розгортання програмної системи