ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка web-застосунку для обміну книжковими вподобаннями»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

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

_____ Тетяна КОЛОСЕНКО
(підпис)

Виконала: здобувачка вищої освіти групи ПД-41

Тетяна КОЛОСЕНКО

Керівник: Оксана ЗОЛОТУХІНА
к.т.н., доцент

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Колосенко Тетяні Євгенівні

1. Тема кваліфікаційної роботи: «Розробка web-застосунку для обміну книжковими вподобаннями»
керівник кваліфікаційної роботи к.т.н., доцент Оксана ЗОЛОТУХІНА,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: технічна документація, інструкції з використання прикладних програмних інтерфейсів, а також методичні матеріали щодо створення вебзастосунків із соціальною взаємодією користувачів.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд та аналіз предметної області, дослідження цифрових бібліотек та онлайн-сервісів з рекомендаційною і соціальною складовою.
2. Дослідження існуючих рішень, виявлення їх особливостей та обґрунтування необхідності створення нового web-застосунку для організації особистих бібліотек.

3.Формування функціональних та нефункціональних вимог до web-застосунку для ведення особистої бібліотеки та системи рекомендацій.
4. Проєктування архітектури клієнт-серверного web-застосунку з підтримкою розмежування прав доступу
5. Програмна реалізація web-застосунку із використанням ASP.NET Core, Razor Pages, Entity Framework Core та SQL Server.
6. Тестування основного функціоналу застосунку.

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма класів.
5. Діаграма діяльності.
6. Екранні форми.
7. Апробація результатів дослідження

6. Дата видачі завдання «25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Визначення функціональних та нефункціональних вимог.	14.03-18.03.2025	
4	Проєктування архітектури web-застосунку для обміну книгами.	19.03-23.03.2025	
5	Реалізація функціонал користувацької бібліотеки.	24.03-17.04.2025	
6	Проведення тестування основного функціоналу web-застосунку.	18.04-28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувачка вищої освіти _____ Тетяна КОЛОСЕНКО
(підпис)

Керівник
кваліфікаційної роботи _____ Оксана ЗОЛОТУХІНА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 68 стор., 2 табл., 22 рис., 19 джерел

Мета роботи - цифровізація процесу обміну книжковими вподобаннями між читачами завдяки використанню web-застосунку.

Об'єкт дослідження - процес обміну книжковими вподобаннями.

Предмет дослідження - технології та інструменти розробки web-застосунку для обміну книжковими вподобаннями.

Для реалізації поставленої мети потрібно вирішити наступні завдання:

1. Провести аналіз предметної області цифрових бібліотек та онлайн-сервісів з рекомендаційною і соціальною складовою.
2. Дослідити існуючі рішення, виявити їх особливості та обґрунтувати необхідність створення нового web-застосунку для організації особистих бібліотек.
3. Сформувати функціональні та нефункціональні вимоги до web-застосунку для ведення особистої бібліотеки та системи рекомендацій.
4. Спроєктувати архітектуру клієнт-серверного web-застосунку з підтримкою розмежування прав доступу.
5. Реалізувати web-застосунок із використанням ASP.NET Core, Razor Pages, Entity Framework Core та SQL Server.
6. Провести тестування функціональності web-застосунку.

Короткий зміст роботи - в роботі проаналізовано особливості створення соціально-орієнтованих WEB-застосунків для керування особистими бібліотеками на основі відкритих даних. Розглянуто структуру та функціональні можливості аналогічних сервісів, зокрема Goodreads, LibraryThing та Knigoholic. Розроблено архітектуру WEB-застосунку, орієнтованого на взаємодію користувачів із книгами та

друзями. Програмно реалізовані ключові функціональні можливості: пошук і додавання книг через OpenLibrary API, ведення персональної бібліотеки з можливістю встановлення статусу читання, оцінювання та коментування книг, перегляд бібліотек друзів, відображення аватарів, а також рекомендації на основі жанрових уподобань і соціальних зв’язків.

В роботі використано платформу ASP.NET Core з Razor Pages, Entity Framework Core для взаємодії з базою даних SQL Server, JavaScript для динамічного оновлення інтерфейсу, а також бібліотеку Bootstrap для адаптивного дизайну. Проведено функціональне тестування основних модулів системи.

Сферою використання застосунку є організація ведення особистої бібліотеки користувачами з можливістю отримання рекомендацій на основі читацьких уподобань та соціальної взаємодії.

КЛЮЧОВІ СЛОВА: WEB-ЗАСТОСУНОК, ОСОБИСТА БІБЛІОТЕКА, РЕКОМЕНДАЦІЇ, КОРИСТУВАЧ, КНИГА, РЕЙТИНГ, ДРУЗІ, АВТОРИЗАЦІЯ.

ЗМІСТ

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ 13
1.1 Сучасні підходи до створення онлайн-бібліотек і читацьких платформ 13
1.2 Goodreads 14
1.3 LibraryThing 16
1.4 Knigoholic 19
1.5 Інтеграція з відкритими API (OpenLibrary) 20
1.6 Висновки до розділу 1 21
3. ПРОЄКТУВАННЯ І АРХІТЕКТУРА WEB-ЗАСТОСУНКУ 24
2.1 Постановка задачі та технічне завдання 24
2.2 Вибір стеку технологій 26
2.3 Моделювання структури даних 38
3. РЕАЛІЗАЦІЯ ФУНКЦІОНАЛУ ВЕБ-СИСТЕМИ 42
3.1 Функціонал додавання книг через OpenLibrary API 42
3.2 Збереження персональної бібліотеки 45
3.4 Система рекомендацій за вподобаннями друзів 49
3.5 Аватар користувача, профіль 51
3.6 Система оцінювання та коментування 52
3.7 Адміністративна частина: керування користувачами та ролями 55
3.8 Висновки до розділу 3 56
4. ТЕСТУВАННЯ WEB-ЗАСТОСУНКУ 58
4.1 Вибір моделі тестування 58

4.2 Опис рівнів тестування 59
4.3 Тестове середовище та методи тестування 73
4.4 Покриття функціональності 74
4.5 Висновки до розділу 4 75
ВИСНОВКИ 77
ПЕРЕЛІК ПОСИЛАНЬ 79
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 82
ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ 89

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface – програмний інтерфейс застосунку

EF Core – Entity Framework Core – ORM-фреймворк для .NET

ORM – об'єктно-реляційне відображення,

ASP.NET Core – кросплатформний фреймворк для створення вебзастосунків

HTTP – HyperText Transfer Protocol – протокол передавання гіпертексту

SQL Server – система управління базами даних від Microsoft

UI – User Interface – інтерфейс користувача

UX – User Experience – досвід користувача

JSON – JavaScript Object Notation – формат обміну даними

CRUD – Create, Read, Update, Delete – основні операції над даними

СУБД – система управління базами даних.

ВСТУП

У сучасному інформаційному суспільстві кількість цифрових джерел літератури постійно зростає, що створює потребу в ефективних інструментах для зберігання, впорядкування та обміну книжковими уподобаннями.

Особливої актуальності набувають системи, що дозволяють не лише керувати власною бібліотекою, а й взаємодіяти з іншими користувачами через рекомендації, рейтинги та обговорення.

Традиційні формати зберігання списків прочитаного поступаються інтерактивним WEB-застосункам із соціальним компонентом, які забезпечують користувачеві більш гнучкий і персоналізований досвід.

В умовах активного використання цифрових інструментів для самоосвіти, самовираження та пошуку однодумців — запропонований застосунок є актуальним і практично значущим.

Мета роботи - цифровізація процесу обміну книжковими вподобаннями між читачами завдяки використанню web-застосунку.

Об’єкт дослідження - процес обміну книжковими вподобаннями.

Предмет дослідження - технології та інструменти розробки web-застосунку для обміну книжковими вподобаннями.

Для реалізації поставленої мети потрібно вирішити наступні завдання:

1. Провести аналіз предметної області цифрових бібліотек та онлайн-сервісів з рекомендаційною і соціальною складовою.
2. Дослідити існуючі рішення, виявити їх особливості та обґрунтувати необхідність створення нового web-застосунку для організації особистих бібліотек.
3. Сформувати функціональні та нефункціональні вимоги до web-застосунку для ведення особистої бібліотеки та системи рекомендацій.
4. Спроєктувати архітектуру клієнт-серверного web-застосунку з

підтримкою розмежування прав доступу.

5. Реалізувати web-застосунок із використанням ASP.NET Core, Razor Pages, Entity Framework Core та SQL Server.

6. Провести тестування функціональності web-застосунку.

Результатом цієї дипломної роботи є функціональний WEB-застосунок, який забезпечує ведення особистої бібліотеки, соціальну взаємодію між користувачами та надає персоналізовані книжкові рекомендації на основі читацької активності інших користувачів.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Сучасні підходи до створення онлайн-бібліотек і читацьких платформ

Сучасні онлайн-бібліотеки та читацькі платформи дедалі частіше орієнтуються не лише на доступ до літератури, а й на побудову соціального простору для обміну читацьким досвідом. Користувачі очікують не просто переглядати інформацію про книги, а мати змогу вести особисту бібліотеку, фіксувати прогрес читання, ділитися думками, залишати оцінки та бачити, що читають інші.

До найбільш популярних сервісів цього типу належать Goodreads, LibraryThing та Knigoholic, які поєднують класичний каталог книг із елементами соціальної мережі: додавання друзів, перегляд їхніх бібліотек, рекомендації на основі схожих вподобань та рейтинги[1][2][3].

Характерними особливостями сучасних рішень є:

- персоналізація книжкової полиці (власні добірки, статуси читання — “читаю”, “прочитано”, “хочу прочитати”);
- соціальна взаємодія (друзі, обговорення);
- відкритість до API для інтеграції нових джерел даних (наприклад, OpenLibrary або Google Books API [4]);

Проте більшість наявних платформ мають надлишкову функціональність, що ускладнює користування для пересічного користувача. Часто відсутня можливість легко налаштувати взаємодію з друзями на рівні перегляду саме їхніх книг чи рекомендацій у реальному часі.

На відміну від подібних платформ, розроблений у цій роботі WEB-застосунок має локалізований та адаптований функціонал, який:

- дозволяє користувачеві додавати книги у власну бібліотеку через OpenLibrary API одним натисканням;

- надає змогу встановлювати статус, оцінку, коментар до книги, зберігаючи це в особистому профілі;
- реалізує систему друзів з можливістю перегляду їхніх бібліотек;
- генерує рекомендації на основі читацької активності друзів;

1.2 Goodreads

Goodreads — це WEB-застосунок для ведення читацьких щоденників і спільної роботи з книжковими базами. Основна його логіка реалізована на стороні серверу з використанням REST API, що обробляє запити на додавання книг до бібліотеки, оновлення статусів, публікацію оцінок і коментарів. Дані зберігаються у централізованій реляційній базі, доступ до якої контролюється виключно внутрішніми механізмами Goodreads. Головна сторінка web-застосунку зображена на рисунок. 1.1.

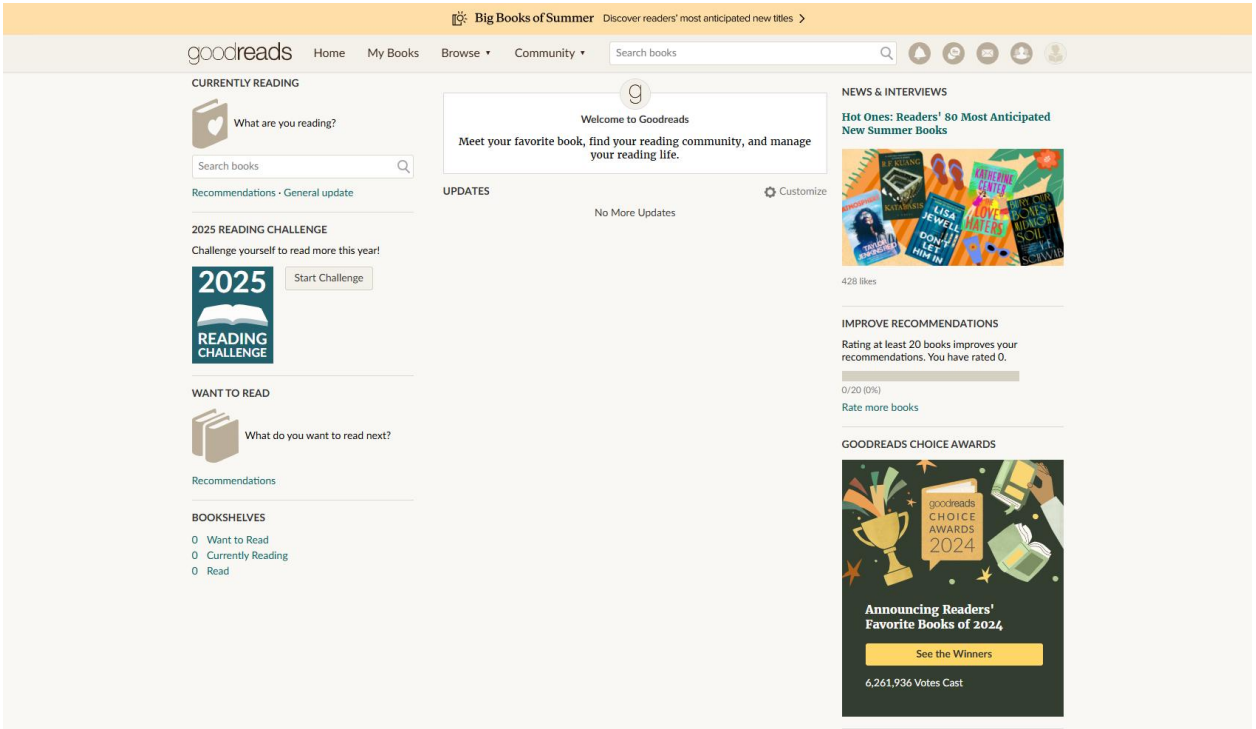


Рис. 1.1 Основне меню Goodreads

Функціональні можливості Goodreads:

- Створення користувацьких полиць (shelves), які є категоріями для сортування книг;
- Фіксація статусів читання (“read”, “currently reading”, “to-read”);
- Можливість виставлення оцінки (1–5 зірок), написання рецензії;
- Обмежена соціальна взаємодія — перегляд бібліотек друзів, участь в групах;
- Рекомендації базуються на оцінках, але не враховують читацьку активність друзів;
- Обмежений доступ до Goodreads API (потрібен ключ, обмеження по кількості запитів, застаріла документація);
- Відсутність гнучкого розширення — неможливо змінити спосіб фільтрації або додати кастомні поля безпосередньо.

Переваги Goodreads:

- Вбудовані алгоритми рекомендацій;
- Можливість імпорту та експорту даних у CSV;
- Велика база метаданих по книгах, синхронізація з Amazon/Kindle.

Недоліки Goodreads:

1. Закритість і обмеженість API:
   - Неможливість здійснювати повноцінний пошук книг з боку клієнта;
   - API дозволяє лише базові запити: пошук за ISBN, автором, назвою — доступ до інформації про друзів, коментарі, полиці — закритий;
2. Недоступність кастомізації та інтеграцій:
   - Немає можливості змінити структуру даних у бібліотеці (неможливо додати додаткові поля, такі як “коментар до статусу”, “рейтинг від друга”);
   - Не підтримується зручна фільтрація книг за додатковими атрибутами (наприклад, лише ті, що додали друзі, або ті, які мають певний жанр і рейтинг).

Відмінності реалізованого WEB-застосунку:

У порівнянні з Goodreads, розроблений застосунок забезпечує гнучкість і відкритість на рівні програмної логіки:

- Використання OpenLibrary API дозволяє отримувати повні метадані про книги з публічної бази без обмежень на комерційне використання. Усі дані використовуються згідно з умовами ліцензії CC0/Public Domain, без порушення авторських прав;
- Прозорість рекомендацій: рекомендації, засновані на конкретній активності друзів, а не на узагальнених рейтингах.
- Розширена система фільтрації дозволяє шукати книги не лише за назвою/автором, а й за рейтингами друзів;
- Гнучке зберігання користувацьких даних: можна додавати коментарі, змінювати статус, оновлювати оцінку без обмежень;
- Соціальний фокус: книги з бібліотек друзів відображаються окремо, реалізовано фільтр на основі спільних вподобань;

1.3 LibraryThing

LibraryThing — це WEB-застосунок для глибокої каталогізації особистих книжкових колекцій із розширеними інструментами сортування та опису кожної книги. Платформа орієнтована передусім на бібліотекарів, колекціонерів та досвідчених користувачів. Основна логіка побудована навколо складної структури метаданих, підтримки професійних стандартів опису (зокрема MARC) та інтеграції з великою кількістю зовнішніх джерел.

Користувацький інтерфейс LibraryThing реалізовано переважно у вигляді таблиць і списків з великою кількістю фільтрів. Уся взаємодія виконується через традиційні багатосторінкові запити, що уповільнює роботу з великою кількістю даних. На рисунку. 1.2 зображений вигляд користувацького інтерфейсу.

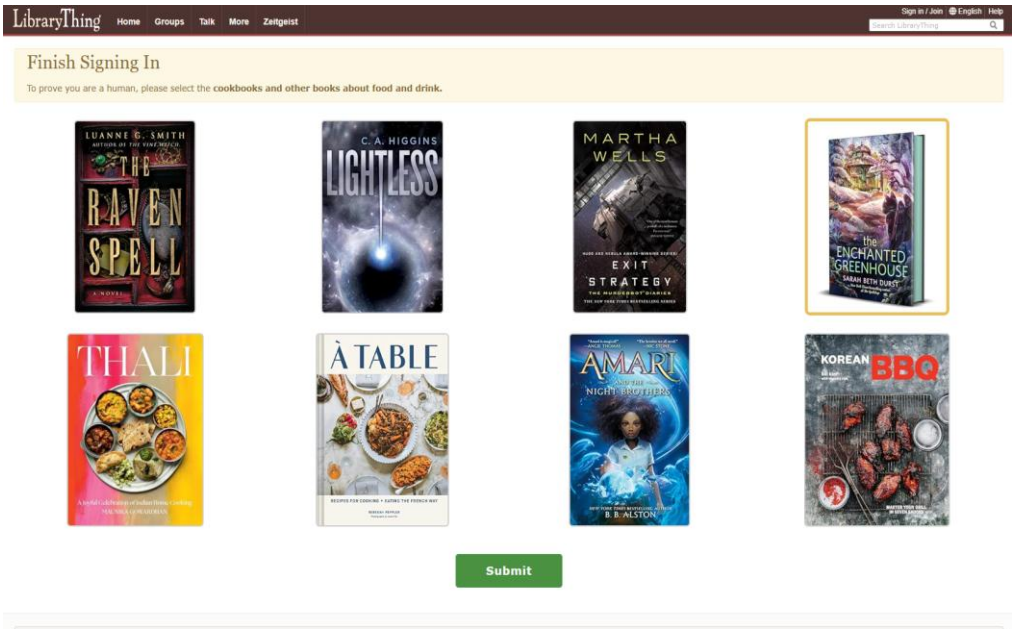


Рис. 1.2 Початковий каталог LibraryThing

Функціональні можливості LibraryThing:

- Додавання книг за ISBN, назвою або вручну;
- Глибока каталогізація книг із підтримкою понад 30 полів, включаючи авторів, переклади, видання, жанри, рік, мову;
- Створення тегів, колекцій, приватних та публічних “полиць”;
- Масове редагування книжкової колекції;
- Експорт даних у форматі MARC, CSV та інші;
- Підключення до зовнішніх джерел (Amazon, WorldCat, Library of Congress).

Переваги LibraryThing:

- Потужна система метаданих: підтримка великої кількості полів, включно з професійними бібліотечними;
- Масове редагування книжкової бази з підтримкою експорту;
- Високий рівень точності при додаванні книг із закритих джерел через бібліотечні протоколи;

- Власний алгоритм рекомендацій на основі тегів, жанрів і даних спільноти.

Недоліки LibraryThing (з технічної точки зору):

1. Обмеження в API та інтеграції:
   - API не підтримує CRUD-операції в повному обсязі — неможливо повноцінно керувати бібліотекою через сторонні застосунки;
   - Відсутність REST-підходу: немає фільтрації, пагінації чи сортування через запити;
2. Складність у використанні для звичайного користувача:
   - Надмірна кількість полів і термінології може відлякувати недосвідчених читачів;
   - Відсутність системи друзів у сучасному розумінні (персональні бібліотеки не взаємодіють напряму);

Відмінності реалізованого WEB-застосунку:

У порівнянні з LibraryThing, розроблений застосунок орієнтований на простоту, соціальну взаємодію та гнучкість інтерфейсу, що робить його більш зручним для повсякденного використання:

- Використання OpenLibrary API дозволяє швидко додавати книги без складного формування MARC-записів, при цьому отримуючи всю необхідну інформацію;
- Фокус на соціальну взаємодію: замість ізольованої бібліотеки користувач бачить, що читають його друзі, які книги вони додали, які оцінили;
- Гнучке редагування статусу, оцінки, коментаря до книги без зайвих полів і складних форм;
- Фільтрація бібліотеки за участю друзів: відображення книг, доданих іншими користувачами, рейтинги, персональні рекомендації на основі читацької активності.

1.4 Knigoholic

Knigoholic — це сучасний україномовний WEB-застосунок для читачів, основним призначенням якого є зберігання та облік особистої бібліотеки, а також публікація рецензій і оцінок до книг. Архітектура платформи побудована на базі традиційної багатосторінкової моделі з формами взаємодії та стрічкою рецензій. Серверна частина працює через REST-запити, дані зберігаються централізовано, з акцентом на бібліографічну простоту.

Knigoholic фокусується на контенті, створеному користувачами — відгуках, рецензіях, рейтингах. При цьому платформа має обмежені можливості для соціальної взаємодії або фільтрації контенту за читацькими колами. API є недокументованим або відсутнім для сторонньої інтеграції.

Функціональні можливості Knigoholic:

- Створення особистої бібліотеки з прив’язкою до облікового запису;
- Додавання книг за назвою, автором або ISBN;
- Можливість залишати оцінки (від 1 до 10) та текстові рецензії;
- Статистика читання та оцінка книг;

Переваги Knigoholic:

- Простий інтерфейс для додавання книг і написання рецензій;
- Можливість фіксувати та переглядати оцінки у відкритому доступі;
- Відсутність зайвих складностей у структурі: усе зосереджено на книгах і відгуках.

Недоліки Knigoholic:

- Відсутня можливість додавати статус читання (“читаю”, “хочу прочитати” тощо);
- Немає перегляду бібліотек друзів або особистих читацьких профілів інших користувачів;

- Не реалізована повноцінна система соціальної взаємодії (немає друзів, груп, фільтрів за активністю інших);
- API для сторонньої інтеграції не надається.

Відмінності реалізованого WEB-застосунку:

У порівнянні з Knigoholic, розроблений у цій дипломній роботі застосунок має розширену логіку персоналізації, з акцентом на соціальні функції та гнучкість у роботі з бібліотекою:

- Додано функцію статусу читання (“читаю”, “прочитано”, “хочу прочитати”) для зручного керування прогресом;
- Кожен користувач має свою бібліотеку, яка доступна іншим — з можливістю фільтрувати книги, додані друзями;
- Реалізовано систему рекомендацій, що ґрунтується на активності друзів (а не глобальному рейтингу);
- Додано підтримку OpenLibrary API, що дозволяє автоматично підвантажувати книги без ручного введення;
- Присутня соціальна взаємодія — користувачі можуть додавати друзів, бачити спільні книги, коментувати та оцінювати.

1.5 Інтеграція з відкритими API (OpenLibrary)

Одним із ключових елементів сучасних WEB-застосунків для роботи з книгами є інтеграція з зовнішніми відкритими джерелами метаданих. Замість ручного введення даних про книги, інтеграція з API дозволяє автоматизувати отримання необхідної інформації — назви, авторів, обкладинок, років видання, жанрів тощо.

У межах цієї дипломної роботи було реалізовано повноцінну інтеграцію з OpenLibrary API — відкритим інтерфейсом, розробленим і підтримуваним Internet Archive, який надає доступ до великого масиву бібліографічних даних у відкритому форматі. OpenLibrary — це некомерційна ініціатива з відкритою ліцензією (CC0/Public

Domain), що дозволяє безкоштовне і легальне використання даних у сторонніх застосунках

Переваги інтеграції з OpenLibrary API:

- Безкоштовне та легальне використання без порушення авторських прав (ліцензія CC0);
- Відсутність обмежень на кількість запитів (на відміну від Goodreads API або Google Books API);
- Простий у використанні REST-інтерфейс із JSON-форматом відповіді;
- Швидкість пошуку та надійність джерела (індексуються мільйони книжкових записів);
- Можливість розширення (OpenLibrary має також API для авторів, суб'єктів, колекцій тощо).

Обмеження, враховані при реалізації:

- Деякі записи містять неповні метадані (наприклад, відсутні обкладинки або описи).
- Немає строгої гарантії щодо якості опису українською мовою — більшість описів англомовні.
- API не дозволяє шукати тільки українські книги, тому відбір потрібен на рівні застосунку.

1.6 Висновки до розділу 1

У результаті проведеного порівняльного аналізу функціональних можливостей таких платформ, як Goodreads, LibraryThing та Knigoholic, виявлено суттєві відмінності у реалізації основних складових, притаманних сучасним веб-сервісам для ведення персональних бібліотек. Незважаючи на те, що всі три сервіси дозволяють працювати з книжковими базами та формувати певну структуру особистої бібліотеки,

характер реалізації окремих компонентів істотно різниться залежно від цільової аудиторії, технічної архітектури та рівня відкритості системи.

З метою систематизації отриманих результатів нижче представлено порівняльну таблицю 1.1, яка дозволяє візуально оцінити функціональні можливості кожного сервісу:

Таблиця 1.1

Порівняння аналогів додатку

	Функціональність	Goodreads	LibraryThing	Knigoholic	Readit
1	Створення особистої бібліотеки	+	+	+	+
2	Додавання статусу читання	+	+	-	+
3	Перегляд бібліотек інших користувачів	+/-	-	-	+
4	Оцінювання та рецензування книг	+	+	+	+
5	Прозорі рекомендації	-	-	-	+

Продовження таблиці 1.1

Порівняння аналогів додатку

	Функціональність	Goodreads	LibraryThing	Knigoholic	Readit
6	Редагування статусу та коментарів до книг	+/-	+	-	+
7	Українська локалізація	+	-	+	+

Аналіз функціональних характеристик трьох популярних платформ — Goodreads, LibraryThing і Knigoholic — показав, що жоден із розглянутих сервісів не забезпечує повного набору можливостей, необхідних для створення сучасної, гнучкої та соціально орієнтованої платформи персональної бібліотеки.

Goodreads має сильну систему рецензій, але обмежений у гнучкості керування даними та прозорості взаємодії. LibraryThing орієнтований на глибоку каталогізацію, проте позбавлений сучасної соціальної складової. Knigoholic, хоч і надає базовий функціонал рецензування, не підтримує статусів читання та взаємодії між користувачами.

Усі зазначені недоліки були враховані при проєктуванні власного застосунку. На відміну від аналогів, розроблений сервіс Readit об’єднує всі ключові компоненти: просте додавання книг із відкритого джерела (OpenLibrary), повний контроль над статусами читання, можливість залишати оцінки й коментарі, систему перегляду бібліотек інших користувачів, а також прозорі соціальні рекомендації, засновані на реальній активності друзів.

2. ПРОЄКТУВАННЯ І АРХІТЕКТУРА WEB-ЗАСТОСУНКУ

2.1 Постановка задачі та технічне завдання

У сучасних умовах цифровізації читання зростає потреба у зручних інструментах для ведення особистих бібліотек із можливістю оцінювання, обговорення та обміну книжковими вподобаннями. Попри наявність низки аналогічних сервісів, жоден із них не забезпечує комплексної підтримки україномовної спільноти користувачів, соціальної інтеграції, прозорих рекомендацій та гнучкої фільтрації бібліотек.

Метою розробки є створення WEB-застосунку для управління особистою бібліотекою користувача з інтеграцією відкритого API (OpenLibrary), підтримкою соціальної взаємодії між користувачами, функцією оцінювання книг, коментуванням, відображенням читацької активності друзів, а також фільтрацією книжкового контенту за статусами, рейтингами та жанрами.

У межах дипломної роботи необхідно реалізувати WEB-застосунок, що має відповідати наступним функціональним і нефункціональним вимогам:

Функціональні вимоги:

- Пошук книг через інтеграцію з OpenLibrary API.
- Додавання книг до особистої бібліотеки користувача.
- Встановлення статусу читання (“читаю”, “прочитано”, “хочу прочитати”).
- Додавання, редагування та видалення особистих оцінок і коментарів до книг.
- Відображення бібліотек інших користувачів (друзів).
- Додавання користувачів у список друзів, перегляд спільних книг.
- Фільтрація бібліотеки за жанром, статусом, рейтингом або читацькою активністю друзів.

- Реєстрація, авторизація, підтримка ролей (звичайний користувач, адміністратор).
- Адаптивна верстка інтерфейсу для ПК та мобільних пристроїв.

Нефункціональні вимоги:

- Стабільна робота застосунку при типовому навантаженні.
- Відповідність стилістичним стандартам сучасного веб-дизайну.
- Захист від несанкціонованого доступу до даних користувача.
- Розширюваність — можливість легко додавати нові функції в майбутньому.
- Відкритість архітектури — можливість повторного використання компонентів.

У межах даного дипломного проєкту був розроблений WEB-застосунок, що дозволяє користувачеві створювати персональну цифрову бібліотеку з можливістю соціальної взаємодії та отримання рекомендацій на основі читацьких вподобань його друзів.

Основна задача полягає в реалізації зручного інструменту для збереження книжок, відстеження статусу прочитання, оцінювання, коментування, формування читацького кола та персоналізованих рекомендацій.

Формалізована постановка задачі передбачає створення системи, що охоплює кілька функціональних блоків:

- бібліотечний блок: додавання книг із відкритого API, встановлення статусів прочитання, ведення власної бібліотеки;
- соціальний блок: створення списку друзів, перегляд їхніх бібліотек, рекомендації на основі їх оцінок;
- аналітичний блок: формування узагальненої інформації про прочитане, оцінки, вподобання;
- модераційно-адміністративний блок: базова підтримка модерації вмісту з боку адміністратора.

Очікується, що користувачі зможуть:

- знаходити книги за назвою або автором;
- додавати їх до особистої бібліотеки;
- вказувати статус прочитання (не прочитано, читаю, прочитано);
- оцінювати книги за п’ятизірковою шкалою;
- залишати коментарі до книг та редагувати їх;
- переглядати, хто з друзів читає ту саму книгу або вже її оцінив;
- взаємодіяти із застосунком українською мовою.

З боку адміністратора передбачено набір функцій: перегляд користувачів, модерування коментарів, керування ролями.

Щодо нефункціональних вимог, застосунок повинен бути:

- доступним із будь-якого сучасного браузера;
- локалізованим українською мовою з можливістю швидкого перемикання мови інтерфейсу;
- захищеним з точки зору зберігання особистих даних користувачів;
- гнучким до розширення.

Передбачено зберігання таких сутностей, як: користувач, бібліотека, зв’язки між користувачами, статуси, оцінки, коментарі. Залежності між цими об’єктами передбачають зв’язки один-до-багатьох та багато-до-багатьох, що вимагає продуманого моделювання структури даних.

2.2 Вибір стеку технологій

Для реалізації WEB-застосунку, що поєднує роботу з даними, динамічний інтерфейс і інтеграцію з зовнішніми сервісами, необхідно обрати технології, які забезпечать надійність, розширюваність та зручність у розробці й подальшій підтримці.

У якості базової платформи було обрано ASP.NET Core — сучасне кросплатформне середовище для створення WEB-додатків, яке відзначається високою продуктивністю, безпекою та широкими можливостями для інтеграції з іншими технологіями Microsodft. Одним із ключових факторів вибору цієї платформи є її підтримка модульності, гнучкої конфігурації, а також активна спільнота та регулярні оновлення.

Інтерфейс користувача реалізовується з використанням Razor Pages — зручного шаблону побудови сторінок, який дозволяє розділити логіку і відображення даних у межах однієї сторінки. На відміну від класичного MVC-підходу, Razor Pages значно спрощує структуру коду, що є особливо корисним для застосунків, де сторінки мають індивідуальну поведінку і взаємодію з базою даних.

Для зберігання даних про користувачів, книги, бібліотеки, зв’язки, статуси, рейтинги та коментарі було обрано Microsoft SQL Server. Ця система управління базами даних добре інтегрується з ASP.NET Core і забезпечує високу надійність, транзакційність та підтримку складних запитів. Для зручної роботи з даними на рівні коду використовується ORM Entity Framework Core [6], яка дозволяє взаємодіяти з базою даних на рівні об’єктів, що значно спрощує розробку та підтримку проєкту.

Клієнтська частина доповнюється використанням Bootstrap 5 — фреймворку, що забезпечує адаптивність інтерфейсу і дозволяє швидко створювати зручні елементи керування, зокрема картки книг, кнопки дій, навігацію тощо. Для інтерактивної взаємодії на сторінках (додавання до бібліотеки без перезавантаження, динамічна фільтрація, зміна статусів) застосовуються JavaScript та jQuery.

Оскільки джерелом книжкових даних є відкрите API сервісу OpenLibrary ,проєкт передбачає реалізацію механізмів взаємодії із зовнішніми JSON-інтерфейсами — отримання інформації про книги, авторів, обкладинки, ISBN тощо.

Таким чином, обраний стек технологій дозволяє створити ефективний, розширюваний і безпечний WEB-застосунок, який задовольняє функціональні вимоги проєкту та відповідає сучасним практикам розробки WEB-програмного забезпечення.

У межах побудови застосунку було обрано шаблон багаторівневої архітектури (Layered Architecture), що дозволяє розділити логіку відображення, бізнес-логіку, керування доступом та зовнішні залежності. На рисунку. 2.1 представлено діаграму компонентів, яка відображає основні рівні та їхню взаємодію.

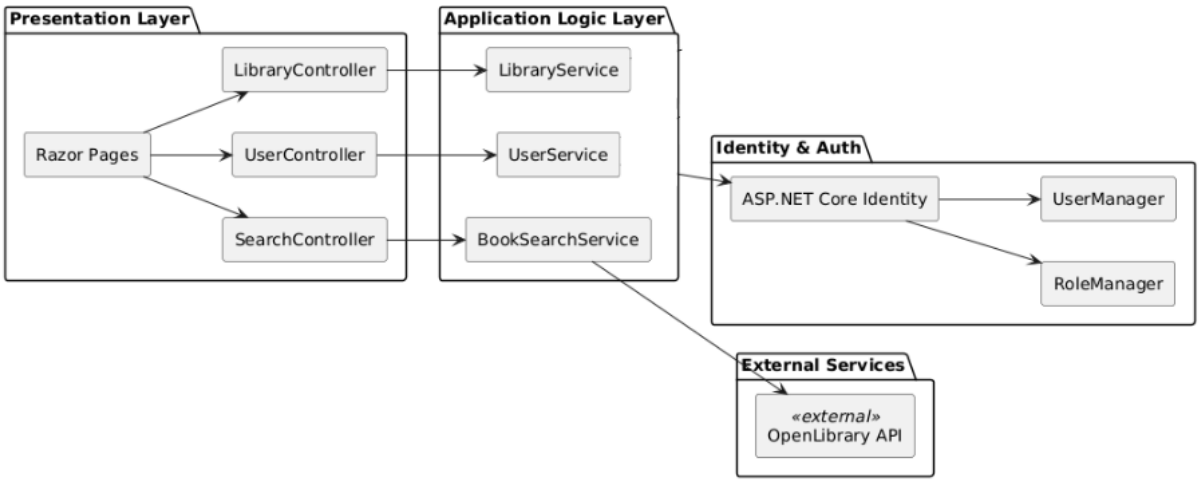


Рис. 2.1 Діаграма компонентів WEB-застосунку Readit
відповідно до багаторівневої архітектури.

2.2.1 Вибір мови програмування

У рамках цієї роботи для реалізації WEB-застосунку було обрано мову C#, яка є частиною екосистеми .NET і вже понад два десятиліття активно використовується в розробці корпоративного, серверного й веб-програмного забезпечення.

C# є строго типізованою, об'єктно-орієнтованою мовою з підтримкою сучасних парадигм програмування. Її синтаксис зрозумілий та логічно структурований, а завдяки тісній інтеграції з платформою .NET, C# дозволяє використовувати потужні вбудовані бібліотеки, що значно спрощує вирішення широкого спектру прикладних задач.

Однією з головних причин вибору C# для розробки Readit є її підтримка об’єктно-орієнтованого програмування (ООП) — парадигми, яка дозволяє

проектувати програму у вигляді набору об’єктів, що відображають реальні сутності системи: користувач, бібліотека, коментар. Завдяки таким механізмам, як інкапсуляція, спадкування і поліморфізм, розробник може побудувати зручну, гнучку і підтримувану модель предметної області.

Водночас C# підтримує асинхронне програмування, що є критично важливим для будь-якого сучасного WEB-застосунку. При взаємодії з базами даних або зовнішніми API (OpenLibrary) виконання запитів має бути неблокувальним, щоб не зупиняти роботу всієї системи або інтерфейсу користувача. Використання ключових слів async та await в C# дозволяє легко створювати асинхронні методи, що зменшує затримки у роботі застосунку і підвищує його загальну продуктивність.

Ще однією перевагою є вбудована мова запитів LINQ (Language Integrated Query). Вона дозволяє писати ефективні запити до колекцій та баз даних у вигляді зрозумілого та компактного C#-синтаксису. Замість написання довгих SQL-запитів розробник може працювати із сутностями як з об’єктами, виконуючи фільтрацію, сортування, групування — що ідеально підходить для роботи з даними про книги, оцінки, статуси та зв’язки між користувачами.

C# також забезпечує високий рівень безпеки роботи з пам’яттю завдяки автоматичному збиранню сміття (Garbage Collection). Це дозволяє уникати витоків пам’яті та зменшити ризик помилок, пов’язаних з ручним керуванням ресурсами, як це властиво деяким низькорівневим мовам програмування.

Ще одним важливим фактором є кросплатформність, досягнута завдяки .NET Core (а нині .NET 7/8). Завдяки цьому C# більше не обмежується Windows-середовищем: застосунки можуть запускатися й на Linux чи macOS. Це особливо важливо в контексті хостингу й розгортання застосунку на сервері з будь-якою ОС або в хмарних платформах.

Мова також вирізняється гарною масштабованістю, що дозволяє створювати проєкти як для невеликої кількості користувачів, так і для десятків тисяч відвідувачів без необхідності повністю змінювати архітектуру системи. Завдяки розвиненій

інфраструктурі, зокрема механізмам логування, обробки помилок, конфігурації, підтримці dependency injection та модулярності, C# у поєднанні з .NET забезпечує стабільне середовище для реалізації як простих CRUD-додатків, так і складних клієнт-серверних рішень.

Загалом, C# поєднує у собі сучасний синтаксис, безпечну архітектуру, високу продуктивність, активну підтримку спільноти та документацію, що робить її ідеальним вибором для створення WEB-застосунку типу Readit. У рамках цього проєкту мова C# дозволяє повністю реалізувати функціональність, пов’язану з обробкою запитів користувача, управлінням базами даних, логікою авторизації, обчисленням рекомендацій, а також з реалізацією адміністративного функціоналу.

2.2.2 Вибір фреймворку

Для реалізації серверної частини WEB-застосунку було обрано фреймворк ASP.NET Core — сучасну, відкриту, кросплатформну платформу для створення високопродуктивних веб-додатків. ASP.NET Core [5] активно розвивається корпорацією Microsoft і є наступником класичного ASP.NET, проте має повністю переписану архітектуру, орієнтовану на продуктивність, модульність і гнучкість.

Однією з основних переваг ASP.NET Core є його легкість і масштабованість. У класичних фреймворках часто наявні вбудовані модулі, які завжди підвантажуються разом із проєктом, навіть якщо не використовуються. Натомість ASP.NET Core дозволяє підключати лише ті компоненти, які справді потрібні, що позитивно впливає на швидкість завантаження, розмір застосунку та ефективність його роботи.

Також фреймворк спроєктовано так, щоб бути повністю кросплатформним, що означає можливість запуску WEB-застосунку не лише на Windows, але й на Linux та macOS. Це відкриває широкі можливості для розгортання — як на віртуальних приватних серверах, так і в хмарних середовищах (Microsoft Azure, AWS, DigitalOcean тощо).

У даному дипломному проєкті використовується не класичний шаблон ASP.NET MVC, а більш сучасний варіант — Razor Pages. Цей підхід передбачає розробку окремих сторінок, кожна з яких має пов'язану модель (PageModel), що обробляє запити до цієї сторінки. Таким чином, логіка і подання тісно пов’язані, але чітко розділені в коді. Razor Pages особливо зручні у проєктах середнього масштабу, таких як Readit, де більшість сторінок мають власну логіку взаємодії з моделями: додавання книги, оновлення статусу, оцінка, коментар тощо.

Завдяки Razor Pages структуру коду легко підтримувати, тестувати й розширювати — що робить цей шаблон ідеальним для навчальних, дипломних і реальних продуктів. Крім того, Razor Pages дозволяє вбудовувати C#-код безпосередньо у розмітку, що пришвидшує реалізацію інтерфейсної логіки без потреби створювати окремі контролери для кожної дії.

ASP.NET Core пропонує інтегровані засоби безпеки, включно з підтримкою аутентифікації, авторизації, захисту від міжсайтової підміни запитів (CSRF) та шифрування конфіденційних даних. Це особливо важливо в застосунку, який оперує персональними даними користувачів, зокрема обліковими записами, аватарами, історією читання, коментарями.

Фреймворк має вбудовану систему залежностей (Dependency Injection[19]), що дозволяє зручно впроваджувати сервіси у класи й сторінки. Це дає змогу уникати жорстких зв’язків між компонентами, а також полегшує модульне тестування, заміну окремих частин логіки або їхню подальшу модифікацію.

Крім того, ASP.NET Core має гнучку систему конфігурації, яка дозволяє змінювати налаштування проєкту без перекомпіляції, а також вбудовану підтримку журналювання, обробки помилок, локалізації, фільтрації запитів тощо. Усе це робить його універсальним інструментом для створення сучасних, масштабованих і безпечних веб-додатків.

У проєкті Readit ASP.NET Core використовується як центральна платформа, яка поєднує роботу з даними, взаємодію з користувачем через Razor Pages, реалізацію

логіки авторизації та ролей, а також обробку запитів до зовнішнього API. Враховуючи потребу у швидкості розробки, структурованості, безпеці й масштабованості, обраний фреймворк повністю відповідає технічним і функціональним вимогам дипломного проєкту.

2.2.3 Вибір середовища розробки

У процесі реалізації програмного забезпечення надзвичайно важливим є правильний вибір середовища розробки, адже саме воно забезпечує комфортну роботу з кодом, ефективне управління проєктом, швидкий пошук помилок та інтеграцію з системами контролю версій, базами даних і бібліотеками. У рамках даного дипломного проєкту для розробки WEB-застосунку було обрано середовище JetBrains Rider.

JetBrains Rider — це кросплатформне IDE (інтегроване середовище розробки), створене компанією JetBrains, яке поєднує в собі переваги .NET-підтримки від ReSharper та швидкодію редакторів IntelliJ [7]. На відміну від традиційного середовища Visual Studio, яке є основним інструментом для розробки .NET-проєктів, Rider працює стабільно не лише на Windows, а й на Linux та macOS, що робить його універсальним вибором для розробників із різних операційних систем.

Основні переваги Rider, що стали вирішальними у виборі:

1. Кросплатформність і стабільність
Rider без проблем працює на будь-якій популярній операційній системі, що дає можливість вільно змінювати середовище розгортання, тестування або розробки. Це важливо в контексті хостингу проєкту на серверах з Linux-архітектурою або використання альтернативних середовищ (наприклад, хмарних IDE або Docker-контейнерів).

2. Потужний аналіз коду та навігація
Середовище має вбудовану підтримку ReSharper — одного з найкращих інструментів для аналізу коду. Це дозволяє знаходити помилки, неефективні конструкції,

дублювання, порушення принципів SOLID чи потенційні проблеми ще до виконання проєкту. Завдяки цьому розробка відбувається швидше, код стає якіснішим, а помилок — менше.

3. Інтеграція з Git та системами керування версіями

У Rider реалізована зручна та гнучка інтеграція з Git, що дає змогу керувати гілками, відстежувати зміни, виконувати злиття, переглядати історію комітів безпосередньо в межах IDE. У контексті дипломного проєкту ця можливість використовувалась для ведення окремих гілок функціональності (наприклад, features/book-rating, features/user-profile), які згодом об’єднувались у основну гілку.

4. Зручна підтримка Razor Pages, HTML, CSS, JS, SQL

Rider має повноцінну підтримку Razor-синтаксису, підсвітку коду, автодоповнення для C#, JavaScript, HTML і CSS, а також зручні засоби для роботи з SQL-запитами та відображення структури бази даних. Для проєкту Readit, де велика частина логіки реалізована у Razor Pages, це дозволяло швидко переходити від представлення до моделі, легко працювати з формами, валідацією та асинхронними запитами.

5. Висока продуктивність

Попри великий набір інструментів, Rider працює швидко й стабільно навіть на середніх за потужністю машинах. Це досягається завдяки ефективному рушію IntelliJ, оптимізованій індексації та багатопотоковій обробці. Порівняно з класичною Visual Studio, яка може сповільнюватися на великих проєктах або під час роботи з великою кількістю NuGet-пакетів, Rider забезпечує комфортну роботу навіть у проєктах зі складною структурою.

6. Інтеграція з NuGet і міграціями EF Core

Вбудований менеджер пакетів NuGet у Rider дозволяє швидко підключати необхідні бібліотеки — такі як Microsoft.EntityFrameworkCore, AutoMapper, Microsoft.AspNetCore.Identity тощо. Окрім того, міграції бази даних, створені через Entity Framework Core, легко керуються з вбудованого терміналу, а результати відображаються в панелі SQL Console з можливістю перегляду SQL-коду.

У проєкті Readit середовище JetBrains Rider забезпечило повноцінне охоплення всіх аспектів розробки: від структурування моделі даних до валідації форм, інтеграції з OpenLibrary API, налаштування автентифікації й організації адаптивного інтерфейсу. Завдяки візуалізації структури проєкту, інструментам профілювання та інтелектуальному автодоповненню, розробка здійснювалася швидко, зручно та з мінімальними затримками при перевірці та тестуванні функціоналу.

Особливу увагу в процесі розробки було приділено керуванню версіями за допомогою Git, інтегрованого безпосередньо в середовище Rider. Структура репозиторію відповідала стандартній практиці сучасної розробки: основна гілка main, гілка для активної розробки develop, а також окремі функціональні гілки (features/...) для реалізації нових модулів, таких як система оцінювання книг, бібліотека друзів, профіль користувача тощо. Такий підхід дозволив ізолювати зміни, зручно проводити тестування, уникати конфліктів при злитті та підтримувати чисту історію проєкту. Це дало змогу ефективно організувати розробку навіть у рамках індивідуального проєкту, із можливістю подальшого масштабування та командної співпраці.

2.2.4 Вибір технологій для підтримки бази даних

У якості основної системи управління базами даних для реалізації дипломного проєкту було обрано Microsoft SQL Server — одну з найпотужніших, найстабільніших і водночас добре підтримуваних реляційних СУБД на сучасному ринку. Її інтеграція з середовищем .NET, висока продуктивність, гнучкість у проєктуванні схеми та зручні інструменти адміністрування роблять цей вибір не лише обґрунтованим, а й стратегічно вдалим.

Перш за все, варто відзначити підтримку складної системи зв’язків між таблицями, яка необхідна для моделювання предметної області проєкту. У застосунку Readit реалізовано численні взаємозв’язки між сутностями: користувачами, книгами, бібліотеками, оцінками, статусами читання, а також списками друзів. Усі ці зв’язки потребують чіткої реалізації реляцій: від "один-до-багатьох" (наприклад, один

користувач — багато коментарів) до "багато-до-багатьох" (наприклад, користувачі-друзі або користувачі та книги в бібліотеці). Microsoft SQL Server забезпечує гнучке та безпечне збереження таких структур без втрати цілісності даних.

Ще однією важливою перевагою, що активно використовується в проєкті, є підтримка транзакційності. Це дозволяє об’єднати кілька дій у межах однієї логічної операції, забезпечуючи цілісність бази навіть у разі збоїв. Наприклад, коли користувач додає книгу до бібліотеки, одночасно оцінює її та залишає коментар, усі ці операції мають бути або успішно завершені разом, або повністю відмінені, якщо одна з них зазнає помилки. Завдяки механізму транзакцій у SQL Server така поведінка реалізується просто і ефективно, відповідаючи вимогам до надійності веб-системи.

У процесі розробки також була помітна висока продуктивність запитів до бази даних, що досягається завдяки добре реалізованому механізму індексації. Це особливо важливо в задачах пошуку: при відображенні результатів за автором, назвою книги, статусом читання тощо. Завдяки правильно налаштованим індексам SQL Server забезпечує швидку вибірку навіть при значних обсягах даних. У контексті масштабування проєкту в майбутньому це відкриває перспективи переходу до більш розгалужених фільтрів і аналітичних звітів.

Крім функціональних переваг, сумісність SQL Server із середовищем .NET є однією з ключових причин його використання. Завдяки прямій інтеграції з Entity Framework Core розробники можуть працювати з базою даних через об’єктно-орієнтовану модель, уникаючи написання складних SQL-запитів [8]. Це особливо актуально в дипломному проєкті, де швидкість реалізації, підтримуваність та модульність коду є критичними. Робота з даними за допомогою C#-моделей у EF Core дає змогу описати всю логіку збереження, фільтрації та обробки інформації про книги, користувачів і їхню активність у звичному для розробника вигляді.

Не менш важливою є і наявність потужних інструментів адміністрування, таких як SQL Server Management Studio (SSMS). Під час розробки проєкту SSMS використовувалося для візуалізації структури бази, виконання прямих запитів,

моніторингу виконання команд та резервного копіювання. Зокрема, в процесі тестування функціоналу додавання книг та коментарів SSMS дозволяло оперативно переглядати змінені таблиці, виявляти проблеми зі зв’язками або помилки в конфігурації моделі. Така прозорість та контроль над базою даних суттєво підвищують якість продукту.

За даними аналітичних звітів Microsoft та незалежних тестів, SQL Server є однією з найкращих СУБД для створення веб-додатків із високим навантаженням і складною структурою даних. Його гнучкість, функціональність і стабільність дозволяють з упевненістю рекомендувати його як на етапі розробки, так і в продакшн-середовищі.

Таким чином, застосування Microsoft SQL Server у поєднанні з Entity Framework Core дозволяє досягти ефективної реалізації всієї серверної частини системи з гарантією масштабованості, швидкодії та структурованості збереження даних, що відповідає найкращим практикам сучасної інженерії програмного забезпечення.

У порівнянні із іншими СУБД я обрала SQL Server за характеристиками, які зображені у Таблиці 2.1

Таблиця 2.1

Популярні СУБД для WEB-застосунків

Критерій	SQL Server	PostgreSQL	MySQL	SQLite
Тип	Реляційна	Реляційна	Реляційна	Реляційна, файлова
Підтримка транзакцій	Так	Так	Так	Частково
Робота з EF Core	Повна	Повна	Часткова*	Так, з обмеженнями

Продовження таблиці 2.1

Популярні СУБД для WEB-застосунків

Критерій	SQL Server	PostgreSQL	MySQL	SQLite
Підтримка зв’язків	Повна	Повна	Повна	Обмежена
Масштабованість	Висока	Висока	Висока	Низька
Інтеграція з .NET	Найкраща	Добра	Добра	Мінімальна
Засоби адміністрування	SSMS	pgAdmin	phpMyAdmin	Відсутні
Ліцензія	Комерційна (безкоштовна версія Express)	Відкрита	Відкрита	Відкрита
Підтримка Windows/Linux/Mac	Так	Так	Так	Так

2.2.5 Вибір ORM-фреймворку

Entity Framework Core (EF Core) — це об’єктно-реляційний фреймворк (ORM), який дозволяє розробникам працювати з базою даних через класи C# і LINQ-запити, не використовуючи сирий SQL.

Основні переваги EF Core:

- Модель Code First: структура таблиць автоматично генерується на основі класів моделей, що значно пришвидшує розробку;
- Міграції: дозволяють вносити зміни до схеми бази даних без втрати даних, відстежувати історію змін;
- Вбудована підтримка зв’язків між об’єктами: один-до-одного, один-до-багатьох, багато-до-багатьох;
- LINQ-запити: надають просту, безпечну і гнучку альтернативу SQL-запитам;
- Використання Data Annotations та Fluent API: для точного контролю над структурою БД і поведінкою зв’язків.

У проєкті Readit EF Core використовується для:

- збереження інформації про книги, статуси, користувачів;
- зв’язку бібліотеки з користувачем;
- вибірки рекомендацій на основі оцінок друзів;
- обробки коментарів, рейтингів та аватарок.

2.3 Моделювання структури даних

Однією з фундаментальних складових архітектури WEB-застосунку є правильне моделювання структури даних, що відображає предметну область і забезпечує логічну та ефективну взаємодію між сутностями. У рамках розробки проєкту Readit — соціального WEB-застосунку для управління особистою бібліотекою та взаємодії між читачами — було сформовано набір ключових моделей, які повністю охоплюють функціональні вимоги системи.

2.3.1 Модель користувача (User)

У системі Readit кожен користувач є зареєстрованим учасником із персональним профілем. Базова модель User успадковує властивості від стандартної ASP.NET Core Identity (IdentityUser), що дозволяє зберігати облікові дані (логін, email, пароль) та одночасно розширювати її додатковими полями, специфічними для проєкту.

Серед розширених властивостей моделі користувача реалізовано:

- AvatarFileName — для зберігання шляху до аватара;
- FirstName та LastName — для персоналізації профілю;
- Friendships — список друзів (двосторонній зв’язок між користувачами);
- UserBooks — колекція книг, які додані цим користувачем у свою бібліотеку;
- Comments — коментарі, залишені користувачем до книг.

Таким чином, модель User реалізує кілька одночасних зв’язків: один-до-багатьох з оцінками та коментарями, а також багато-до-багатьох через таблицю зв’язку з іншими користувачами у системі дружби.

2.3.2 Модель бібліотеки (UserBook)

Модель UserBook реалізує таблицю зв’язку багато-до-багатьох між користувачами та книгами, а також містить додаткову інформацію, що стосується індивідуального досвіду читача.

Ця модель містить такі поля:

- UserId, BookId — зовнішні ключі;
- ReadingStatus — статус читання (не прочитано / читаю / прочитано);
- Rating — оцінка за п’ятизірковою шкалою;
- Comment — текстовий коментар;
- CreatedAt — дата додавання книги;
- UpdatedAt — останнє оновлення статусу або оцінки.

Таким чином, бібліотека — це не просто список доданих книг, а повноцінна модель персоналізованої взаємодії користувача з кожною книгою, процес додавання книги зображено на рисунку 2.2.

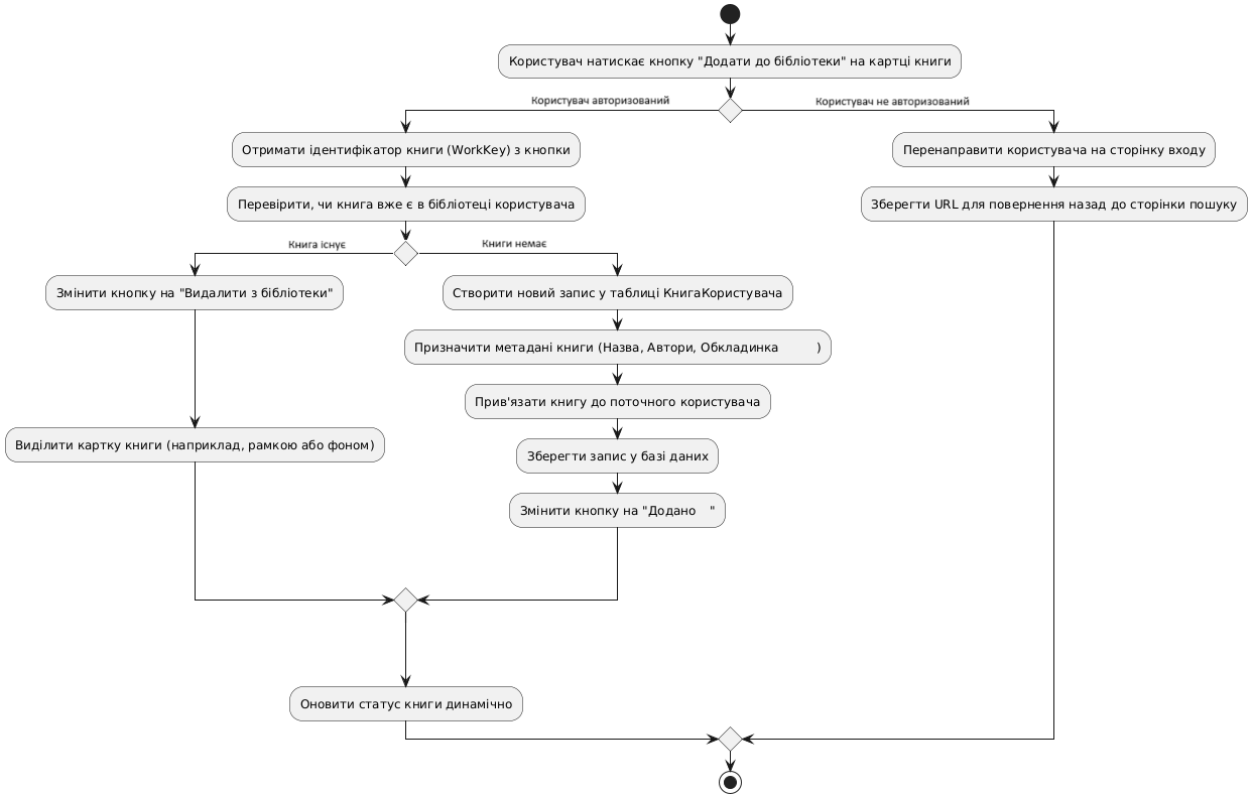


Рис. 2.2 Діаграма діяльності: додавання книги до бібліотеки користувача.

2.3.3 Модель дружби (Friendship)

Соціальна взаємодія — одна з головних особливостей Readit, тому було реалізовано окрему модель Friendship, яка зберігає асиметричний двосторонній зв’язок між користувачами (аналогічно до системи підписок у соціальних мережах).

Модель містить:

- UserId — ініціатор підписки;
- FriendId — цільовий користувач;
- CreatedAt — дата встановлення зв’язку.

Ця структура дозволяє легко визначати: хто є підписником, хто є другом, хто ще не доданий у відповідь, а також реалізувати фільтрацію рекомендацій книг на основі дій друзів, що є ключовою особливістю застосунку.

Для наочності зв'язків між сутностями, що використовуються в системі Readit, представлена діаграма класів рисунок. 2.3, яка відображає структуру моделей, сервісів і перерахувань, а також їхню взаємодію.

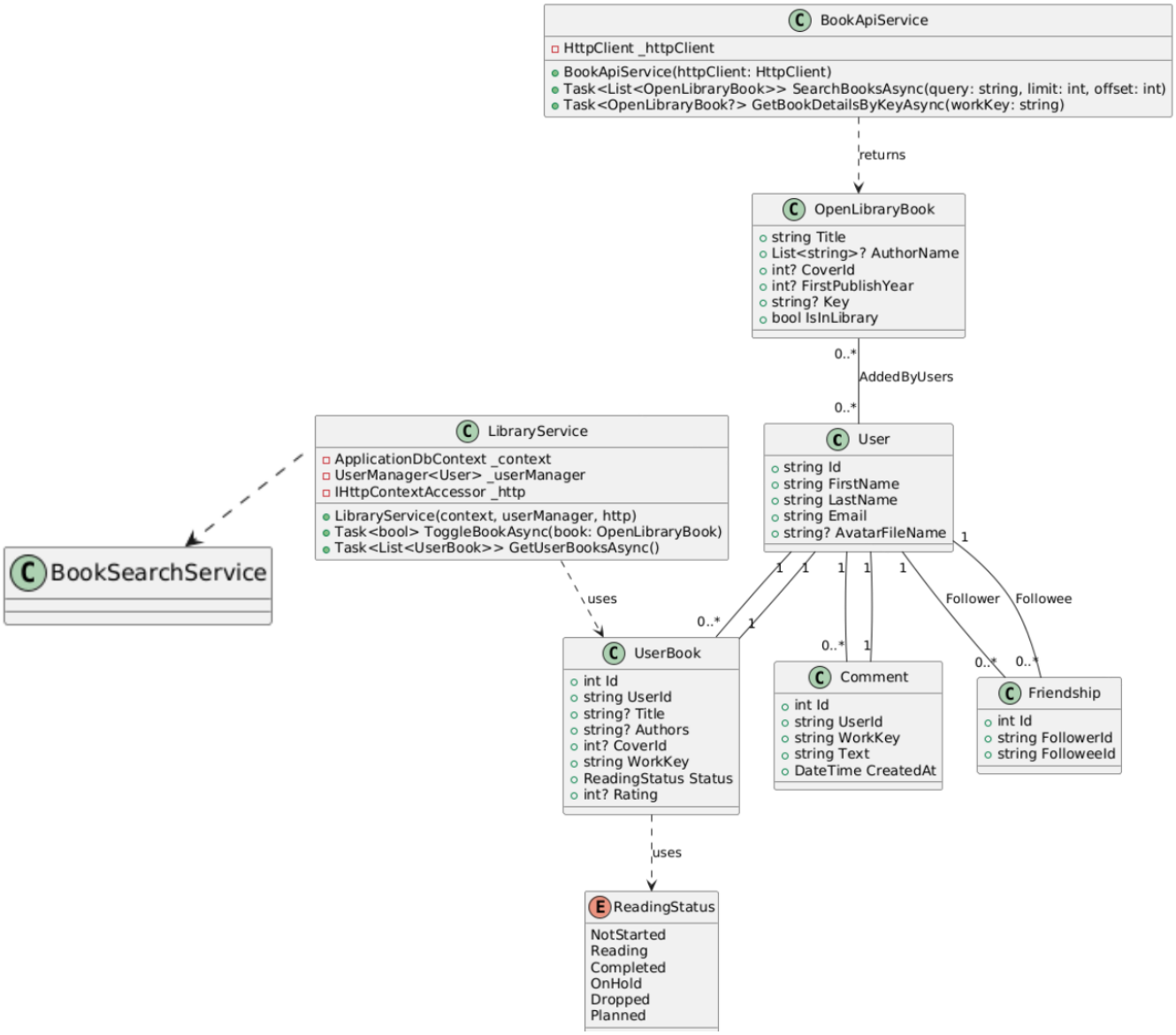


Рис. 2.3 Діаграма класів моделей, сервісів і зв’язків

3. РЕАЛІЗАЦІЯ ФУНКЦІОНАЛУ ВЕБ-СИСТЕМИ

3.1 Функціонал додавання книг через OpenLibrary API

OpenLibrary API у моєму проєкті виступає основним джерелом книжкових даних— саме завдяки йому я змогла уникнути потреби створювати та наповнювати власну базу даних вручну, що значно зменшило час реалізації Readit. Цей API надає доступ до глобального каталогу книжкових метаданих, зібраного в межах проєкту Internet Archive, що є одним з найбільших відкритих сховищ інформації у світі. Його головна перевага полягає в тому, що дані структуровані, безкоштовні для використання та доступні без складної авторизації — саме це дозволило інтегрувати OpenLibrary у мій застосунок без надмірних бар'єрів.

3.1.1 Алгоритм взаємодії з API

На серверній стороні застосунку створено сервіс BookApiService, який відповідає за виконання HTTP-запитів до ендпоінту https://openlibrary.org/search.json. Клас використовує HttpClient, який інжектується через конструктор за допомогою DI-контейнера ASP.NET Core[5].

Під час запиту:

- Параметр q формується на основі введеного користувачем запиту (Query), наприклад ?q=orwell+1984;
- Після отримання відповіді виконується десеріалізація JSON-об'єкта у C#-модель OpenLibrarySearchResult, яка містить список книг (Docs), кожна з яких представлена як OpenLibraryBook [10].

Модель OpenLibraryBook спеціально адаптована під структуру OpenLibrary: вона включає:

- Key (унікальний ідентифікатор книги)

- Title(заголовок)
- AuthorName (список авторів)
- CoverId(Ідентифікатор обкладинки)
- FirstPublishYear(Дату першої публікації книги)

Для уникнення надлишкового зберігання, ці книги не додаються у локальну базу, поки користувач самостійно не натисне кнопку "Додати до бібліотеки".

Для кращого розуміння логіки обробки пошукового запиту та збереження книги в особисту бібліотеку користувача, на рисунку 3.1 наведено структуровану схему взаємодії компонентів у процесі пошуку та додавання книги через OpenLibrary API. Діаграма охоплює всі основні блоки, починаючи з дій користувача у браузері, запитів до зовнішнього API, обробки на стороні Razor Pages, і завершуючи взаємодією з LibraryService, який відповідає за створення запису UserBook у базі даних.

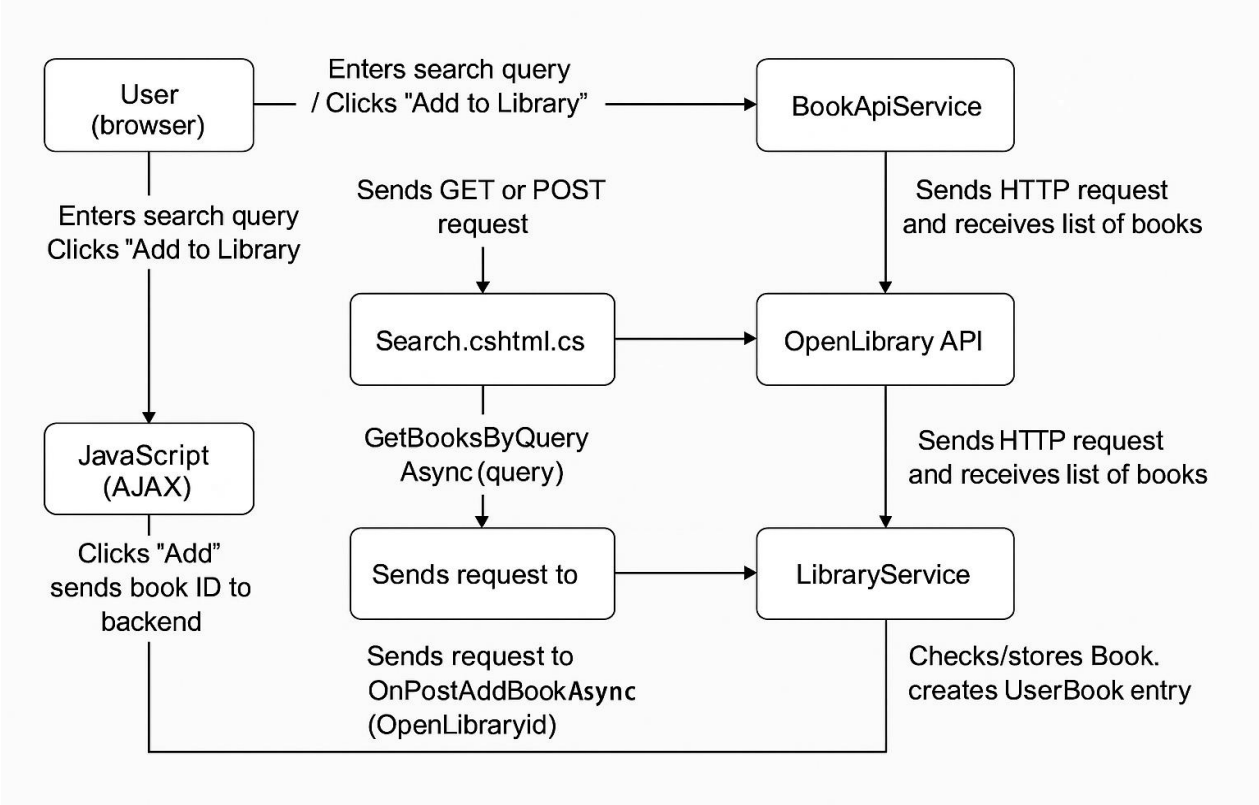


Рис. 3.1 Схема взаємодії компонентів на сторінці Search через OpenLibrary API

У цій схемі показано поділ між синхронною взаємодією (наприклад, введення пошукового запиту) та асинхронною — як-от обробка кнопки "Додати до бібліотеки" через JavaScript, без перезавантаження сторінки. Такий підхід дозволяє забезпечити реактивну поведінку інтерфейсу і зберігати позитивний користувацький досвід навіть при великій кількості одночасних запитів до OpenLibrary.

3.1.2 Додавання книги в бібліотеку

На Razor-сторінці /Search відображається список результатів пошуку у вигляді карток. Кожна картка містить:

- Назву книги,
- Автора,
- Обкладинку (завантажується за допомогою CoverId через covers.openlibrary.org)
- Кнопку "Додати"(реалізована у вигляді AJAX-кнопки з асинхронним запитом[11])

Коли користувач натискає кнопку додавання то відправляється запит на метод AddBookToLibraryAsync сервісу LibraryService. Потім цей метод спершу перевіряє, чи існує книга з таким OpenLibraryId у таблиці Books і якщо ні — створюється новий запис у таблиці Books із базовими метаданими. Далі створюється зв’язок у таблиці UserBooks, що включає UserId, BookId, початковий статус читання (Unread) та дату додавання. У випадку повторного додавання виконується перевірка на дублікат, що запобігає створенню зайвих записів.

3.1.3 Унікальні особливості реалізації

- Фронтенд-реакція не виконує повного перезавантаження сторінки: після натискання кнопки Add, через JavaScript вона замінюється на "У бібліотеці", не змінюючи DOM-контексту.

- Система унікального ключа використовує саме OpenLibraryId, а не внутрішній ID бази даних, що дозволяє зберігати лише ті книги, які реально були обрані користувачем.
- Кешування обкладинок реалізується за допомогою зберігання лише ID обкладинки (CoverId) замість завантаження зображення у базу даних, що значно зменшує розмір застосунку.

3.2 Збереження персональної бібліотеки

У Readit персональна бібліотека реалізована не як простий список книг, а як окрема, спеціалізована сутність, що забезпечує збереження індивідуального досвіду кожного користувача. Основою реалізації цієї функціональності є модель UserBook, яка поєднує конкретного користувача з конкретною книгою, надаючи змогу відокремлено зберігати усі користувацькі дії, пов’язані з цією книгою.

Ця архітектура була додана для уникнути дублювання записів про книги в базі, оскільки кожна книга зберігається один раз у таблиці Books, а вся інформація, яка залежить від конкретного користувача [12](додавання книги до бібліотеки), фіксується в UserBooks .Під час додавання книги в бібліотеку система спочатку перевіряє, чи існує ця книга в глобальному каталозі Books. Якщо ні — вона створюється на основі відповіді з OpenLibrary API. Якщо книга вже є, система лише створює запис у UserBooks, який містить ID користувача, ID книги та службові поля для подальших дій (рейтинг, статус, коментар — хоча вони обробляються окремо, саме тут зберігаються їх значення).

Механізм працює наступним чином: при натисканні на кнопку «Додати до бібліотеки», з фронтенду (через JavaScript/AJAX) надсилається запит до Razor Page-обробника OnPostAddBookAsync(). Там і відбувається вся перевірка: чи існує книга, чи є вона вже у бібліотеці поточного користувача. Якщо все пройдено успішно — створюється новий запис UserBook.

Ключова перевага такого підходу — гнучкість. Кожен користувач має свою бібліотеку, навіть якщо книги в ній ті самі, що й у друзів. Це дає змогу надалі розширювати функціонал.

Технічно UserBook — це окрема таблиця з зовнішніми ключами на Users та Books, з індексацією по комбінації UserId + BookId. Це дозволяє швидко знаходити книги певного користувача, перевіряти наявність книги в бібліотеці та не допускати дублювання записів. Усі зміни відбуваються через LibraryService, який виконує централізовану логіку створення і взаємодії з цими записами. Таким чином, Razor Pages не містять дублів бізнес-логіки.

Архітектура з UserBook як проміжною таблицею створює чітку розділеність між загальною інформацією про книгу і персональними діями, що робить систему масштабованою та підтримуваною. У майбутньому можна реалізувати експортування бібліотеки, синхронізацію з іншими сервісами або фільтрацію бібліотеки за будь-яким параметром — усе це базується на вже наявній структурі.

Саме форма на рисунку 3.2 є кінцевим результатом усієї взаємодії між класами User, Book та UserBook. На цьому зображенні користувач бачить список книг, які він особисто додав до своєї бібліотеки. Кожна картка книги — це візуалізація конкретного запису з таблиці UserBook, що містить загальну та персоналізовану інформацію: статус, рейтинг а також технічно — зв’язок із його обліковим записом.

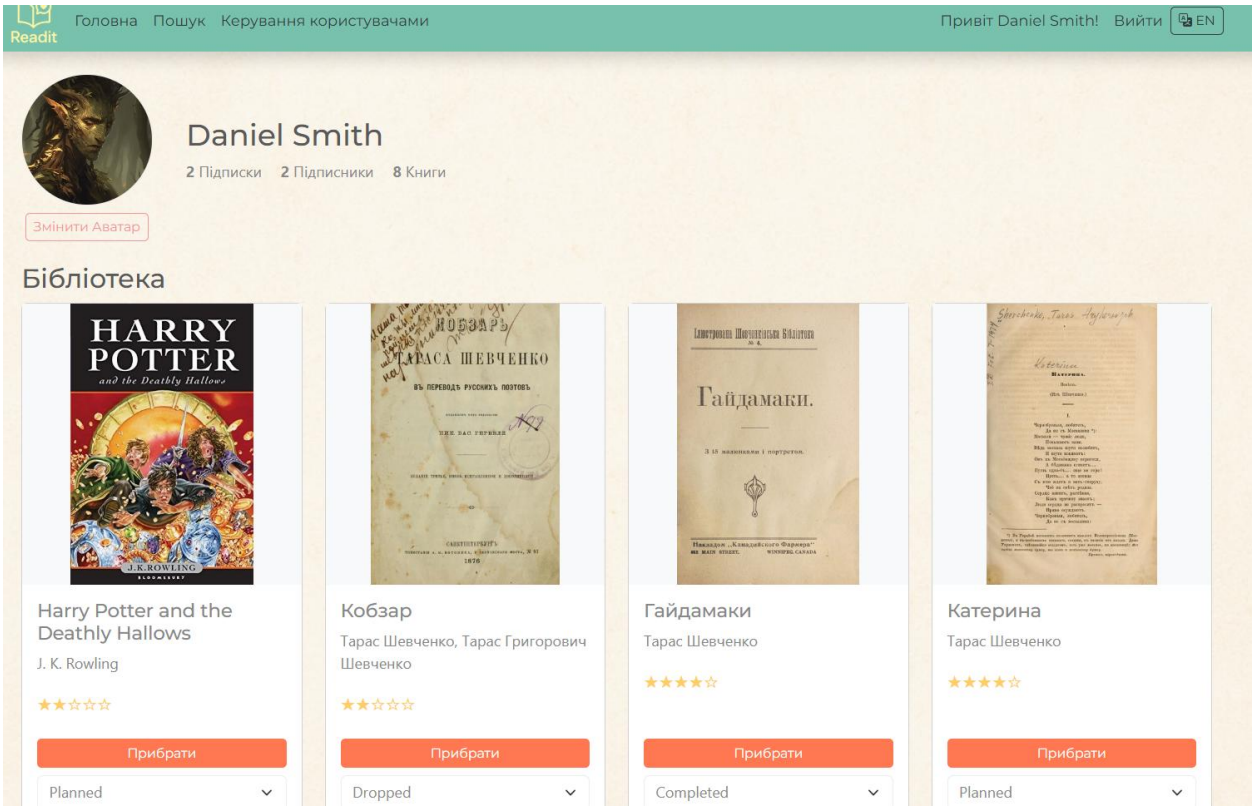


Рис. 3.2 Екранна форма додатку сторінки користувача

3.3 Реалізація системи друзів: додавання, перегляд бібліотек, фільтри

Дружба у Readit реалізована як односторонній зв’язок — тобто як підписка. Користувач може додати іншого користувача до списку друзів без взаємного підтвердження. Це дозволяє уникнути складних сценаріїв очікування або запрошень, і водночас формує систему, подібну до тих, що використовуються в сучасних соціальних медіа таких як: Instagram, YouTube, LinkedIn. Такий підхід забезпечує простоту та швидкість взаємодії, а також дозволяє краще масштабувати систему у майбутньому[14].

Процес додавання друга відбувається через відповідну кнопку на сторінці профілю іншого користувача або в загальному списку користувачів рисунок 3.3 . Після додавання, цей користувач автоматично з’являється у списку «друзів» у

профілі, а всі його книги стають доступними для перегляду на сторінці бібліотеки, якщо вони публічні.

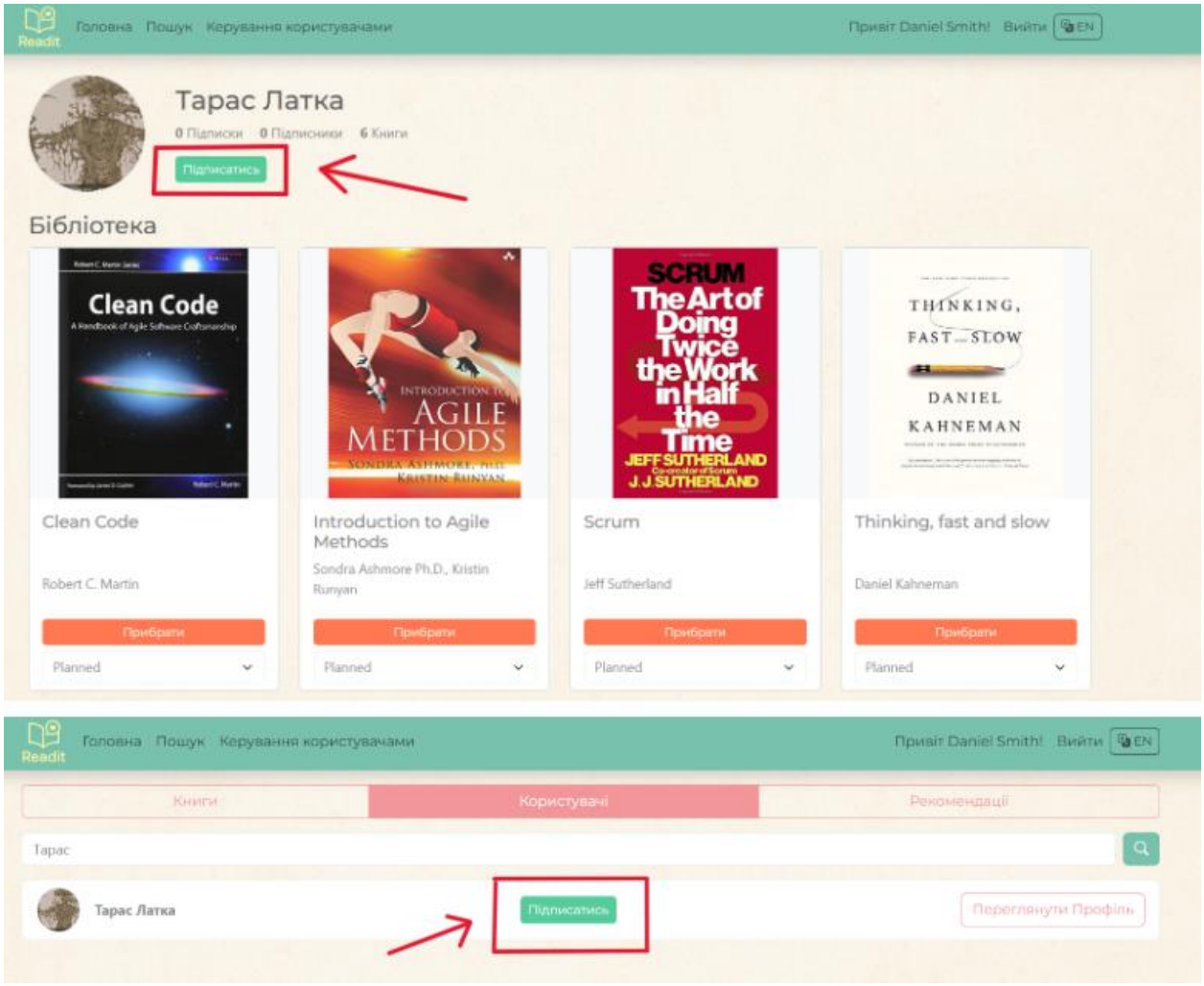


Рис. 3.3 Екранні форми підписки на іншого користувача

У моєму застосунку реалізовано можливість переглядати бібліотеки друзів — користувач бачить ті самі книжкові картки, що що додав інший користувач, але з персональним статусом та оцінкою. Це створює додатковий шар контексту і дозволяє не просто дізнатися, що прочитав друг, а й як він це сприйняв. Такий підхід стимулює живе обговорення книг, пошук нових видань через знайомі оцінки, а також підвищує читацьку мотивацію: якщо хтось із друзів щойно прочитав щось цікаве, є шанс, що й інші спробують.

Архітектурно дружні зв’язки реалізовано через окрему таблицю Friendship, яка містить ідентифікатор відправника та ідентифікатор одержувача — таким чином кожен зв’язок є окремим записом і легко обробляється в обидві сторони. Всі дані про користувача зібрані на окремій сторінці підписок рисунок. 3.4 Це дозволяє ефективно реалізовувати перегляд «взаємних друзів», «хто на тебе підписався», або «користувачі, які додали ті самі книги».

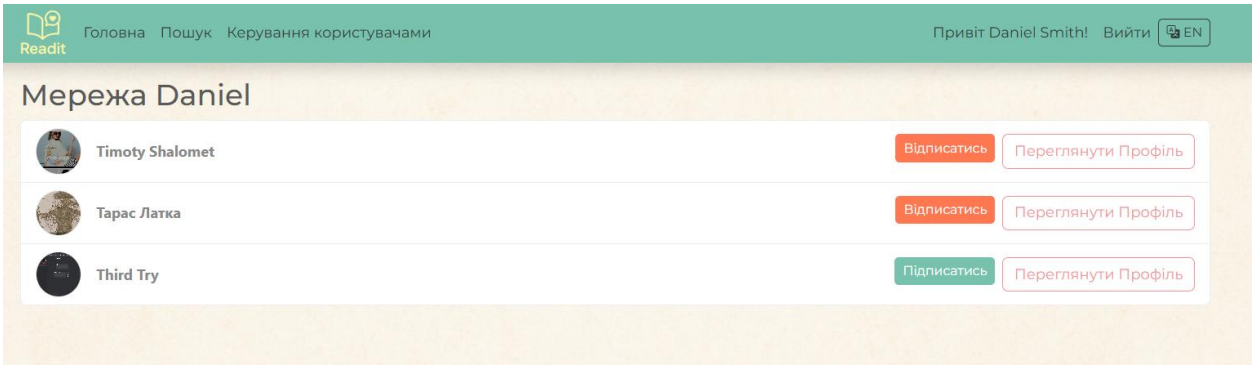


Рис. 3.4 Перегляд взаємодії профілю із іншими користувачами

3.4 Система рекомендацій за вподобаннями друзів

На відміну від універсальних алгоритмів, що ґрунтуються виключно на жанрових фільтрах або популярності, у моєму застосунку реалізовано персоналізовану модель рекомендацій, побудовану на вподобаннях друзів. Це створює живий, соціально орієнтований простір, де джерелом натхнення стають не абстрактні рейтинги, а конкретні люди зі схожими читацькими смаками.

Суть полягає в тому, що система формує список книг, які були додані друзями користувача. Якщо книга була прочитана кількома друзями, вона з’явиться у блоку рекомендацій першою, що зображено на рисунку 3.5 . Такий підхід не потребує складної обробки текстів або аналізу жанрової структури, але водночас ґрунтується на реальних діях реальних людей, які відслідковуються користувачем.

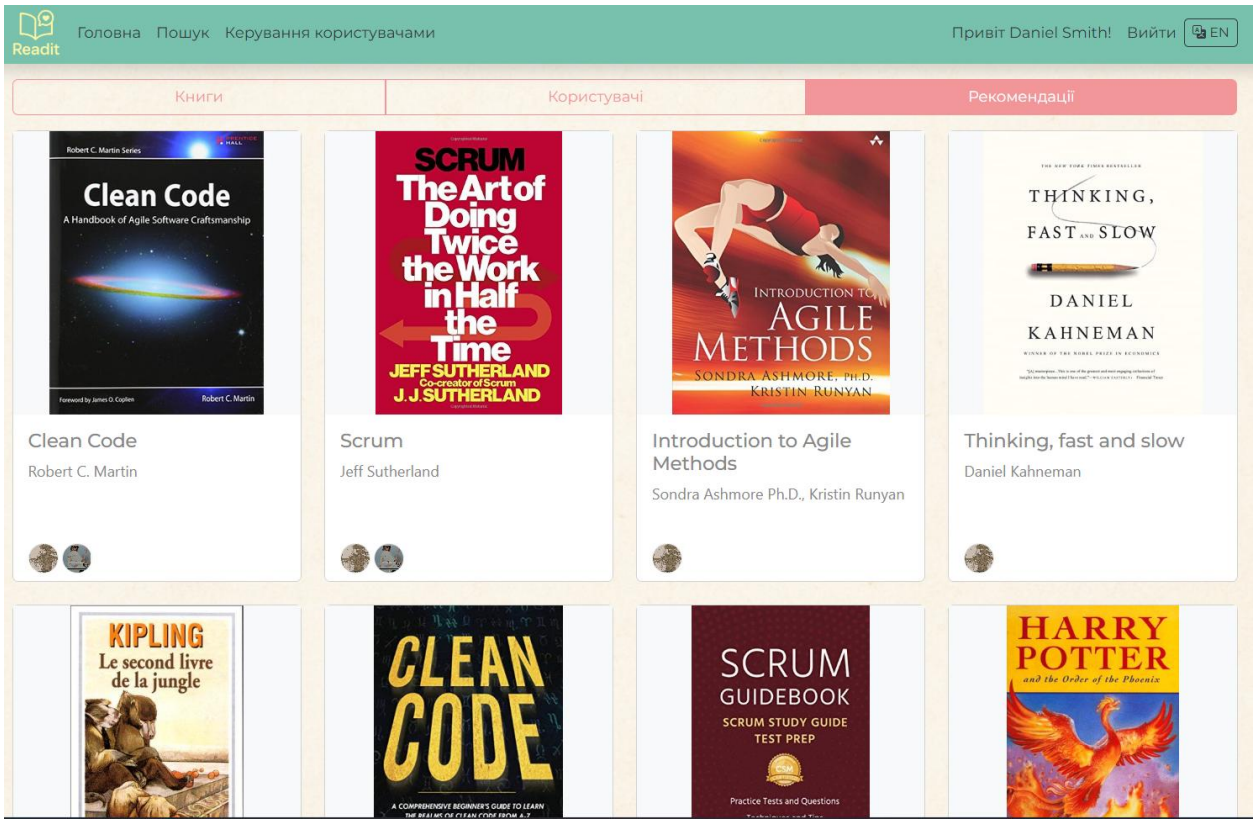


Рис. 3.5 Екранна форма рекомендацій за вподобаннями відслідковуваних акаунтів

Це також вирішує проблему холодного старту — коли новий користувач ще не має власної історії читання, але вже може бачити, що читають його друзі, і таким чином вбудовується у систему. Якщо ж навпаки, користувач давно у Readit, його бібліотека дає змогу іншим орієнтуватися на нього як на активного читача.

Механізм відображення аватарок друзів, які додали певну книгу, реалізований у вигляді інтерактивного елементу на картці книги. На рівні бекенду, кожна книга, яка потрапляє у список рекомендацій, містить список об'єктів користувачів, які її додали. Ці дані передаються у вигляді властивості AddedByUsers, що заповнюється в моделі сторінки на основі запитів до бази даних. Під час побудови інтерфейсу Razor-компоненти перевіряють, чи є в моделі книги друзі поточного користувача, які мають цю книгу у своїй бібліотеці. Якщо такі користувачі є, то їхні аватарки відображаються у нижній частині картки книги.

Для кожного зображення використовується стилізоване відображення: це маленька кругла аватарка з обрамленням, яка додається до горизонтального блоку. Під час наведення курсору на таку аватарку спрацьовує JavaScript-механізм, що показує підпис у вигляді спливаючого меню — в ньому вказується ім’я та прізвище користувача, а також активне посилання на його профіль. Таким чином, користувач платформи Readit одразу бачить, що певну книгу читає його друг, і має змогу миттєво перейти до його профілю, щоб дізнатися більше або ознайомитися з відгуками.

3.5 Аватар користувача, профіль

Після реєстрації, якщо користувач не завантажує власне зображення, йому автоматично встановлюється дефолтний аватар — сірий кружечок із умовним зображенням людини. Цей підхід дозволяє одразу сформувати цілісний вигляд інтерфейсу навіть для нових облікових записів. Користувач у будь-який момент може змінити аватар через спеціальну кнопку редагування аватару, яка розташована на головній сторінці користувача. Якщо користувач заходить на інший профіль - форма зникає, так як він не має доступу до редагування деталей інших профілів.

При виборі зображення користувач завантажує файл із локального пристрою. Файл перевіряється за типом (лише зображення у форматах .jpg, .png, .webp), це перевіряється як на фронтенді при виборі файлу, так і на сервері, де додатково контролюється розширення.

Якщо користувач не вибрав файл або спробував надіслати порожній запит, серверна логіка припиняє виконання дії (AvatarUpload == null || AvatarUpload.Length == 0), що запобігає виникненню винятків або збереженню порожнього файлу.

Незалежно від розміру вихідного файлу, на сторінці аватар відображається зі стилем object-fit: cover у форматі кола (rounded-circle), із жорстко заданими розмірами (140×140 пікселів у профілі). Це дозволяє уникнути спотворень, навіть якщо користувач завантажив нестандартне зображення.

Для уникнення ситуацій, коли браузер показує старе зображення з кешу після оновлення аватара, у src додається параметр ?ts=@DateTime.Now.Ticks. Це унікальний часовий штамп, що примушує браузер завантажити свіжу версію зображення після кожного оновлення.

Після цього файл зберігається у внутрішню папку проєкту (wwwroot/avatars) під унікальним іменем, сформованим на основі ідентифікатора користувача й мітки часу. Це дозволяє уникати конфліктів і випадкового перезапису файлів із однаковими назвами.

У моделі користувача (User) зберігається лише назва файлу аватару, а не сам файл або його абсолютний шлях[11]. При відображенні профілю аватар завантажується динамічно, шляхом підстановки збереженого імені в URL до папки. Якщо зображення з певних причин не знайдено (наприклад, його було видалено), система автоматично показує дефолтне зображення — це виключає помилки та поламані посилання.

На сторінці профілю аватар відображається у вигляді круглого елемента (з фіксованими розмірами), розташованого поруч з іменем та списком друзів. Для зручності, навіть у списках друзів чи в коментарях книга користувача позначається його аватаром — у мініатюрному розмірі.

3.6 Система оцінювання та коментування

У структурі реалізації оцінювання та коментування в Readit ключову роль відіграє чітке розмежування між різними типами взаємодії користувача з книгою: особиста оцінка, відгук (коментар) та соціальний контекст (профіль автора, дата, зв'язок з рейтингом). Архітектура проєкту побудована так, щоб усі дії були індивідуалізованими та інтегрованими у загальний користувацький досвід, без перезавантаження сторінки та з чіткими обмеженнями доступу.

Рейтинг книги зберігається в таблиці UserBooks. Це — сутність, яка поєднує конкретного користувача та конкретну книгу. Якщо користувач поставив оцінку, вона записується в поле Rating. Коментарі зберігаються окремо в таблиці Comments, яка має зв’язок із користувачем (UserId) та з книгою (WorkKey). Кожен коментар фіксує текст, дату створення і є окремою сутністю, не залежною від рейтингу, але може з ним співіснувати.

На сторінці детального перегляду книги BookDetails.cshtml реалізована форма оцінювання зірками — п’ять інтерактивних зірок, які підсвічуються при наведенні і зберігають вибраний рейтинг при кліку. Вибір відправляється як POST-запит на сервер без перезавантаження сторінки. Блок коментарів — нижче розміщується список усіх наявних відгуків до книги, якщо вони є. Кожен коментар відображає:

- аватар автора (завантажений або стандартний),
- ім’я користувача (з посиланням на його профіль)
- дату публікації
- текст відгуку
- зіркову оцінку (якщо така була поставлена) — витягується із UserBooks і виводиться за допомогою словника CommenterRatings.

Із коментарем можна взаємодіяти наступним чином :

- Додавання: якщо користувач залишив новий коментар, він відправляється через POST-запит і зберігається у базу.
- Редагування: при натисканні кнопки «Редагувати» виконується завантаження тексту в текстове поле, що дозволяє оновити коментар без створення нового.
- Видалення: доступне лише автору або адміністратору. У разі підтвердження, запис видаляється з бази.
- Усі дії мають перевірку прав доступу — щоб уникнути маніпуляцій з боку інших користувачів.

Таким чином на рисунку 3.6 Зображена взаємодія користувачів із коментарями до книги

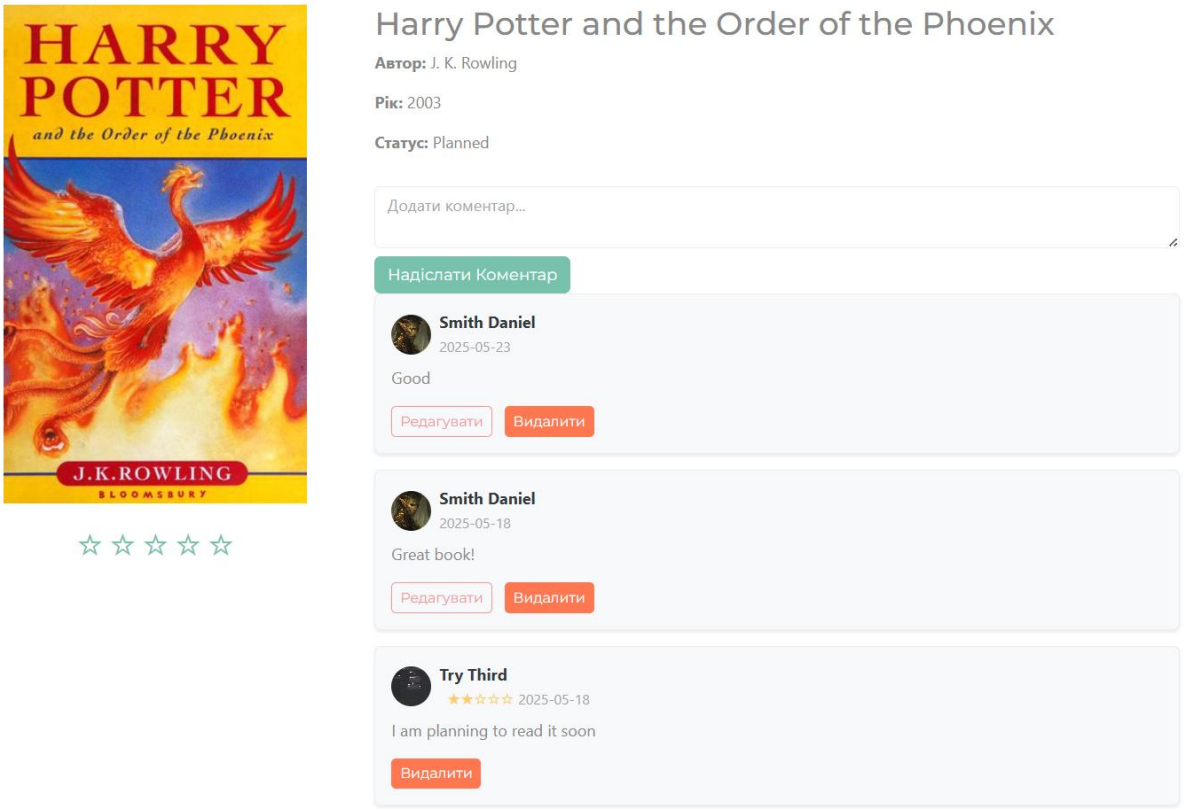


Рис. 3.6 Вигляд блоку коментарів із акаунту адміністратора

Усі форми (оцінювання, коментування, редагування) працюють асинхронно, тобто без перезавантаження сторінки. Це досягається за допомогою Razor Pages, частково JavaScript (для зірок) і стандартних форм POST із перезапитом даних у відповідь. Завдяки цьому: система працює плавно, DOM оновлюється лише частково, користувач одразу бачить результати своїх дій.

У BookDetails.cshtml.cs реалізовано логіку для:

- OnPostRateAsync — обробка рейтингу;
- OnPostAddCommentAsync — додавання коментаря;
- OnPostEditCommentAsync — збереження оновленого тексту;
- OnPostDeleteCommentAsync — перевірка прав та видалення;

- OnPostEditCommentFormAsync — передзавантаження сторінки з відкритою формою редагування.

Усе це дає змогу зберігати одну сторінку для повного взаємодійного циклу — від читання до коментування, не створюючи додаткових сторінок або дублювання коду.

3.7 Адміністративна частина: керування користувачами та ролями

Адміністративна панель Readit доступна лише користувачам, які мають роль "admin". Це реалізовано через атрибут: [Authorize(Roles = "admin")], який встановлюється у Razor Page /Pages/Admin/ManageUsers.cshtml [12]. Якщо користувач не є адміністратором — доступ блокується, і він перенаправляється на сторінку логіну або помилку 403.

Модель User розширює IdentityUser, що дозволяє зберігати основні облікові дані, такі як UserName, Email, PasswordHash. Додатково були додані поля:

- FirstName та LastName — для виведення повного імені у списку користувачів;
- AvatarFileName — для виведення аватарки;
- DateCreated (опційно) — можна додати для майбутнього логування/аналітики.

У файлі ManageUsers.cshtml.cs реалізовано логіку для завантаження списку користувачів через метод OnGetAsync(), у якому виконується завантаження всіх користувачів із бази. Після чого для кожного з них окремо перевіряється чи має він роль адміністратора. Також реалізовано підвищення до адміністратора на сторінці керування користувачами. При натисканні кнопки «Зробити адміністратором» на форму відправляється POST-запит з обробником. У ньому відбувається пошук користувача, далі додають до "admin", якщо він досі не мав таких прав доступу.

Інтерфейс сторінки ManageUsers.cshtml реалізовано за допомогою Razor Pages та Bootstrap. Основні елементи:

- Таблиця користувачів — з колонками: аватар, ім’я, email, роль, дата реєстрації;
- Кнопки для ролей — Promote, Demote, реалізовані у формі, яка передає ID користувача;
- Кнопка "View Profile" — посилання на UserPage для перегляду публічного профілю.

Для безпеки були створені обмеження де адміністратор не може зняти з себе роль, що запобігає втраті доступу. Та якщо у системі лише один admin, пониження блокується.

3.8 Висновки до розділу 3

Результатом моєї роботи стало створення зручної, динамічної та гнучкої платформи, яка дозволяє користувачам не лише зберігати книги, а й формувати особисте читацьке середовище.

Одним із перших кроків я реалізувала інтеграцію з OpenLibrary API, яка дозволяє шукати книги без створення власної бази метаданих. Користувачі можуть додати книгу до бібліотеки всього в один клік — і вона одразу зберігається з персональними параметрами. Завдяки цій інтеграції я оптимізувала час розробки та значно зменшила обсяг початкових даних.

Я впровадила механізм персональної бібліотеки, де кожна книга зберігається окремо для кожного користувача. Статуси, оцінки, коментарі — усе це не загальні, а індивідуальні дані. Я забезпечила, щоб книга могла бути однаковою для кількох користувачів, але досвід взаємодії з нею — унікальним для кожного.

Ще однією важливою частиною стала система дружби, реалізована у форматі підписок. Я зробила так, щоб можна було стежити за активністю інших користувачів,

бачити їхню бібліотеку, і навіть отримувати рекомендації на основі їхніх доданих книг. Для цього я налаштувала фільтрацію контенту за друзями, інтегрувала відображення аватарок біля карток книг і реалізувала асинхронну логіку для плавної роботи інтерфейсу.

Також я повністю реалізувала систему оцінювання та коментування книг. Кожен користувач може залишати власну оцінку у вигляді зірок і писати коментарі, які доступні для редагування чи видалення. При цьому я розмежувала логіку: оцінки зберігаються в UserBooks, а коментарі — у власній таблиці, що забезпечує масштабованість і незалежність.

Створила особисту сторінку користувача, де реалізована можливість змінювати аватар, переглядати друзів та власну бібліотеку.

Реалізувала адміністративну панель, яка дозволяє керувати користувачами, змінювати їхні ролі (user/admin) і бачити основну інформацію. Цей функціонал я створила з урахуванням безпеки, ролей та контролю над доступом до критичних функцій.

4 ТЕСТУВАННЯ WEB-ЗАСТОСУНКУ

4.1 Вибір моделі тестування

V-модель тестування (від англ. Verification and Validation Model) — це одна з класичних моделей життєвого циклу ПЗ, яка базується на принципі симетричності між етапами розробки та відповідними етапами тестування. У вигляді діаграми вона формує букву V, де ліва частина представляє етапи аналізу та проєктування, а права — відповідні типи тестування.

У контексті мого проєкту ця модель виявилась особливо корисною, оскільки Readit має чітко поділену логіку на окремі сервіси, компоненти інтерфейсу та функціональні модулі (бібліотека, рекомендації, друзі, профіль, адмін-панель). V-модель дозволила:

- системно тестувати кожен функціональний блок ще до інтеграції в загальну систему;
- передбачити специфічні тести для інтеграції між Razor Pages і сервісними класами;
- зафіксувати зміни на ранніх етапах, ще до появи складно виявлюваних багів;
- уникнути дублювання логіки перевірки: кожен тип тесту відповідав своєму рівню абстракції.

Завдяки цій моделі було створено не лише тести для перевірки окремих сервісів (наприклад, LibraryService чи CommentService), але й запроваджено ручне тестування інтегрованих сценаріїв, що враховують повну взаємодію між користувачем, базою даних і зовнішніми API. Таким чином, я змогла досягти більш надійної, передбачуваної поведінки додатку та впевненості у стабільності його функціоналу перед фінальним розгортанням.

4.2 Опис рівнів тестування

У процесі розробки програмного забезпечення важливо не лише дотримуватись поетапної реалізації, але й забезпечити відповідне тестування на кожному з етапів. Це дозволяє своєчасно виявляти та усувати помилки, підвищуючи якість кінцевого продукту. Нижче наведено відповідність між основними етапами розробки WEB-застосунку та типами тестування, які доречно застосовувати на кожному з них Таблиця 4.1.

Таблиця 4.1

Рівні тестування V-моделі

Етап розробки	Відповідний етап тестування
1. Визначення вимог (Requirements Analysis)	1. Приймальне тестування (Acceptance Testing)
2. Специфікація системи (System Design)	2. Системне тестування (System Testing)
3. Архітектурне проєктування (High-Level Design)	3. Інтеграційне тестування (Integration Testing)
4. Детальне проєктування (Low-Level Design)	4. Модульне тестування (Unit Testing)
5. Реалізація (Implementation / Coding)	Реалізація, яку далі тестується вручну

4.2.1. Визначення вимог

На цьому етапі я сформулювала головні вимоги до функціональності системи:

- Користувач повинен мати змогу шукати книги через сторонній сервіс — OpenLibrary API, без створення власної великої бази даних.
- Повинна бути можливість додавання книг до персональної бібліотеки — збереження інформації саме для конкретного користувача.
- До кожної книги в бібліотеці користувач має змогу встановити статус читання, оцінку (рейтинг), залишити коментар.
- Платформа повинна підтримувати систему друзів (підписок), щоб можна було переглядати бібліотеки інших користувачів.
- Система рекомендацій має будуватись на основі книг, які додали друзі.
- Кожен користувач має профіль із можливістю редагування аватара, переглядом та редагування власної бібліотеки та підписок.
- Повинна бути реалізована адміністративна панель, у якій адміністратори можуть переглядати користувачів, змінювати їхні ролі, видаляти порушників;
- Інтерфейс має бути зрозумілим, швидким і адаптивним, із підтримкою локалізації (англійською та українською).
- Система повинна бути захищеною, з розмежованими ролями доступу та перевіркою прав при виконанні критичних операцій.

Також було визначено нефункціональні вимоги:

- Підтримка асинхронної взаємодії — без повного перезавантаження сторінок (AJAX).
- Висока продуктивність при великій кількості даних (наприклад, при прокрутці списку книг або рекомендацій).
- Система повинна бути сумісною з основними сучасними браузерами (Opera, Microsoft Edge, Arc) з метою забезпечення доступності функціоналу незалежно від платформи користувача.

- Забезпечення надійного зберігання даних у базі SQL Server із чіткими зв’язками між сутностями.

Ці вимоги не були статичними — в ході реалізації деякі з них уточнювались, доповнювались або адаптувались. Наприклад, спочатку я не планувала реалізовувати підказки з аватарами друзів у рекомендаціях, але згодом вирішила, що це підвищує взаємодію між користувачами — і функціональність було додано.

4.2.2 Вибір архітектури та структури проєкту

Я вирішила розбити застосунок на окремі проєкти(шари), що дозволило краще структуризувати код і підготувати його до розширення. Такий підхід відповідає шаблону Layered Architecture (багаторівневої архітектури), де:

- Readit — відповідає за представлення (Razor Pages, логіка відображення).
- Readit.Models — зберігає всі класи-сутності (User, Book, UserBook, Comment, Friendship, ReadingStatus).
- Readit.DataAccess — працює з контекстом бази даних через ApplicationDbContext, репозиторіями й патерном UnitOfWork[15].
- Readit.Test - використовується для інтеграції тестування у додаток

Я спроєктувала модель даних із урахуванням логіки персональної бібліотеки та соціальної взаємодії. Основними сутностями стали:

- User: обліковий запис користувача (розширений IdentityUser)
- UserBook: зв’язок між користувачем і книгою — саме тут зберігаються статус, рейтинг, коментар
- Comment: окрема таблиця з відгуками користувачів до книги
- Friendship: структура підписки між користувачами (односторонній зв’язок)

Всі ці таблиці зв’язані через зовнішні ключі, а для запобігання дублюванню книг в базі — я використовувала унікальне поле WorkKey, яке приходить з OpenLibrary API.

Щоб не дублювати логіку у Razor Pages, я створила окремі сервіси:

- BookApiService — для взаємодії з OpenLibrary API (пошук, деталі книги),
- LibraryService — для додавання книг до бібліотеки, перевірки на дублікати, зміни статусів,
- UserService — для отримання інформації про користувачів, підписок, аватарів тощо.

Інтерфейс був спроєктований у вигляді адаптивних Razor-сторінок, де кожен розділ логічно розміщено:

- Search — пошук книг, друзів та рекомендації,
- UserPage — профіль користувача з його бібліотекою,
- BookDetails — сторінка книги з оцінкою та коментарями,
- FollowersList — перегляд підписок та підписників,
- Admin/ManageUsers — адміністрування.

Для стилізації використано Bootstrap 5, який забезпечив адаптивність інтерфейсу на різних пристроях. Його використання дало змогу мінімізувати час на розробку фронтенду й водночас отримати естетично привабливий вигляд завдяки використанням шаблонів.

Для вибору способі зберігання зображеннь, такі як аватарки,було обрано не базу даних, а проєктна папка /wwwroot/avatars, а в базу зберігається тільки їхня назва. Це рішення значно зменшує розмір БД та спрощує роботу з файлами.

Для запровадження безпеки було закладено:

- систему автентифікації та авторизації через ASP.NET Core Identity[16],
- дві ролі: user та admin,
- розмежування прав доступу через [Authorize] та [Authorize(Roles="admin")] атрибути.

Інтерфейс із самого початку було спроєктовано з урахуванням підтримки локалізації: усі написи реалізовані через IStringLocalizer та .resx-файли в Resources/Pages [13][18].

4.2.3 Архітектурне проєктування

Архітектурне проєктування у проєкті Readit виконує ключову роль у забезпеченні стійкості, масштабованості та підтримуваності WEB-застосунку. В умовах, коли система взаємодіє з зовнішніми API, має багаторівневу модель даних та включає функціонал для обробки користувацьких ролей, бібліотек, дружніх зв’язків і коментарів, стає необхідним чітке структурування взаємодії між компонентами.

У контексті застосування V-моделі тестування архітектурне проєктування відповідає етапу High-Level Design і має бути перевірене відповідним рівнем тестування. Його основне завдання — упевнитися, що всі основні блоки застосунку не лише правильно реалізовані, але й взаємодіють між собою відповідно до закладеної логіки.

Завдяки цьому підходу я виявила конфлікти у структурах даних (порушення зовнішніх ключів) та підготуватла платформу до модульного тестування без ризику базових помилок на рівні проєктування.

У моєму проєкті архітектура побудована навколо двох основних моделей: User та UserBook.

- User — реалізує облікові дані користувача, включаючи розширення від IdentityUser. Містить інформацію про ім’я, прізвище, email, аватар, а також взаємодіє з системою ролей [16].
- UserBook — є проміжною сутністю, яка поєднує користувача з книгою. Саме тут зберігається персоналізована інформація: статус, рейтинг, коментарі, дата додавання тощо.

Таке розділення дозволяє уникати дублювання книг у базі при одночасному збереженні повної персоналізації. Наприклад, одна й та сама книга з таблиці Books може бути присутня в бібліотеках багатьох користувачів, але кожен із них буде мати власний запис у UserBooks.

Для перевірки роботи такої структури я реалізувала інтеграційний тест AddBook_ThenFetchBooks_WorksWithRealDb рисунок 4.1, який симулює основну дію в застосунку — додавання книги до бібліотеки користувача та подальший її перегляд.

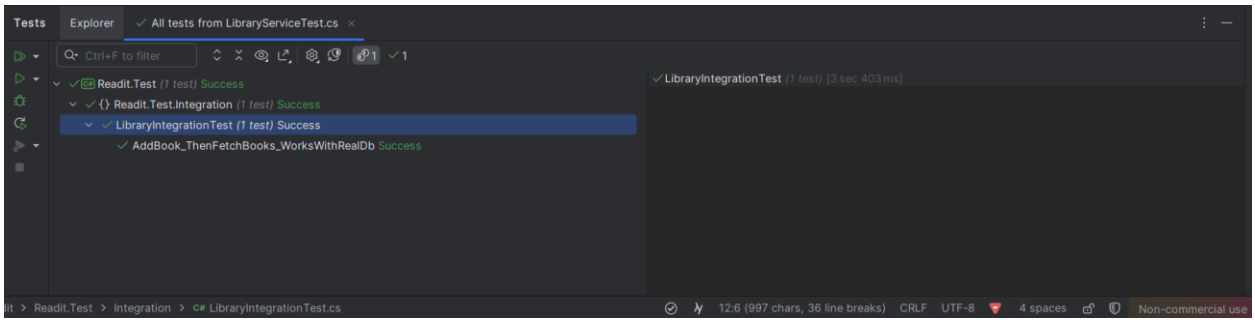


Рис. 4.1 Результат виконання тесту

У цьому тесті:

- Створюється in-memory база даних за допомогою UseInMemoryDatabase, що дозволяє перевірити логіку без підключення до реального SQL Server[15].
- Ініціалізується користувач та додається до бази через DbContext.
- Створюється новий запис у UserBooks, який прив’язується до цього користувача.

Здійснюється вибірка всіх книжок для користувача, і результат перевіряється за допомогою Assert.

Тест підтверджує, що:

- Зовнішні ключі працюють коректно (книга зберігається лише у зв’язку з користувачем).
- Дані не дублюються та доступні для подальшої обробки.
- Механізм взаємодії між сутностями працює відповідно до проєктної логіки.

4.2.4 Модульне тестування

У межах реалізації WEB-застосунку Readit було проведено модульне тестування ключових сервісів, що відповідають за логіку пошуку книг через OpenLibrary API та додавання їх до персональної бібліотеки. Модульне тестування виконувалося з метою перевірки правильності поведінки окремих компонентів системи в ізоляції — без залежності від бази даних, зовнішніх API чи інтерфейсу користувача.

Для створення юніт-тестів застосовувались xUnit як тестовий фреймворк та Moq для емуляції залежностей. Тестування проводилось у двох основних напрямах: взаємодія з OpenLibrary API та логіка роботи сервісу бібліотеки користувача[16].

Компонент BookApiService відповідає за відправлення HTTP-запитів до [9] OpenLibraryAPI та перетворення отриманих JSON-даних у модель OpenLibraryBook. Щоб уникнути реальних мережевих запитів під час тестування, було створено мок-об'єкт HttpClient, який імітує відповіді зовнішнього сервісу. Це дозволило перевірити різні сценарії:

- Чи правильно парсяться книги зі звичайного JSON (два успішні результати).
- Що буде, якщо запит повертає порожній список (книга не знайдена).
- Як поводиться метод при некоректному ключі ("").
- Чи правильно працює пошук конкретної книги за ключем (OL123).

Таким чином, була протестована вся критична логіка обробки зовнішніх даних.

Сервіс опрацьовує логіку додавання та видалення книг у персональну бібліотеку користувача. Основна перевірка стосувалася методу ToggleBookAsync, який змінює статус наявності книги в бібліотеці.

Для тестування було створено мок-об'єкти UserManager та IHttpContextAccessor, які дозволили імітувати реального користувача без потреби автентифікації.

Використовуючи InMemoryDatabase від Entity Framework Core, вдалося створити повністю ізольоване середовище з тестовою базою.

Були перевірені два сценарії:

- Книга ще не була в бібліотеці — метод повинен її додати, і результатом повертається true.
- Книга вже додана — метод повинен її видалити, повертаючи false.

На рисунку 4.2 підтверджено коректність реалізації додавання та видалення книги відповідно до очікуваної бізнес-логіки.

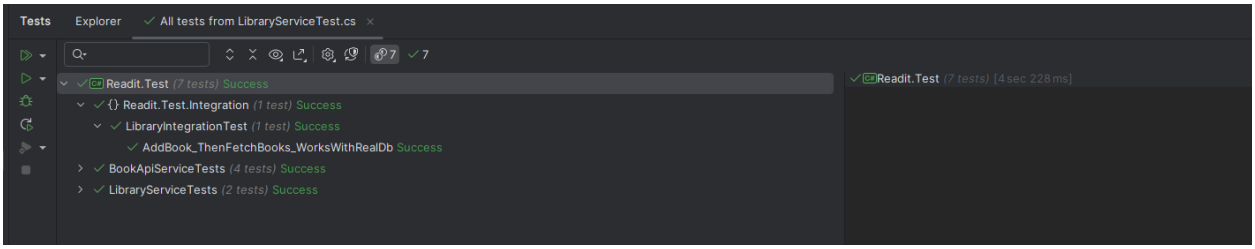


Рис. 4.2 Результати виконання тестів

4.2.5 Реалізація

Реалізація стала центральною фазою у життєвому циклі проєкту Readit, яка поєднує попереднє архітектурне та технічне проєктування із наступним тестуванням. На цьому етапі безпосередньо було створено програмний код WEB-застосунку з використанням стеку технологій ASP.NET Core, Razor Pages, Entity Framework Core та SQL Server.

Було реалізовано кілька ключових модулів:

- BookApiService — для взаємодії з OpenLibrary API, пошуку книг та отримання метаданих;
- LibraryService — для додавання та видалення книг із персональної бібліотеки;
- CommentService та логіка в BookDetails.cshtml.cs — для коментування та оцінювання;

- Система керування друзями та підписками — через таблицю Friendship;
- Адміністративна частина — із перевіркою ролей та керуванням правами доступу.

Особливу увагу приділено тому, щоб сервіси були ізольованими та придатними до тестування. Для цього використовувались інтерфейси, залежності передавались через конструктори (DI), а логіка взаємодії з базою даних структурувалась через DbContext[15].

Також на цьому етапі була інтегрована підтримка локалізації, адаптивного дизайну через Bootstrap 5 [14] і захист доступу через ASP.NET Core Identity. Отриманий код ліг в основу модульного, інтеграційного та ручного тестування, кожен тип якого перевіряє окремі рівні реалізованого функціоналу.

4.2.6 Ручне тестування

Ручне тестування у проєкті Readit дозволило перевірити ті аспекти взаємодії з користувачем, які складно охопити автоматизованими тестами, зокрема — поведінку інтерфейсу, візуальні зміни, динамічне оновлення сторінок та контроль доступу. Тестування проводилось безпосередньо у браузері під час використання застосунку, з обліковими записами звичайного користувача та адміністратора.

Було перевірено додавання книги до бібліотеки. При натисканні кнопки «Додати» на картці книги, вона змінюється на «Видалити», що підтверджує успішне додавання до персональної бібліотеки. Водночас елемент не викликає перезавантаження сторінки, а зміни фіксуються в базі даних рисунок. 4.3 .

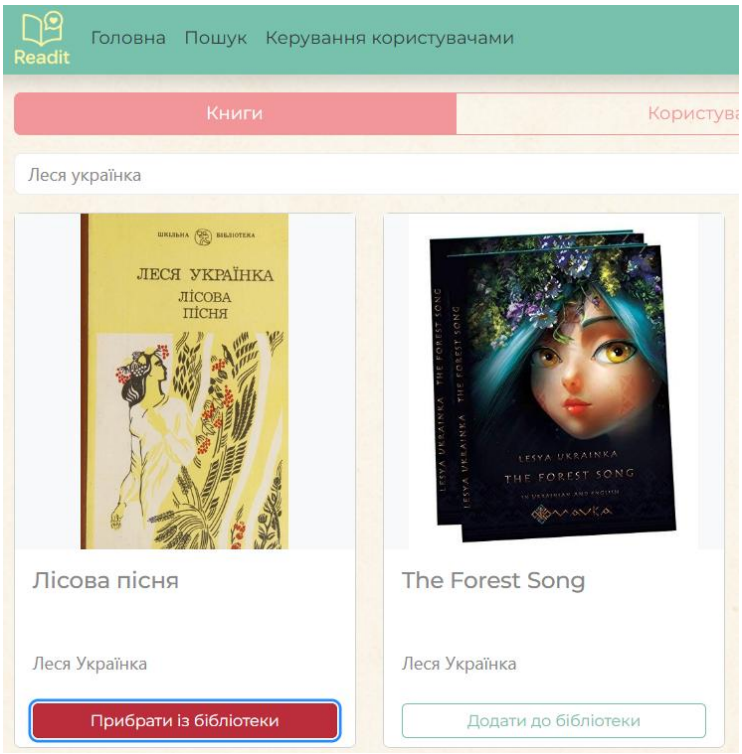


Рис. 4.3. Додавання книги до бібліотеки — зміна кнопки на «Прибрати із бібліотеки».

Протестовано оцінювання книги за допомогою зірок. Після додавання книги з’являється можливість поставити оцінку. Перевірено, що натискання на зірку зберігає обраний рейтинг у базі, а в інтерфейсі зірки підсвічуються відповідно до вибору. Також оцінка відображається у коментарях та рекомендаціях Рис. 4.4.

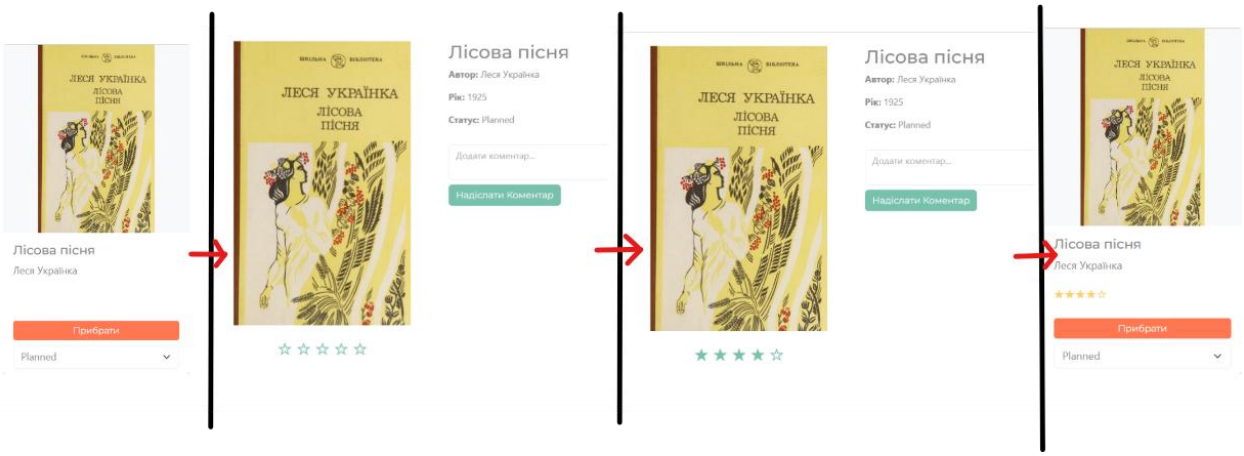


Рис. 4.4 Сценарій оцінювання книги зірками.

Коментування книги та редагування

Коментарі доступні лише після додавання книги. Було перевірено, що користувач бачить свій аватар, ім’я, дату коментаря, а також кнопки редагування і видалення. Інші користувачі або адміністратори не мають права редагувати чужі записи рисунку. 4.5.

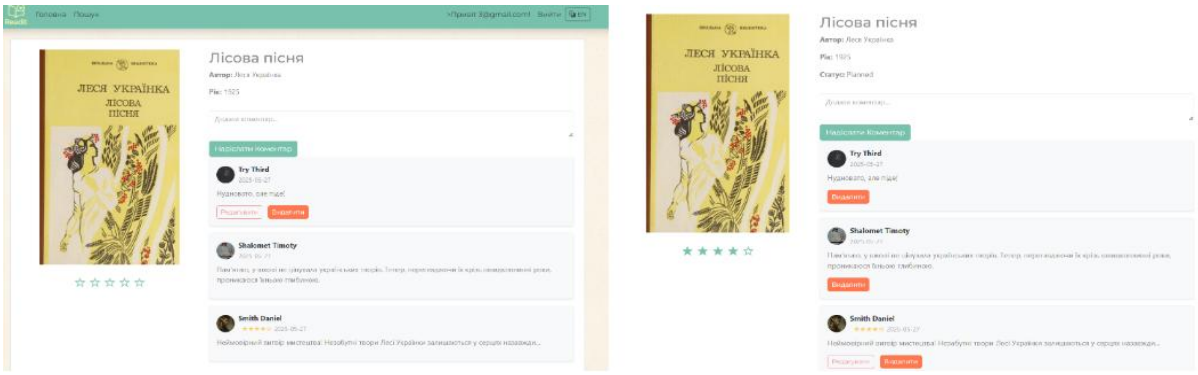


Рис. 4.5 Коментар користувача з можливістю редагування.

Відображення рекомендацій друзів

У режимі «Рекомендації» на сторінці пошуку відображаються книги, які додали користувачі зі списку друзів. Під кожною такою книгою показуються аватарки друзів, які додали її до бібліотеки. При наведенні на аватар з’являється підказка з іменем користувача рисунок 4.6.

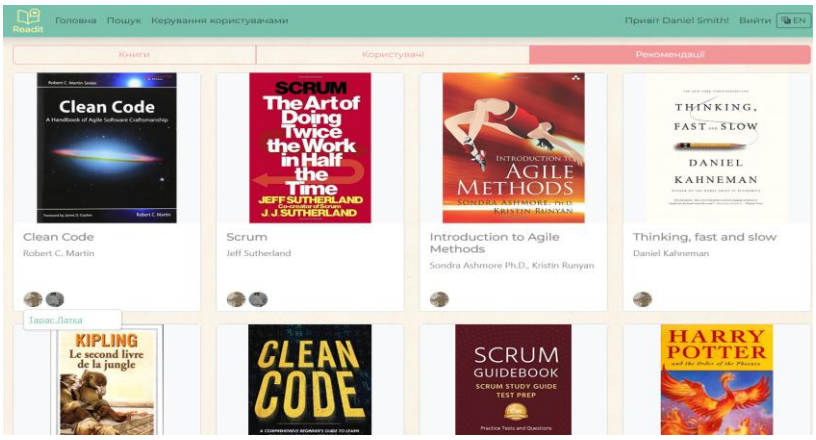


Рис. 4.6. Рекомендація від друга з підказкою при наведенні на аватар.

Перемикання мови інтерфейсу

Тестувалась можливість зміни мови між українською та англійською. Перемикання працює миттєво — всі написи у застосунку автоматично оновлюються, без перезавантаження або втрати даних на сторінці рисунок. 4.7.

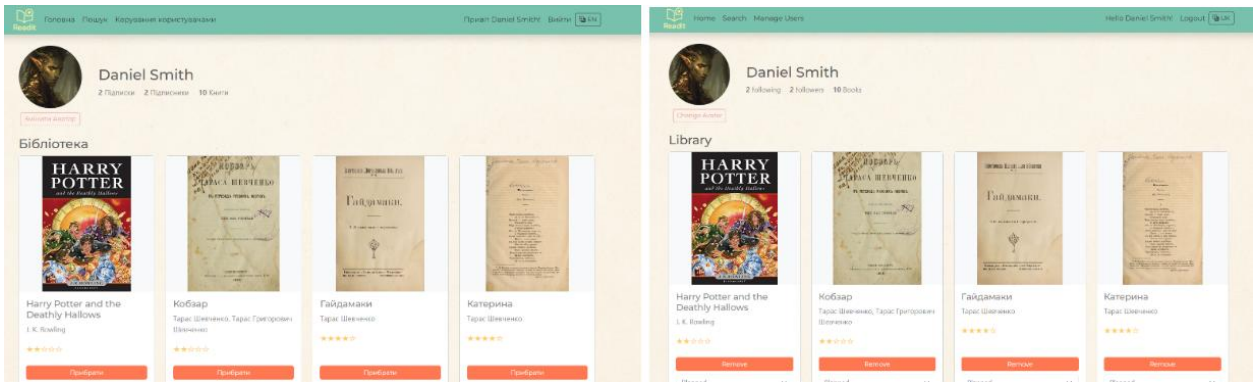


Рис. 4.7. Режим української та англійської мови у застосунку.

Адміністративна панель

Проведено перевірку керування користувачами з облікового запису адміністратора. Тестувались сценарії зміни ролі на admin, блокування функції самозміни, візуальне виділення ролей та обмеження доступу через атрибут [Authorize(Roles = "admin")] рисунок 4.8.

Рис. 4.8. Панель адміністратора зі зміною ролі користувача.

Окрему увагу було приділено перевірці кросбраузерної сумісності. Основні сценарії використання WEB-застосунку Readit тестувались вручну в різних браузерах: Opera, Microsoft Edge та Arc. Перевірка охоплювала ключові аспекти інтерфейсу — коректність верстки, стабільність асинхронних запитів, роботу динамічних елементів (наприклад, кнопок додавання книги, зіркових оцінок, фільтрів), перемикання мови, та правильне відображення адаптивного дизайну на різних розмірах екранів. У результаті тестування не було виявлено критичних відмінностей у поведінці чи зовнішньому вигляді між браузерами, що підтверджує високу ступінь сумісності застосунку із сучасними платформами. Результати перевірки кросбраузерної сумісності зображені на: Рис 4.9, Рис 4.10, Рис 4.11

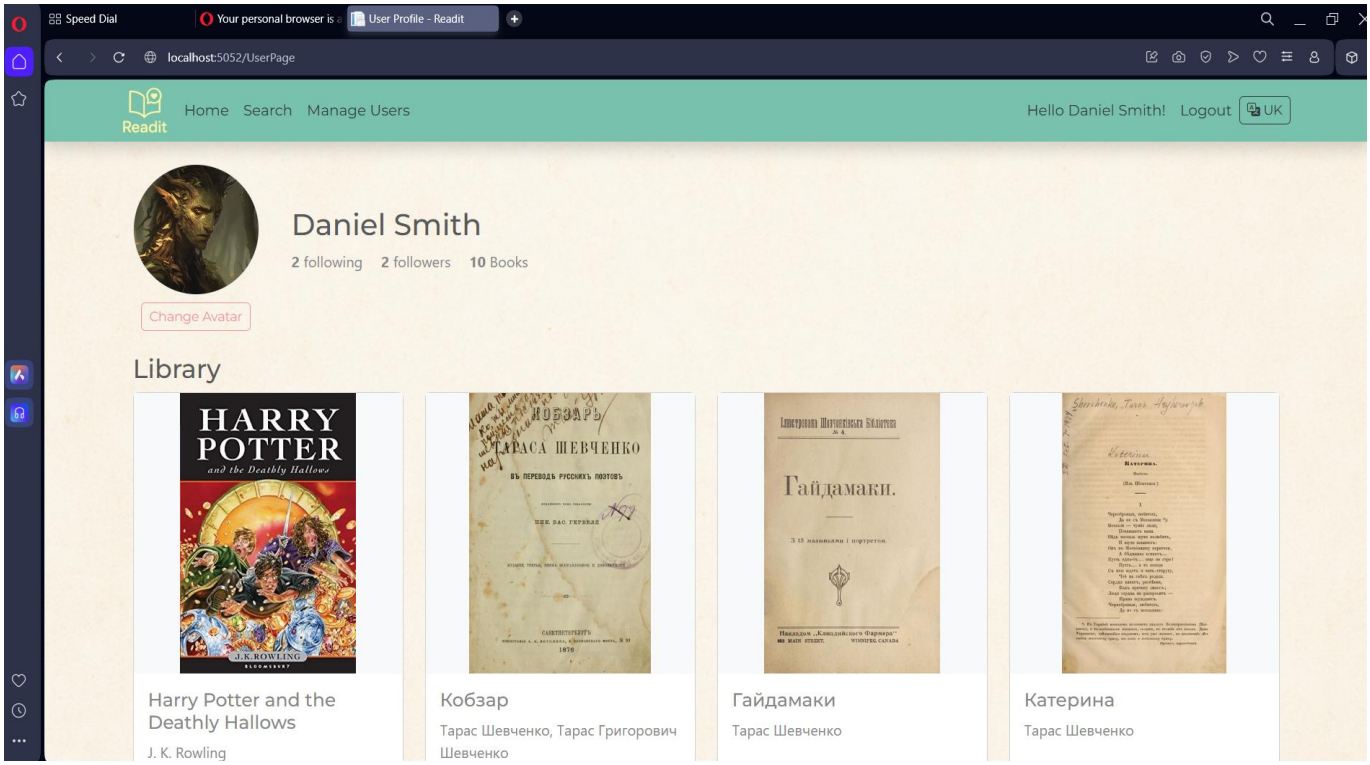


Рис 4.9 Запуск додатку у браузері Opera

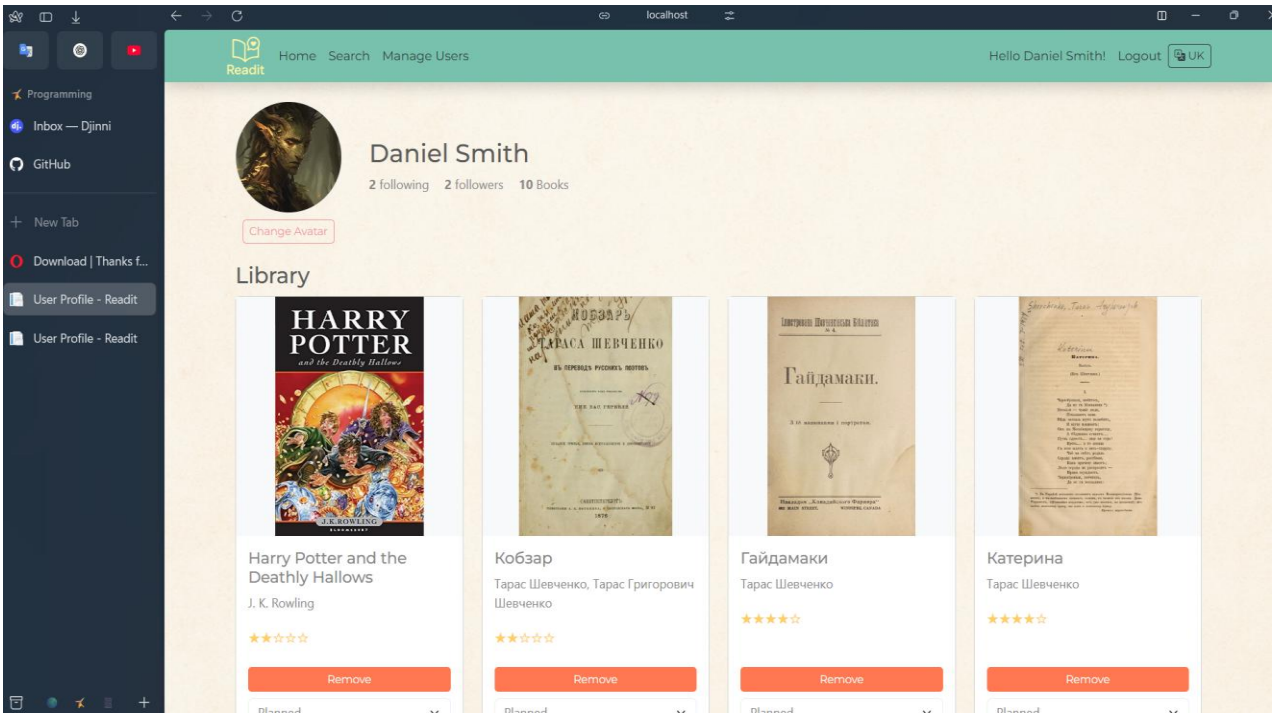


Рис 4.10 Запуск додатку у браузері Arc

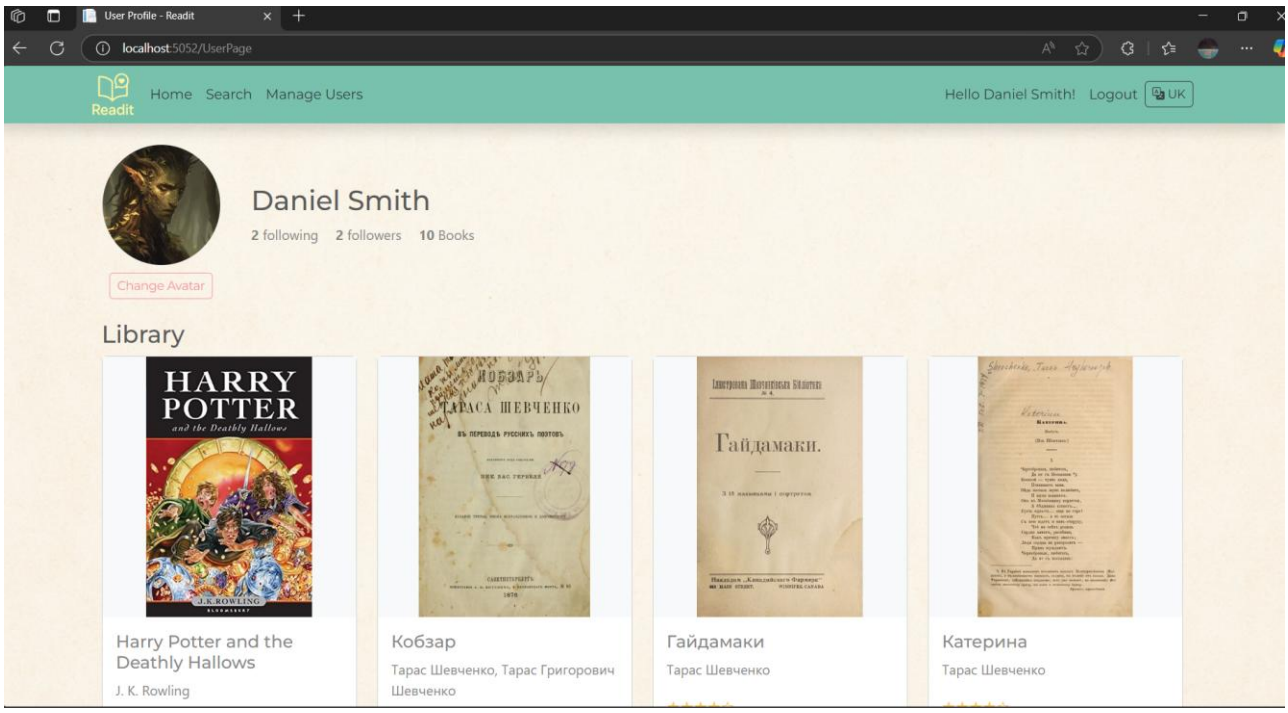


Рис 4.11 Запуск додатку у браузері Microsoft Edge

4.3 Тестове середовище та методи тестування

Тестування WEB-застосунку Readit відбувалося у локальному середовищі розробника, із повноцінною емуляцією основних сценаріїв роботи користувача. Для запуску та перевірки застосунку використовувалась IDE Rider, внутрішній Kestrel сервер, а також інтегрована база даних на основі InMemoryDb — що дозволяло виконувати інтеграційні тести без впливу на продуктивне середовище.

Тестове середовище включало такі компоненти:

- .NET SDK (версія 8.0) — для компіляції та запуску;
- EF Core InMemoryProvider — для створення ізольованої тестової бази даних;
- xUnit — як фреймворк модульного тестування;
- Moq — для створення фейкових об'єктів UserManager, HttpContext та зовнішніх залежностей[17];
- HttpClient з кастомним MockHandler — для імітації відповідей від OpenLibrary API у BookApiServiceTests.

Методи тестування розділялись за рівнями згідно з V-моделлю:

- Модульне тестування (Unit Testing) застосовувалось для перевірки окремих сервісів, таких як BookApiService і LibraryService. Для цього використовувались підготовлені тести, які моделювали очікувані вхідні дані та перевіряли відповідь функції (наприклад, додавання книги, пошук за ключем тощо).
- Інтеграційне тестування (Integration Testing) перевіряло зв’язки між сутностями User, Book, UserBook у базі даних. У класі LibraryIntegrationTest імітувався сценарій додавання користувача, запису книги в UserBooks та подальшого читання цих даних. Це дозволило переконатися, що вся ланка DbContext → таблиці → сервіси працює як очікується.
- Ручне тестування (Manual Testing) було застосоване до кінцевого UI: перевірялась логіка додавання/видалення книг, зміни статусів, завантаження аватарів,

підписки на друзів, коментування, зміна мови інтерфейсу. Це здійснювалось через браузер у режимі реального користувача без застосування автотестів, із врахуванням поведінки елементів DOM, обробки подій та асинхронної взаємодії.

- Функціональне тестування включало перевірку повних користувацьких сценаріїв: від пошуку книги через API до її оцінювання, перегляду рекомендацій друзів та змін у профілі.

4.4 Покриття функціональності

У процесі тестування WEB-застосунку Readit було перевірено основну функціональність кожного модуля згідно з запланованими можливостями проєкту.

- Клас BookApiServiceTests перевіряє роботу методів SearchBooksAsync та GetBookDetailsByKeyAsync. Перевірено: коректність десеріалізації відповіді API, правильність обробки порожньої відповіді, повернення null при відсутності збігів та роботу з декількома результатами.
- Клас LibraryServiceTests, метод ToggleBookAsync. Перевірено що книга додається, якщо її ще немає, книга видаляється, якщо вона вже додана та ідентифікація користувача відбувається через HttpContext.
- Клас LibraryIntegrationTest у якому перевірено: створення користувачаб, додавання книги до UserBooks, зчитування даних із бази через EF Core, відповідність збережених назв, ключів, статусів.
- Оцінювання книг у якому відбувалась перевірка рейтингу зберігається у таблиці UserBooks, що зірки відображаються згідно з попереднім вибором, асинхронність збереження змін без перезавантаження сторінки та перевірку на повторну зміну оцінки.

У реалізації додавання та редагування коментарів перевірена доступність форми коментарів лише після додавання книги. Також перевірені:

- перевірка відображення аватара, дати, рейтингу;

- можливість редагування власного коментаря;
- відображення кнопок лише для автора/адміна;
- серверна перевірка прав доступу.

- За системою підписок і рекомендацій перевірено додавання користувачів до списку друзів, фільтрація рекомендацій по друзях, відображення аватарок друзів, які додали книгу та підказка при наведенні — tooltip з іменем.
- Адміністративна панель протестована за зміною ролей через кнопки "Зробити адміністратором / Повернути користувача", перевіркою відображення червоного кольору для ролі admin, обмеженням на самозміну ролі та
  - обмеження доступу до сторінки за [Authorize(Roles = "admin")]

4.5 Висновки до розділу 4

У цьому розділі я реалізувала повноцінний процес тестування WEB-застосунку Readit, обравши V-модель тестування як основу для побудови логіки перевірки на всіх рівнях життєвого циклу ПЗ. Завдяки цій моделі мені вдалося вибудувати чіткий зв’язок між етапами розробки та відповідними етапами тестування, що дозволило забезпечити якість, надійність та передбачуваність поведінки системи.

Кожен етап тестування — від модульного до інтеграційного і завершуючи ручним тестуванням — був націлений на перевірку конкретних функціональних блоків і їхньої взаємодії. Я не лише протестувала технічну реалізацію сервісів, а й охопила повні користувацькі сценарії.

Особливо цінним стало ручне тестування, яке дозволило оцінити поведінку інтерфейсу, візуальні реакції, логіку кнопок та роботу локалізації. Важливим досягненням є також тестування безпекових аспектів, зокрема обмеження доступу до адмін-функціоналу та захисту критичних операцій.

На основі проведеного тестування можу з упевненістю сказати, що застосунок Readit:

- відповідає функціональним і нефункціональним вимогам;
- витримує навантаження і коректно реагує на взаємодії користувача;
- захищений від поширених помилок на рівні логіки, структури БД та ролей доступу;
- має гнучку структуру, яка дозволяє легко розширювати функціонал і підтримувати застосунок у майбутньому.

Таким чином, завдяки системному підходу до тестування, застосунок було приведено до стабільного і готового до розгортання стану, що є важливою умовою завершення розробки та передачі продукту для реального використання.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було реалізовано повноцінний web-застосунок Readit — платформу для обміну книжковими вподобаннями з можливістю зберігати персональну бібліотеку, взаємодіяти з іншими користувачами та отримувати персоналізовані рекомендації на основі активності друзів.

1.Проведено аналіз предметної області, що охоплює цифрові бібліотеки та сервіси з рекомендаціями і соціальною взаємодією користувачів.

2.Досліджено особливості існуючих рішень та визначено потребу у створенні web-застосунку для організації особистих бібліотек із рекомендаціями за жанрами. Були виявлені основні недоліки існуючих рішень: обменежна соціальна взаємодія, відсутність персоналізації власної бібліотеки.

3.Сформовано функціональні та нефункціональні вимоги до системи, з урахуванням потреб кінцевого користувача.

4.Розроблено архітектуру клієнт-серверного web-застосунку із розмежуванням доступу та підтримкою ролей. Для реалізації автентифікації та авторизації було використано ASP.NET Core Identity, зберігання ролей і користувачів організовано у SQL Server.

5.Реалізацію функціоналу особистої бібліотеки, статусів книг, оцінювання, додавання друзів та перегляду чужих бібліотек виконано з використанням мови C# для серверної логіки, Razor Pages для побудови динамічного інтерфейсу, Entity Framework Core для взаємодії з базою даних, а також JavaScript для реалізації інтерактивних елементів на клієнтській стороні. Вся логіка побудована у межах фреймворку ASP.NET Core із застосуванням шаблонів Repository та Unit of Work.

6.Проведено тестування основного функціоналу, підтверджено відповідність вимогам.

Застосунок забезпечує зручну навігацію, швидке реагування на дії користувача без перезавантаження сторінки, захищений доступ до персональних даних, адаптивний дизайн і підтримку української та англійської мов. Усі ці характеристики роблять Readit сучасним, гнучким і масштабованим рішенням для взаємодії читачів.

Результати роботи демонструють можливість ефективного поєднання технологій .NET, відкритих API та об'єктно-орієнтованого моделювання для створення повноцінного соціального WEB-застосунку. У майбутньому система може бути розширена новими функціями — наприклад, обговореннями книг, списками прочитаного за роками, повідомленнями між користувачами чи мобільною версією.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Колосенко Т.Є., Золотухіна О.А.. Технологічне забезпечення цифрового сервісу обміну книгами: обґрунтування вибору платформи .NET. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.2025, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.101-102.
2. Колосенко Т.Є., Золотухіна О.А. Персоналізація користувацького досвіду у веб-застосунках: практичні рішення для книжкових платформ. V Всеукраїнська науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті», 16.05.2025, Київ, Державний університет інформаційно-комунікаційних технологій (подано до друку)

ПЕРЕЛІК ПОСИЛАНЬ

1. Goodreads API. Goodreads, Inc. [Електронний ресурс]. – Режим доступу: https://www.goodreads.com/api – Назва з екрана. – Дата звернення: 10.05.2024.

2. LibraryThing Developer API. LibraryThing. [Електронний ресурс]. – Режим доступу: https://www.librarything.com/services – Назва з екрана. – Дата звернення: 10.05.2024.

3. Knigoholic — українська платформа для рецензій. [Електронний ресурс]. – Режим доступу: https://knigoholic.com.ua – Назва з екрана. – Дата звернення: 11.05.2024.

4. Open Library API Documentation. Internet Archive. [Електронний ресурс]. – Режим доступу: https://openlibrary.org/developers/api – Назва з екрана. – Дата звернення: 28.05.2025.

5. Документація ASP.NET Core. Microsoft Learn. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/aspnet/core – Назва з екрана. – Дата звернення: 24.05.2024.

6. Документація Entity Framework Core. Microsoft Learn. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/ef/core – Назва з екрана. – Дата звернення: 24.05.2024.

7. JetBrains Rider Documentation. JetBrains. [Електронний ресурс]. – Режим доступу: https://www.jetbrains.com/help/rider/Introduction.html – Назва з екрана. – Дата звернення: 24.05.2024.

8. Microsoft SQL Server documentation. Microsoft Learn. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/sql/sql-server – Назва з екрана. – Дата звернення: 24.05.2024.

9. Use IHttpClientFactory to implement resilient HTTP requests. Microsoft Learn. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-

us/dotnet/architecture/microservices/implement-resilient-applications/use-httpclientfactory-to-implement-resilient-http-requests – Назва з екрана. – Дата звернення: 28.05.2025.

10. Working with AJAX in Razor Pages. Learn Razor Pages. [Електронний ресурс]. – Режим доступу: https://www.learnrazorpages.com/razor-pages/ajax – Назва з екрана. – Дата звернення: 28.05.2025.

11. Add Custom Fields To User in ASP.NET Core Identity. Tektutorialshub. [Електронний ресурс]. – Режим доступу: https://www.tektutorialshub.com/asp-net-core/add-custom-fields-to-user-in-asp-net-core-identity – Назва з екрана. – Дата звернення: 28.05.2025.

12. Identity model customization in ASP.NET Core. Microsoft Learn. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/customize-identity-model – Назва з екрана. – Дата звернення: 28.05.2025.

13. Localization in ASP.NET Core. Microsoft Learn. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/localization – Назва з екрана. – Дата звернення: 28.05.2025.

14. Bootstrap v5.3 Documentation. Bootstrap. [Електронний ресурс]. – Режим доступу: https://getbootstrap.com/docs/5.3/getting-started/introduction – Назва з екрана. – Дата звернення: 28.05.2025.

15. Using InMemory Database Provider in Entity Framework Core. Microsoft Learn. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/ef/core/testing/in-memory – Назва з екрана. – Дата звернення: 28.05.2025.

16. Unit testing with xUnit in ASP.NET Core. Microsoft Learn. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-dotnet-test – Назва з екрана. – Дата звернення: 28.05.2025.

17. Mocking in Unit Tests with Moq. Moq GitHub Repository. [Електронний ресурс]. – Режим доступу: https://github.com/moq/moq4/wiki/Quickstart – Назва з екрана. – Дата звернення: 28.05.2025.

18. How to Use IStringLocalizer in Razor Pages. Learn Razor Pages. [Електронний ресурс]. – Режим доступу: https://www.learnrazorpages.com/localization/localize-text – Назва з екрана. – Дата звернення: 28.05.2025.

19. Using Dependency Injection in Razor Pages. Microsoft Learn. [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection – Назва з екрана. – Дата звернення: 28.05.2025

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка web-застосунку для обміну книжковими вподобаннями

Виконала студентка 4 курсу
групи ПД-41
Колосенко Тетяна Євгенівна
Керівник роботи
К.т.н, доц., доцент кафедри ІПЗ Золотухіна Оксана Анатолівна

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи** - цифровізація процесу обміну книжковими вподобаннями між читачами завдяки використанню web-застосунку.
- **Об’єкт дослідження -** процес обміну книжковими вподобаннями.
- **Предмет дослідження** - технології та інструменти розробки web-застосунку для обміну книжковими вподобаннями.

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Провести аналіз предметної області цифрових бібліотек та онлайн-сервісів з рекомендаційною і соціальною складовою.
2. Дослідити існуючі рішення, виявити їх особливості та обґрунтувати необхідність створення нового web-застосунку для організації особистих бібліотек.
3. Сформувати функціональні та нефункціональні вимоги до web-застосунку для ведення особистої бібліотеки та системи рекомендацій.
4. Спроєктувати архітектуру клієнт-серверного web-застосунку з підтримкою розмежування прав доступу.
5. Реалізувати web-застосунок із використанням ASP.NET Core, Razor Pages, Entity Framework Core та SQL Server.
6. Провести тестування функціональності web-застосунку.

3

АНАЛІЗ АНАЛОГІВ

	Knigoholic	Goodreads	LibraryThing	Readit
Створення особистої бібліотеки	+	+	+	+
Перегляд	-	+	-	+
Перегляд бібліотек інших користувачів	-	-	-	+
Вбудована соціальна взаємодія	-	-	-	+
Можливість залишати рецензії та оцінки	+	+	+	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги:

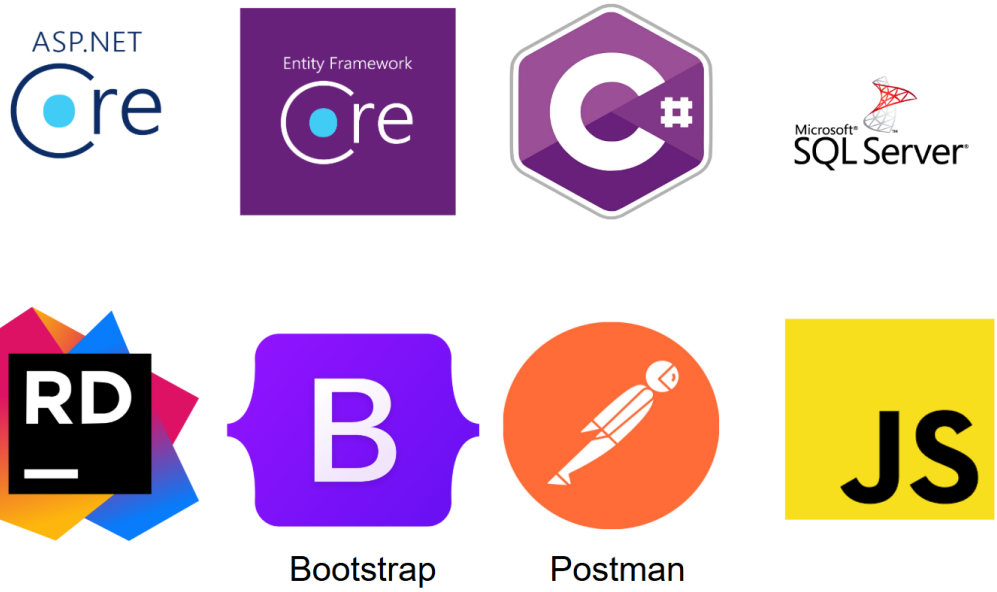
- Перегляд, створення, редагування та видалення книг у власній бібліотеці.
- Додавання друзів та перегляд їхніх бібліотек.
- Оцінювання та рецензування книг.
- Реєстрація та авторизація користувача.
- Встановлення статусу прочитання книги.
- Пошук книг і користувачів.
- Панель адміністратора для управління контентом.

Нефункціональні вимоги:

- Забезпечити кросбраузерну сумісність.
- Реалізувати систему резервного копіювання бази даних.
- Оптимізувати швидкодію системи, мінімізувавши час відповіді при завантаженні сторінок до 2 секунд.

5

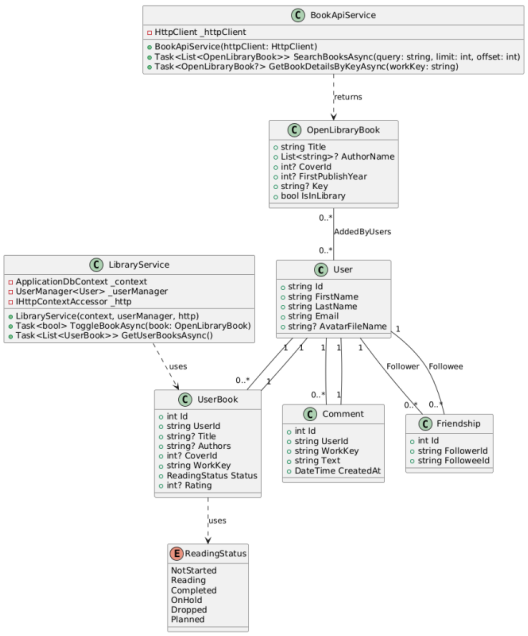
ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ



Bootstrap Postman

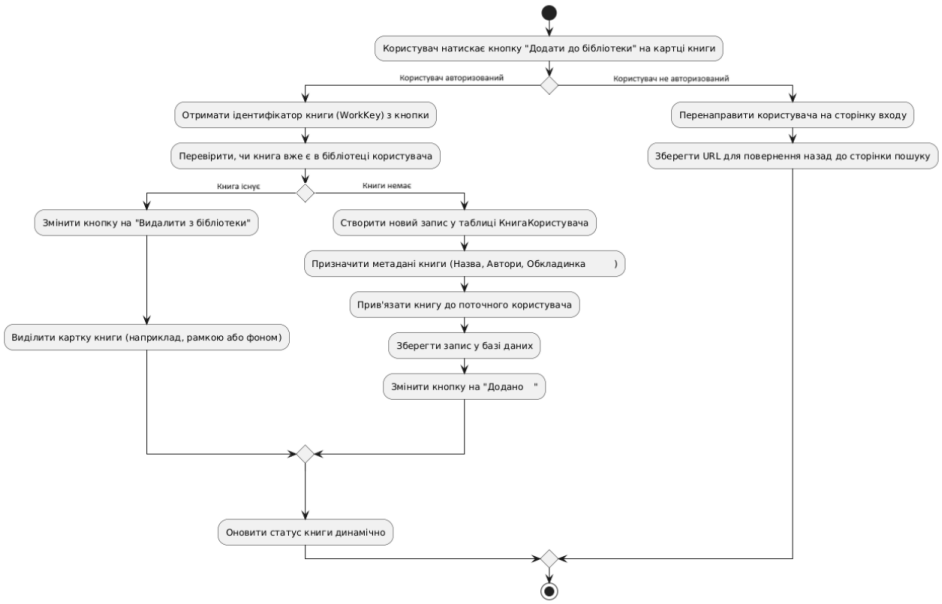
6

Діаграма класів



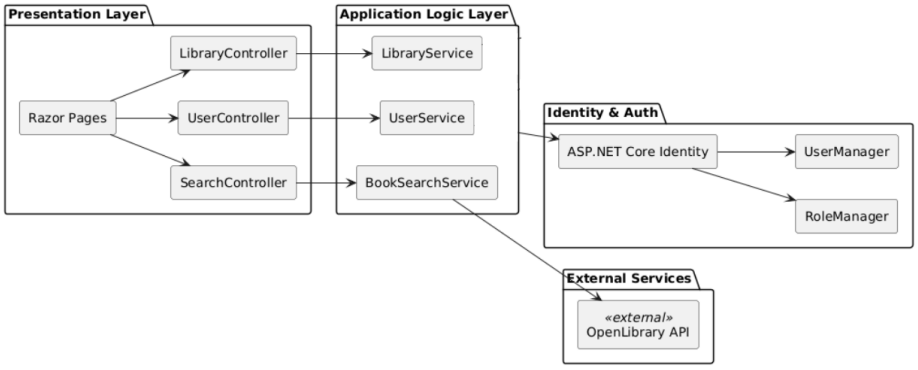
7

Діаграма діяльності



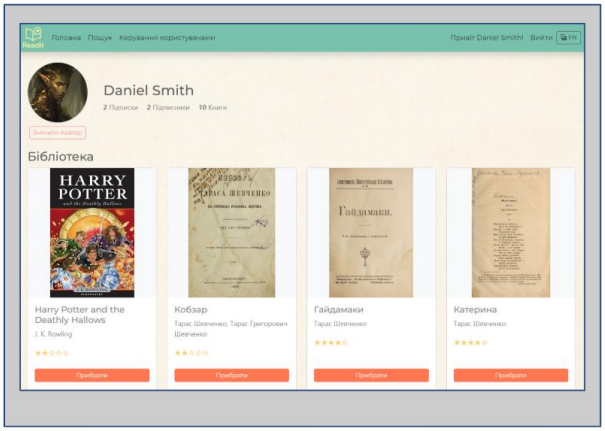
8

Діаграма компонентів

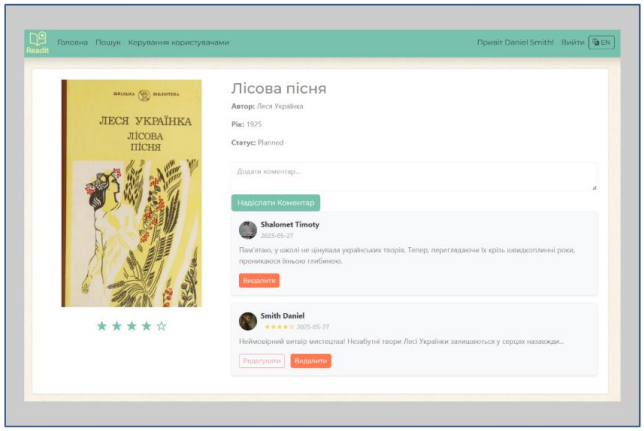


9

ЕКРАННІ ФОРМИ



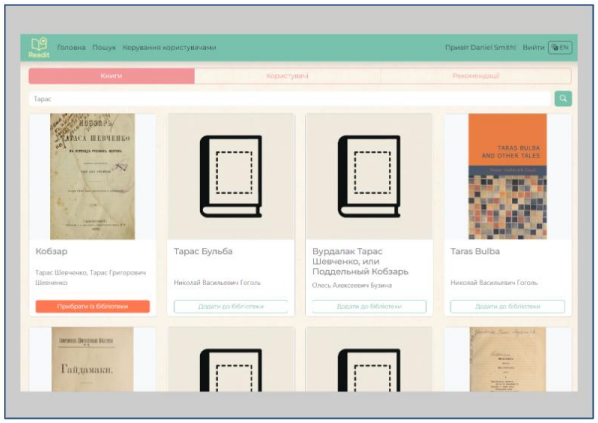
Персональний кабінет



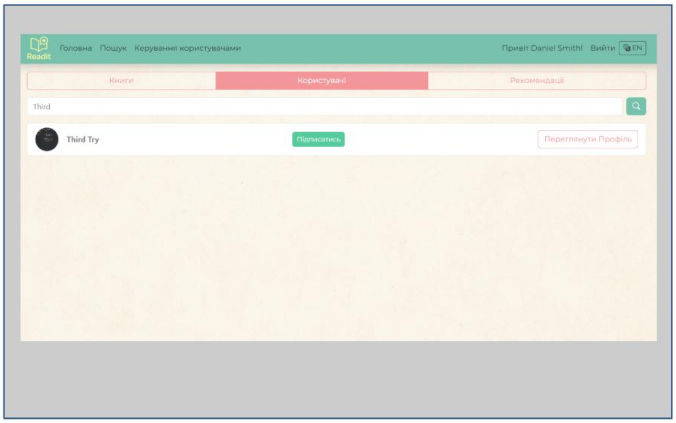
Деталі книги

10

ЕКРАННІ ФОРМИ



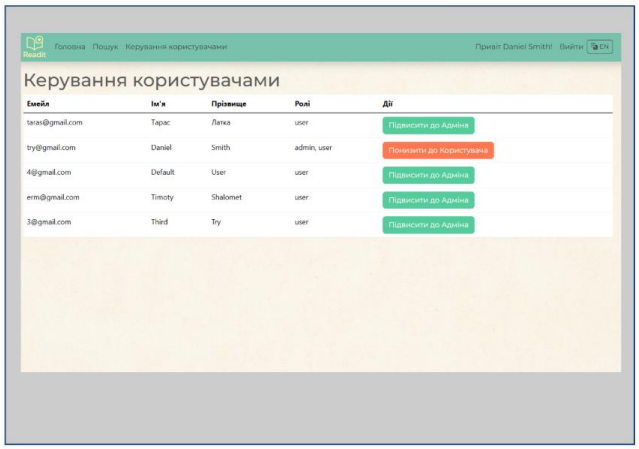
Пошук книг



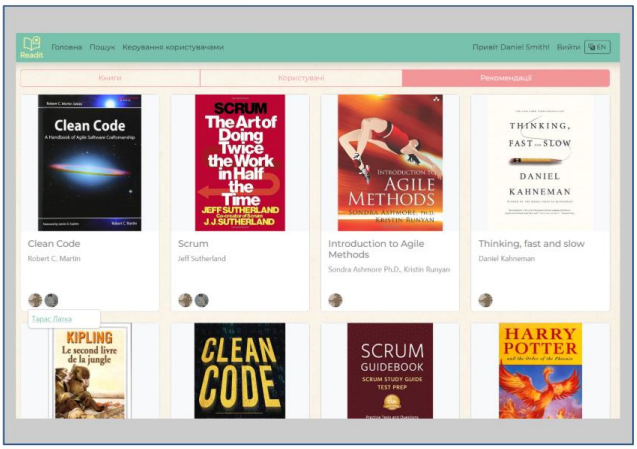
Пошук користувачів

11

ЕКРАННІ ФОРМИ



Панель адміністратора



Рекомендації

12

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Колосенко Т.Є., Золотухіна О.А.. Технологічне забезпечення цифрового сервісу обміну книгами: обґрунтування вибору платформи .NET. VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». 24.04.2025, Київ, Державний університет інформаційно-комунікаційних технологій Збірник тез. К. ДУІКТ, 2025. С.101-102
2. Колосенко Т.Є., Золотухіна О.А. Персоналізація користувацького досвіду у веб-застосунках: практичні рішення для книжкових платформ. V Всеукраїнська науково-практична конференція «Сучасні інтелектуальні інформаційні технології в науці та освіті. 16.05.2025, Київ, Державний університет інформаційно-комунікаційних технологій. (Подано до друку)

13

ВИСНОВКИ

1. Проведено аналіз предметної області, що охоплює цифрові бібліотеки та сервіси з рекомендаціями і соціальною взаємодією користувачів.
2. Досліджено особливості існуючих рішень та визначено потребу у створенні web-застосунку для організації особистих бібліотек із рекомендаціями за жанрами. Були виявлені основні недоліки існуючих рішень: обменежна соціальна взаємодія, відсутність персоналізації власної бібліотеки.
3. Сформовано функціональні та нефункціональні вимоги до системи, з урахуванням потреб кінцевого користувача.
4. Розроблено архітектуру клієнт-серверного web-застосунку із розмежуванням доступу та підтримкою ролей. Для реалізації автентифікації та авторизації було використано ASP.NET Core Identity, зберігання ролей і користувачів організовано у SQL Server.
5. Реалізацію функціоналу особистої бібліотеки, статусів книг, оцінювання, додавання друзів та перегляду чужих бібліотек виконано з використанням мови C# для серверної логіки, Razor Pages для побудови динамічного інтерфейсу, Entity Framework Core для взаємодії з базою даних, а також JavaScript для реалізації інтерактивних елементів на клієнтській стороні. Вся логіка побудована у межах фреймворку ASP.NET Core із застосуванням шаблонів Repository та Unit of Work.
6. Проведено тестування основного функціоналу, підтверджено відповідність вимогам.

14

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

Основна бізнес-логіка застосунку

```
@
@page
@inject
Microsoft.Extensions.Localization.IStringLocalizer<Readit.Pages
.Admin.ManageUsersLocalizer> Localizer
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Mvc.TagHelpers
@inject UserManager<User> UserManager
@inject RoleManager<IdentityRole> RoleManager
@model Readit.Pages.Admin.ManageUsersModel

@attribute [Authorize(Roles = "admin")]

<h1>@Localizer["Manage Users"]</h1>

<table class="table">
   <thead>
   <tr>
      <th>@Localizer["Email"]</th>
      <th>@Localizer["First Name"]</th>
      <th>@Localizer["Last Name"]</th>
      <th>@Localizer["Roles"]</th>
      <th>@Localizer["Actions"]</th>
   </tr>
   </thead>
   <tbody>
   @foreach (var user in Model.Users)
   {
      <tr>
         <td>@user.Email</td>
         <td>@user.FirstName</td>
         <td>@user.LastName</td>
         <td>@string.Join(", ", await
UserManager.GetRolesAsync(user))</td>
         <td>
            @if (!await UserManager.IsInRoleAsync(user,
"admin"))
            {
               <form method="post" asp-page-
handler="Promote" asp-route-id="@user.Id">
                  <button class="btn btn-success"
type="submit">@Localizer["Promote to Admin"]</button>
               </form>
            }
            else
            {
               <form method="post" asp-page-handler="Demote"
asp-route-id="@user.Id">
                  <button class="btn btn-danger"
type="submit">@Localizer["Demote to User"]</button>
               </form>
            }
         </td>
      </tr>
   }
   </tbody>
</table>
@page "{workKey?}"
@inject
Microsoft.Extensions.Localization.IStringLocalizer<Readit.Pages
.BookDetailsLocalizer> Localizer
@model Readit.Pages.BookDetails
@using System.ComponentModel.DataAnnotations
@using System.Reflection
@using Microsoft.AspNetCore.Mvc.TagHelpers

@functions {
   string GetDisplayName(Enum value)
   {
      return value.GetType()
         .GetMember(value.ToString())
         .First()
         .GetCustomAttribute<DisplayAttribute>()?.GetName() ??
value.ToString();
   }
}
@{
   ViewData["Title"] = "Book Details";
}

<div class="container mt-4">
<div class="card p-4">
   <div class="row">
      <div class="col-md-4 text-center">
         @if (Model.Book.CoverId.HasValue)
         {
            <img
src="https://covers.openlibrary.org/b/id/@(Model.Book.CoverId)
-L.jpg" class="img-fluid mb-3" />
         }
         else
         {
            <img src="~/images/NoCover.png"
               class="img-fluid" style="max-height: 100%; max-
width: 100%;" alt="No Cover" />
         }
         <form method="post" asp-page-handler="Rate"
class="d-inline">
            <input type="hidden" name="WorkKey"
value="@Model.WorkKey" />
            <div id="rating-stars" class="mb-3">
               @for (int i = 1; i <= 5; i++)
               {
                  <button type="submit" name="SelectedRating"
value="@i"
                     class="btn btn-link p-0"
                     style="font-size: 2rem; text-decoration:
none;">
                     @(Model.UserRating.HasValue &&
Model.UserRating >= i ? "★" : "☆")
                  </button>
               }
            </div>
         </form>

      </div>

      <div class="col-md-8">
```

```
        <h2>@Model.Book.Title</h2>
        <p><strong>@Localizer["Author"]:</strong>
@string.Join(", ", Model.Book.AuthorName ?? new List<string>
{ "Unknown" })</p>
        <p><strong>@Localizer["Year"]:</strong>
@Model.Book.FirstPublishYear</p>

        <div class="d-flex gap-2 mb-3">
            @if (Model.UserBookStatus != null)
            {
                <p><strong>@Localizer["Status"]:</strong>
@GetDisplayName(Model.UserBookStatus.Value)</p>
            }
        </div>

        <form method="post" asp-page-
handler="AddComment">
            <input type="hidden" asp-for="WorkKey" />
            <textarea class="form-control" asp-
for="NewCommentText" placeholder="@Localizer["Add a
comment..."]"></textarea>
            <button type="submit" class="btn btn-primary mt-
2">@Localizer["Post Comment"]</button>
        </form>


        @foreach (var comment in Model.Comments)
        {
            <div class="border p-3 mb-3 rounded shadow-sm bg-
light">
                <div class="d-flex align-items-center mb-2">
                    <a asp-page="/UserPage" asp-route-
id="@comment.User.Id">
                        <img src="@(comment.User.AvatarFileName
!= null
                                    ?
$"/avatars/{comment.User.AvatarFileName}"
                                    : "https://via.placeholder.com/40")"
                             class="rounded-circle me-2"
                             style="width: 40px; height: 40px; object-fit:
cover;" />
                    </a>

                    <div>
                        <a asp-page="/UserPage" asp-route-
id="@comment.User.Id" class="text-decoration-none text-dark
fw-bold">
                            @comment.User.LastName
@comment.User.FirstName
                        </a><br />
                        @if
(Model.CommenterRatings.TryGetValue(comment.UserId, out
var rating) && rating.HasValue)
                        {
                            <span class="text-warning ms-2" title="Rated
@rating.Value/5">
                                @for (int i = 1; i <= 5; i++)
                                {
                                    <text>@(i <= rating ? "★" : "☆")</text>
                                }
                            </span>
                        }

                        <small class="text-
muted">@comment.CreatedAt.ToString("yyyy-MM-
dd")</small>
                    </div>
                </div>


                @if (Model.EditCommentId == comment.Id)
                {
                    <form method="post" asp-page-
handler="EditComment">
                        <input type="hidden" asp-for="WorkKey" />
                        <input type="hidden" asp-
for="EditCommentId" />
                        <textarea class="form-control" asp-
for="EditedCommentText">@comment.Text</textarea>
                        <button type="submit" class="btn btn-primary
mt-2">@Localizer["Save"]</button>
                        <a href="@Url.Page("BookDetails", new {
workKey = Model.WorkKey })" class="btn btn-secondary mt-2
ms-2">@Localizer["Cancel"]</a>
                    </form>
                }
                else
                {
                    <p>@comment.Text</p>
                }


                @if (User.Identity?.IsAuthenticated == true)
                {
                    var currentUserId =
User.FindFirst(System.Security.Claims.ClaimTypes.NameIdenti
fier)?.Value;
                    var isAuthor = comment.UserId == currentUserId;
                    var isAdmin = User.IsInRole("admin");

                    @if (isAuthor || isAdmin)
                    {
                        // Hide Edit/Delete buttons when this comment
is in edit mode
                        if (Model.EditCommentId != comment.Id)
                        {
                            <div class="mt-2">
                                @if (isAuthor)
                                {
                                    <form method="post" asp-page-
handler="EditCommentForm" asp-route-id="@comment.Id"
class="d-inline">
                                        <button type="submit" class="btn
btn-sm btn-outline-secondary me-
2">@Localizer["Edit"]</button>
                                    </form>
                                }
                                <form method="post"
                                      asp-page-handler="DeleteComment"
                                      asp-route-id="@comment.Id"
                                      class="d-inline"
                                      onsubmit="return confirm('Are you
sure you want to delete this comment?');">
                                    <button class="btn btn-sm btn-danger"
type="submit">@Localizer["Delete"]</button>
                                </form>

                            </div>
```

```
                    }
                }

            }
        </div>
    }
</div>
</div>
</div>
using System.Text.Json.Serialization;
using Readit.Models;

namespace Readit.Api.Models;

public class OpenLibraryBook
{
    [JsonPropertyName("title")]
    public string Title { get; set; }

    [JsonPropertyName("author_name")]
    public List<string>? AuthorName { get; set; }

    [JsonPropertyName("cover_i")]
    public int? CoverId { get; set; }
    public List<User>? AddedByUsers { get; set; } = new();


    [JsonPropertyName("first_publish_year")]
    public int? FirstPublishYear { get; set; }
    [JsonPropertyName("key")]
    public string? Key { get; set; }
    [JsonIgnore]
    public bool IsInLibrary { get; set; }
}
using System.Text.Json.Serialization;

namespace Readit.Api.Models;

public class OpenLibrarySearchResult
{
    [JsonPropertyName("docs")] public List<OpenLibraryBook>
Docs { get; set; } = new();
}
using Readit.Api.Models;

namespace Readit.Services;
using System.Net.Http;
using System.Text.Json;

public class BookApiService
{
    private readonly HttpClient _httpClient;

    public BookApiService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public async Task<List<OpenLibraryBook>>
SearchBooksAsync(string query, int limit = 12, int offset = 0)
    {
        var url =
$"https://openlibrary.org/search.json?q={query}&limit={limit}&
offset={offset}";
        var response = await _httpClient.GetStringAsync(url);
        var searchResult =
JsonSerializer.Deserialize<OpenLibrarySearchResult>(response);
        return searchResult?.Docs ?? new
List<OpenLibraryBook>();
    }
    public async Task<OpenLibraryBook?>
GetBookDetailsByKeyAsync(string workKey)
    {
        if (string.IsNullOrWhiteSpace(workKey))
            return null;

        var searchUrl =
$"https://openlibrary.org/search.json?q={workKey}";
        var searchResponse = await
_httpClient.GetAsync(searchUrl);
        if (!searchResponse.IsSuccessStatusCode)
            return null;

        var searchJson = await
searchResponse.Content.ReadAsStringAsync();
        var searchData =
JsonSerializer.Deserialize<OpenLibrarySearchResult>(searchJso
n);

        var book = searchData?.Docs?.FirstOrDefault(b =>
b.Key?.EndsWith(workKey) == true);
        return book;
    }


}
using Microsoft.EntityFrameworkCore;
using Readit.DataAccess;
using Readit.Models;
using Xunit;

namespace Readit.Test.Integration;

public class LibraryIntegrationTest
{
    [Fact]
    public async Task
AddBook_ThenFetchBooks_WorksWithRealDb()
    {
        var options = new
DbContextOptionsBuilder<ApplicationDbContext>()
            .UseInMemoryDatabase("IntegrationTestDb")
            .Options;

        var context = new ApplicationDbContext(options);

        var user = new User { Id = "userX", UserName =
"integrationuser" };
        context.Users.Add(user);
        await context.SaveChangesAsync();

        context.UserBooks.Add(new UserBook
        {
            Title = "Integration Book",
            UserId = user.Id,
            WorkKey = "OL987"
        });
        await context.SaveChangesAsync();
```

```
        var books = context.UserBooks.Where(b => b.UserId ==
user.Id).ToList();

        Assert.Single(books);
        Assert.Equal("Integration Book", books[0].Title);
    }

}
using System.Net;
using System.Net.Http;
using System.Text;
using System.Text.Json;
using Moq;
using Moq.Protected;
using Readit.Api.Models;
using Readit.Services;
using Xunit;

public class BookApiServiceTests
{
    private static HttpClient CreateMockHttpClient(string
responseContent, HttpStatusCode statusCode =
HttpStatusCode.OK)
    {
        var handlerMock = new Mock<HttpMessageHandler>();

        handlerMock.Protected()
            .Setup<Task<HttpResponseMessage>>(
                "SendAsync",
                ItExpr.IsAny<HttpRequestMessage>(),
                ItExpr.IsAny<CancellationToken>()
            )
            .ReturnsAsync(new HttpResponseMessage
            {
                StatusCode = statusCode,
                Content = new StringContent(responseContent,
Encoding.UTF8, "application/json")
            });

        return new HttpClient(handlerMock.Object)
        {
            BaseAddress = new Uri("https://openlibrary.org/") // used
for formatting but doesn't send real request
        };
    }

    [Fact]
    public async Task
SearchBooksAsync_ReturnsBooks_WhenValidJson()
    {
        // Arrange
        var mockJson = JsonSerializer.Serialize(new
OpenLibrarySearchResult
        {
            Docs = new List<OpenLibraryBook>
            {
                new OpenLibraryBook { Title = "Test Book 1", Key =
"OL1" },
                new OpenLibraryBook { Title = "Test Book 2", Key =
"OL2" }
            }
        });

        var httpClient = CreateMockHttpClient(mockJson);
        var service = new BookApiService(httpClient);

        // Act
        var result = await service.SearchBooksAsync("test");

        // Assert
        Assert.Equal(2, result.Count);
        Assert.Equal("Test Book 1", result[0].Title);
    }

    [Fact]
    public async Task
GetBookDetailsByKeyAsync_ReturnsCorrectBook_WhenMatch
Found()
    {
        // Arrange
        var mockJson = JsonSerializer.Serialize(new
OpenLibrarySearchResult
        {
            Docs = new List<OpenLibraryBook>
            {
                new OpenLibraryBook { Title = "Matched Book", Key
= "/works/OL123" },
                new OpenLibraryBook { Title = "Other Book", Key =
"/works/OL456" }
            }
        });

        var httpClient = CreateMockHttpClient(mockJson);
        var service = new BookApiService(httpClient);

        // Act
        var result = await
service.GetBookDetailsByKeyAsync("OL123");

        // Assert
        Assert.NotNull(result);
        Assert.Equal("Matched Book", result!.Title);
    }

    [Fact]
    public async Task
GetBookDetailsByKeyAsync_ReturnsNull_WhenKeyIsEmpty()
    {
        var httpClient = CreateMockHttpClient("{}",
HttpStatusCode.OK);
        var service = new BookApiService(httpClient);

        var result = await service.GetBookDetailsByKeyAsync("");

        Assert.Null(result);
    }

    [Fact]
    public async Task
GetBookDetailsByKeyAsync_ReturnsNull_WhenNotFound()
    {
        var mockJson = JsonSerializer.Serialize(new
OpenLibrarySearchResult
        {
            Docs = new List<OpenLibraryBook>() // empty list
        });

        var httpClient = CreateMockHttpClient(mockJson);
        var service = new BookApiService(httpClient);
```

```
        var result = await
service.GetBookDetailsByKeyAsync("OL999");

        Assert.Null(result);
    }
}
using System.Security.Claims;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Moq;
using Readit.Api.Models;
using Readit.DataAccess;
using Readit.Library;
using Readit.Models;
using Xunit;

public class LibraryServiceTests
{
    private static LibraryService CreateService(string dbName, out
ApplicationDbContext context)
    {
        var options = new
DbContextOptionsBuilder<ApplicationDbContext>()
            .UseInMemoryDatabase(databaseName: dbName)
            .Options;

        context = new ApplicationDbContext(options);

        var userStoreMock = new Mock<IUserStore<User>>();
        var userManagerMock = new
Mock<UserManager<User>>(userStoreMock.Object, null, null,
null, null, null, null, null, null);

        var testUser = new User { Id = "user123", UserName =
"testuser" };
        userManagerMock.Setup(m =>
m.GetUserAsync(It.IsAny<ClaimsPrincipal>())).ReturnsAsync(te
stUser);

        var httpContext = new DefaultHttpContext();
        httpContext.User = new ClaimsPrincipal(new
ClaimsIdentity(new[]
        {
            new Claim(ClaimTypes.NameIdentifier, testUser.Id),
            new Claim(ClaimTypes.Name, testUser.UserName)
        }));

        var httpAccessor = new Mock<IHttpContextAccessor>();
        httpAccessor.Setup(x =>
x.HttpContext).Returns(httpContext);

        return new LibraryService(context,
userManagerMock.Object, httpAccessor.Object);
    }

    [Fact]
    public async Task
ToggleBookAsync_ShouldAddBook_WhenNotExists()
    {
        var service = CreateService("AddBookDb", out var context);

        var book = new OpenLibraryBook
        {
            Title = "Test Book",
            CoverId = 123,
            Key = "OL123",
            AuthorName = new List<string> { "Author One" }
        };

        var result = await service.ToggleBookAsync(book);

        Assert.True(result);
        Assert.Single(context.UserBooks);
    }

    [Fact]
    public async Task
ToggleBookAsync_ShouldRemoveBook_WhenExists()
    {
        var service = CreateService("RemoveBookDb", out var
context);

        // First add
        await service.ToggleBookAsync(new OpenLibraryBook
        {
            Title = "Test Book",
            Key = "OL123"
        });

        // Then remove
        var result = await service.ToggleBookAsync(new
OpenLibraryBook
        {
            Title = "Test Book",
            Key = "OL123"
        });

        Assert.False(result);
        Assert.Empty(context.UserBooks);
    }
}
```