

**Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення**

**КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»**

**РОЗРОБКА ОНЛАЙН-ПЛАТФОРМИ ДЛЯ ЧИТАННЯ І ЗБЕРІГАННЯ
КОМІКСІВ ВИКОРИСТОВУЮЧИ REACT**

**DEVELOPING AN ONLINE PLATFORM FOR READING AND STORING
COMICS USING REACT**

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-42
Шубер Тарас Ярославович

Керівник:
Яшук Андрій Анатолійович

Кваліфікаційну роботу
допущено до захисту
«__» _____ 20__ р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних та інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Галузь знань: 12 «Інформаційні технології»

Спеціальність: 121 «Інженерія програмного забезпечення»

Освітня програма: «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

«__» _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Шуберу Тарасу Ярославовичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка онлайн-платформи для читання і зберігання коміксів використовуючи React»

Керівник роботи: асистент Ящук А.

затверджені наказом закладу вищої освіти від «31» грудня 2025 р. № 541/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра «29» травня 2026 р.

3. Вихідні дані до роботи: зберігання коміксів; технології розробки вебзастосунків (React, TypeScript, C#, ASP.NET Core); засоби роботи з базами даних (Entity Framework Core, SQL Server); вимоги до сучасних інформаційних систем щодо функціональності, продуктивності.

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): аналіз веб платформ для розповсюдження та читання коміксів і постановка завдання, формування вимог до розроблюваної інформаційної системи, обґрунтування вибору технологій та засобів розробки, практична реалізація об'єкта проектування, розробка структури бази даних, обґрунтування принципів захисту інформаційно-комп'ютерної системи, тестування та налагодження програмного забезпечення.

5. Перелік графічного матеріалу: 20 рисунків, 13 лістингів, 2 додатки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	<i>Ящук А. А.</i>		
Специфікація вимог до розробленої системи	<i>Ящук А. А.</i>		
Розробка об'єкта проектування	<i>Ящук А. А.</i>		
Нормоконтроль	<i>Суринович О. М.</i>		
Гарант ОП	<i>Ліщина Н. М.</i>		
Показник запозичень тексту	<u> </u> %		
Академічна доброчесність	<i>Ящук А. А.</i>		

7. Дата видачі завдання «3» лютого 2026 р.

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 14.02.2026 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 03.03.2026 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 14.03.2026 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 14.04.2026 р.	
5	Практична реалізація об'єкта проектування та розробка бази даних	до 20.04.2026 р.	
6	Тестування та налагодження об'єкта проектування	до 04.05.2026 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедрі	до 29.05.2026 р.	

Здобувач вищої освіти _____

Тарас ШУБЕР

Керівник кваліфікаційної роботи _____

Андрій ЯЩУК

АНОТАЦІЯ

Шубер Т. Я. Розробка онлайн-платформи для читання і зберігання коміксів використовуючи React. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел та двох додатків.

У першому розділі проведено аналіз предметної області онлайн-платформ для читання коміксів, досліджено існуючі вебсервіси, їхні переваги та недоліки, а також обґрунтовано необхідність створення власної інформаційної системи. У другому розділі сформовано функціональні та нефункціональні вимоги до програмного забезпечення, виконано проектування архітектури системи, бази даних та інформаційних потоків, а також обґрунтовано вибір технологій та інструментальних засобів розробки. У третьому розділі описано процес реалізації клієнтської та серверної частин вебзастосунку, розглянуто реалізацію REST API, системи авторизації та захисту інформації, а також наведено результати тестування, налагодження та розгортання програмного забезпечення. У висновках підсумовано результати виконаної роботи та підтверджено досягнення поставленої мети й виконання основних завдань дослідження.

Ключові слова: вебплатформа, комікси, React, TypeScript, ASP.NET Core, REST API, Entity Framework Core, Microsoft SQL Server, інформаційна система.

ABSTRACT

Shuber T. Ya. Developing an online platform for reading and storing comics using React. Manuscript. Bachelor's qualification thesis in the Educational Program "Software Engineering" in the specialty "Software Engineering". Lutsk National Technical University. Lutsk, 2026.

The bachelor's qualification thesis consists of an introduction, three chapters, conclusions, and a list of references.

The first chapter analyzes the subject area of online platforms for reading comics, examines existing web services, their advantages and disadvantages, and substantiates the necessity of developing a custom information system. The second chapter defines the functional and non-functional requirements for the software product, presents the design of the system architecture, database, and information flows, and justifies the choice of technologies and development tools. The third chapter describes the implementation process of the client-side and server-side parts of the web application, considers the implementation of the REST API, authorization and information security systems, and presents the results of testing, debugging, and deployment of the software. The conclusions summarize the results of the completed work and confirm the achievement of the research objectives and the fulfillment of the main tasks of the study.

Keywords: web platform, comics, React, TypeScript, ASP.NET Core, REST API, Entity Framework Core, Microsoft SQL Server, information system.

ЗМІСТ

ВСТУП	7
РОЗДІЛ 1 АНАЛІЗ ВЕБ ПЛАТФОРМ ДЛЯ РОЗПОВСЮДЖЕННЯ ТА ЧИТАННЯ КОМІКСІВ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ	9
1.1 Аналіз сучасного стану проблеми	9
1.2 Постановка завдання на кваліфікаційну роботу бакалавра.	13
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ	15
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення	15
2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання	28
РОЗДІЛ 3 РОЗРОБКА ВЕБ ПЛАТФОРМ ДЛЯ РОЗПОВСЮДЖЕННЯ ТА ЧИТАННЯ ЦИФРОВИХ КОМІКСІВ	42
3.1 Практична реалізація об'єкта проектування	42
3.2 Тестування та налагодження інформаційно-комп'ютерної системи.....	70
3.3 Захист інформаційно-комп'ютерної системи.....	73
3.4 SEO інформаційно-комп'ютерної системи	75
3.5 Розгортання та хостинг інформаційної системи.....	76
ВИСНОВКИ.....	78
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	80
ДОДАТКИ.....	82

ВСТУП

Актуальність теми. У сучасних умовах стрімкого розвитку інформаційних технологій та цифрових медіа значної популярності набувають онлайн-платформи для читання та зберігання графічного контенту, зокрема манги, коміксів і вебтунів. Використання вебтехнологій дозволяє користувачам отримувати швидкий доступ до великої кількості контенту з будь-якого пристрою, а також забезпечує можливість зручного зберігання, пошуку та перегляду творів у цифровому форматі.

Сучасні користувачі очікують від подібних платформ не лише наявності великої бібліотеки контенту, але й зручного інтерфейсу, адаптивного дизайну, швидкої навігації та персоналізованих можливостей взаємодії із системою. До таких можливостей можна віднести створення власних списків, збереження історії читання, систему коментарів, рекомендацій та фільтрації контенту. Водночас значна частина існуючих платформ має ряд недоліків, серед яких перевантажений інтерфейс, обмежені можливості персоналізації, застарілі технології або надмірна кількість рекламного контенту.

Особливо актуальною є проблема створення сучасного адаптивного інтерфейсу, оптимізованого для різних типів пристроїв. Значна частина користувачів переглядає контент зі смартфонів та планшетів, тому вебплатформа повинна забезпечувати коректне відображення сторінок, швидке завантаження зображень та зручну навігацію незалежно від розміру екрана.

Ще одним важливим аспектом є організація зручного зберігання та управління контентом. Онлайн-платформа повинна забезпечувати централізоване збереження інформації про твори, розділи, жанри, користувачів та історію читання, а також підтримувати механізми авторизації та розмежування доступу до адміністративних функцій системи.

Для реалізації подібної системи доцільно використовувати сучасні вебтехнології. Зокрема, бібліотека React дозволяє створювати динамічні та інтерактивні користувацькі інтерфейси на основі компонентного підходу, забезпечуючи високу швидкодію та зручність масштабування проєкту.

Використання React у поєднанні з TypeScript та сучасними серверними технологіями дозволяє створити гнучку та зручну у використанні інформаційну систему .

Отже, тема кваліфікаційної роботи, присвячена розробці онлайн-платформи для читання і зберігання коміксів із використанням React, є актуальною, оскільки спрямована на створення сучасної інформаційної системи для зручного перегляду, зберігання та управління графічним контентом.

Метою кваліфікаційної роботи є розробка онлайн-платформи для читання і зберігання коміксів із використанням бібліотеки React.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати сучасні онлайн-платформи для читання коміксів та манги;
- визначити функціональні та нефункціональні вимоги до системи;
- обґрунтувати вибір технологій розробки;
- спроектувати архітектуру системи та структуру бази даних;
- реалізувати клієнтську та серверну частини вебзастосунку;
- забезпечити механізми авторизації та захисту інформації;
- провести тестування та налагодження програмного забезпечення.

Об'єктом дослідження є процес організації онлайн-перегляду та зберігання графічного контенту.

Предметом дослідження є методи та засоби розробки вебплатформи для читання коміксів із використанням сучасних вебтехнологій.

Практичне значення роботи полягає у створенні сучасної вебплатформи, яка забезпечує зручне читання, зберігання та управління коміксами, а також може бути використана як основа для подальшого розвитку повноцінного онлайн-сервісу для роботи з графічним контентом.

РОЗДІЛ 1

АНАЛІЗ ВЕБ ПЛАТФОРМ ДЛЯ РОЗПОВСЮДЖЕННЯ ТА ЧИТАННЯ КОМІКСІВ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

У сучасному світі цифрові технології активно впливають на всі сфери людської діяльності, зокрема на індустрію розваг та медіаконтенту. Розвиток інтернет мережі, поява високошвидкісного з'єднання та постійне вдосконалення вебтехнологій сприяли значному зростанню популярності онлайн-сервісів для перегляду мультимедійного контенту. Одним із таких напрямків є онлайн-платформи для читання коміксів, манги та графічних новел, які за останні роки стали невід'ємною частиною цифрової культури та сучасного інформаційного простору.

Сьогодні велика кількість користувачів надає перевагу саме електронному формату читання, оскільки він забезпечує швидкий і зручний доступ до контенту. Онлайн-платформи дозволяють користувачам у будь-який момент переглядати улюблені твори, відкривати нові жанри та авторів, а також організовувати власну бібліотеку коміксів. Такий підхід значно спрощує процес взаємодії з контентом у порівнянні з традиційними друкованими виданнями, які потребують фізичного зберігання та придбання.

Особливої популярності набули вебдодатки, що орієнтовані на персоналізацію користувацького досвіду. Сучасні платформи надають можливість створювати власні акаунти, формувати списки улюблених творів, відстежувати прогрес читання та отримувати рекомендації на основі вподобань користувача. Наявність систем пошуку, сортування за жанрами, рейтингами, популярністю та датою оновлення значно підвищує зручність використання таких сервісів та дозволяє швидко знаходити необхідний контент серед великої кількості доступних матеріалів.

Важливою особливістю сучасних онлайн-платформ є їх доступність на різних типах пристроїв. Завдяки адаптивному дизайну користувачі можуть комфортно читати комікси як на персональних комп'ютерах, так і на планшетах або смартфонах. Це робить цифрові сервіси універсальними та зручними для повсякденного використання. Крім того, вебдодатки не потребують встановлення, оскільки доступ до функціоналу здійснюється безпосередньо через браузер.

У процесі аналізу предметної області було розглянуто існуючі онлайн-платформи для читання коміксів і манги, які є найближчими аналогами розроблюваної системи. Серед основних рішень були проаналізовані такі сервіси, як MangaDex, MangaPlus та manga.in.ua, які відрізняються підходами до організації контенту, функціональністю та моделлю розповсюдження.

Платформа MangaDex (рис. 1.1) є однією з найбільш популярних спільнотних платформ для читання коміксів.

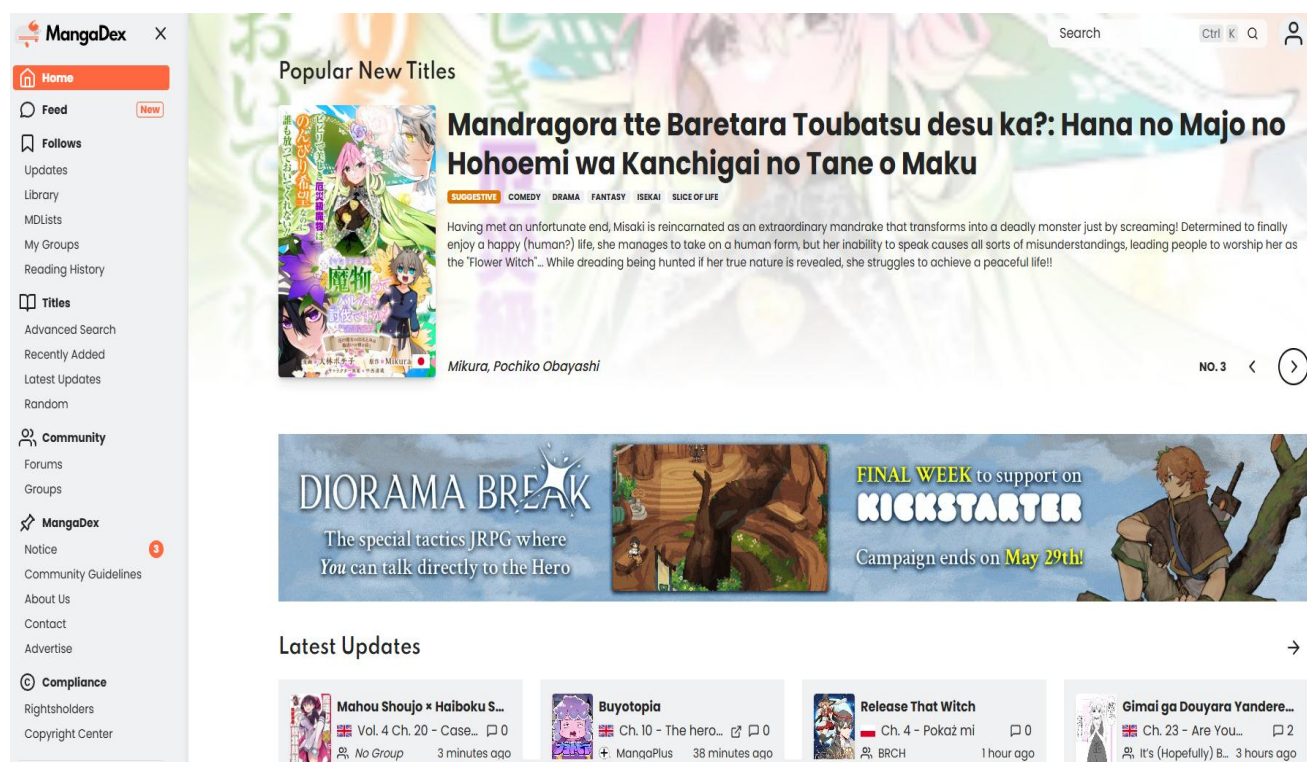


Рисунок 1.1 – Скриншот головної сторінки MangaDex

Вона підтримує велику кількість мов, має розвинену систему фільтрів і категоризації контенту, а також дозволяє користувачам формувати власні бібліотеки та списки читання. Водночас платформа має застарілий та перевантажений інтерфейс. Основною особливістю сервісу є його орієнтація на спільноту та наявність контенту, доданого користувачами й перекладацькими групами, що забезпечує широкий вибір матеріалів.

Перевагою MangaDex також є відсутність надмірної кількості реклами та підтримка великої кількості функцій для читачів, зокрема системи закладок, історії читання та сповіщень про оновлення розділів. Платформа забезпечує швидкий доступ до великої бібліотеки контенту та активно підтримується спільнотою користувачів. Однак через значну кількість функціональних елементів інтерфейс сервісу може бути складним для нових користувачів, а структура сторінок перевантаженою інформацією.

Сервіс MangaPlus є офіційною платформою видавництва Shueisha, що надає доступ до ліцензованого контенту. Його основними перевагами є висока якість перекладу, стабільність роботи та легальність контенту. Однак функціональні можливості платформи є обмеженими у порівнянні зі спільотно орієнтованими сервісами, зокрема щодо персоналізації бібліотеки та гнучкого управління контентом. Крім того, контент обмежений виключно творами видавництва Shueisha, що зменшує загальну різноманітність матеріалів.

Інтерфейс MangaPlus має сучасний мінімалістичний дизайн та добре оптимізований для мобільних пристроїв. Платформа забезпечує швидкий доступ до популярних та актуальних тайтлів, а також підтримує систему рейтингів і категорій для навігації контентом.

Приклад сторінки рейтингу платформи наведено на рисунку 1.2. Водночас сервіс орієнтований переважно на офіційне читання нових релізів, тому можливості взаємодії користувачів із контентом є значно обмеженішими, ніж у спільнотних платформ.

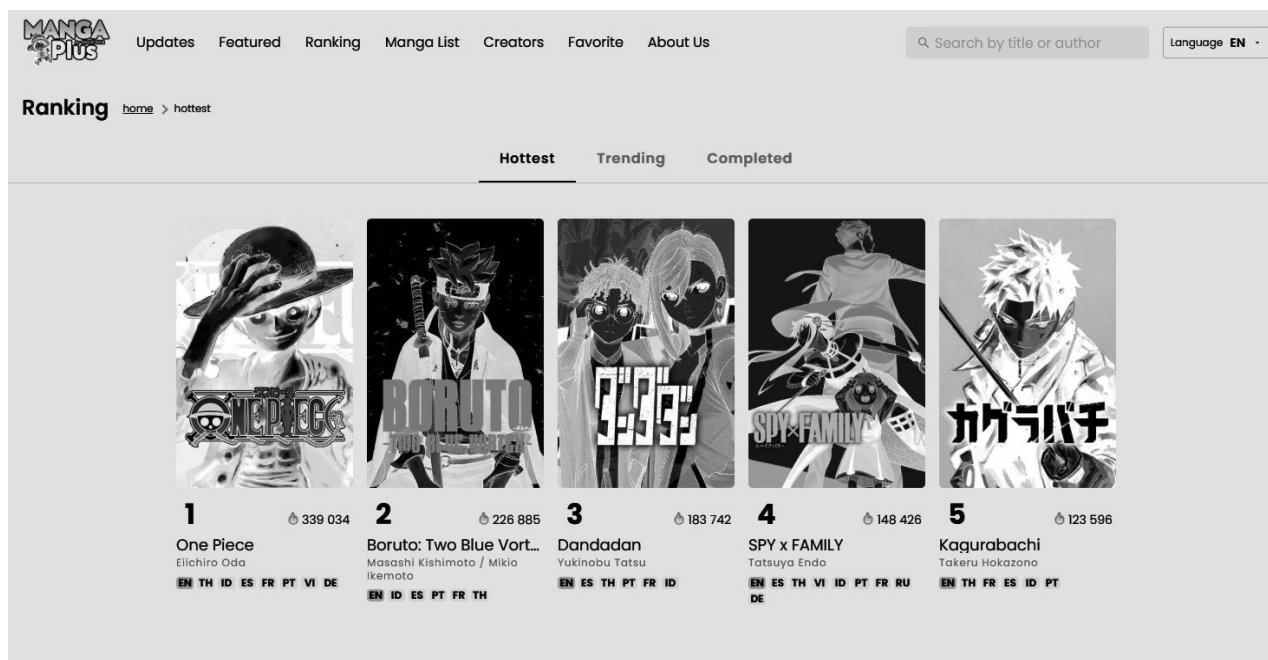


Рисунок 1.2 – Скриншот платформи Manga Plus

Вебсайт manga.in.ua є однією з перших і найбільших українських платформ для публікації коміксів, орієнтованою на локальну аудиторію та перекладені українською мовою мангу і комікси. Платформа забезпечує базовий функціонал перегляду та пошуку контенту, однак має обмежені можливості щодо персоналізації, взаємодії користувачів.

Однак інтерфейс платформи є достатньо функціональним та містить систему категорій, фільтрів і блоків із популярними та новими релізами. Також сайт має адаптацію для мобільних пристроїв та підтримує зручний перегляд контенту на смартфонах. Приклад мобільної версії платформи manga.in.ua наведено на рисунку 1.3. Також вона має певні архітектурні проблеми, оскільки була створена у 2013 році на основі російської CMS DataLife Engine (DLE) [1]. Після початку повномасштабного вторгнення стало неможливим отримання оновлень як з технічної, так і з етичної точки зору через санкційні обмеження. Було здійснено кілька спроб міграції на інші CMS, однак через масштаб платформи та обмежені ресурси вони не дали успішного результату.

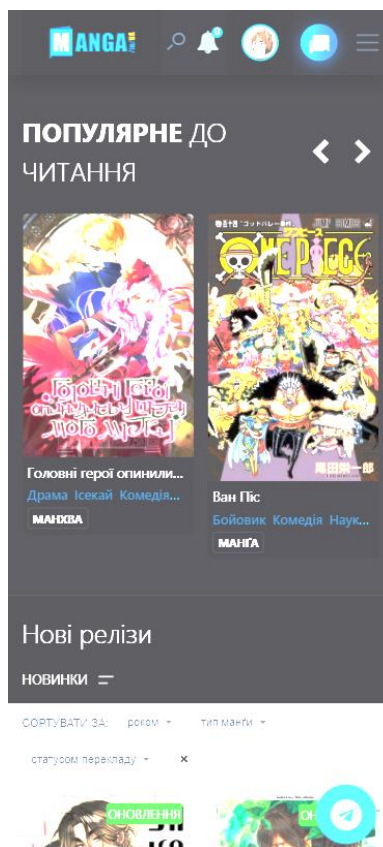


Рисунок 1.3 – Скриншот мобільної версії сайту «manga.in.ua»

Аналіз існуючих аналогів показав, що більшість платформ або орієнтовані на офіційний ліцензований контент з обмеженим функціоналом, або надають широкий вибір матеріалів, але мають застарілий чи перевантажений інтерфейс. Також не всі сервіси забезпечують зручну систему персональної бібліотеки та гнучке управління контентом.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Метою кваліфікаційної роботи є розробка сучасної онлайн-платформи для читання, зберігання та організації цифрових коміксів, яка забезпечує зручну взаємодію користувачів із контентом через вебінтерфейс.

У процесі виконання роботи необхідно дослідити особливості сучасних вебплатформ для читання коміксів, проаналізувати існуючі аналоги та визначити їх переваги й недоліки. На основі проведеного аналізу потрібно спроектувати структуру системи, базу даних, архітектуру клієнтської та серверної частин

застосунку, а також створити сучасний UI/UX-дизайн інтерфейсу для забезпечення комфортної взаємодії користувачів із платформою.

Основними завданнями роботи є:

- проаналізувати сучасні онлайн-платформи для читання коміксів та манги;
- визначити функціональні та нефункціональні вимоги до системи;
- обґрунтувати вибір технологій розробки;
- спроектувати архітектуру системи та структуру бази даних;
- реалізувати клієнтську та серверну частини вебзастосунку;
- забезпечити механізми авторизації та захисту інформації;
- провести тестування та налагодження програмного забезпечення.

Результатом виконання кваліфікаційної роботи має стати функціональна вебплатформа для читання коміксів із сучасним та зручним інтерфейсом, яка забезпечує комфортний доступ до контенту, підтримує персоналізацію користувацького досвіду та має можливість подальшого масштабування й удосконалення.

РОЗДІЛ 2

СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення

Метою розробки онлайн-платформи для читання і зберігання коміксів є створення сучасного вебзастосунку, який забезпечить користувачам зручний доступ до цифрового графічного контенту через мережу Інтернет. Система повинна об'єднати функції перегляду, читання, пошуку та організації коміксів у єдиному інформаційному середовищі, забезпечуючи комфортну взаємодію користувачів із контентом та між собою.

Під час проектування онлайн-платформи було визначено основні функціональні та нефункціональні вимоги, які забезпечують коректну роботу системи, комфортну взаємодію користувачів із програмним забезпеченням та стабільність функціонування вебзастосунку.

Функціональні вимоги визначають основні можливості системи та дії, які повинні виконуватися користувачами й серверною частиною програмного забезпечення.

Розроблювана платформа повинна забезпечувати можливість реєстрації та авторизації користувачів із подальшим збереженням персональних даних та налаштувань профілю. Необхідно реалізувати каталог, з можливістю сортування контенту відповідно до потреб користувача, також передбачити зручний пошук творів за назвою, жанром або іншими параметрами.

Функціональні можливості, доступні неавторизованому користувачу, представлені на діаграмі прецедентів для гостя (рис. 2.1). Дана діаграма демонструє можливість перегляду каталогу коміксів, пошуку творів, перегляду сторінок окремих коміксів, читання контенту, а також переходу до процедур реєстрації та авторизації в системі.

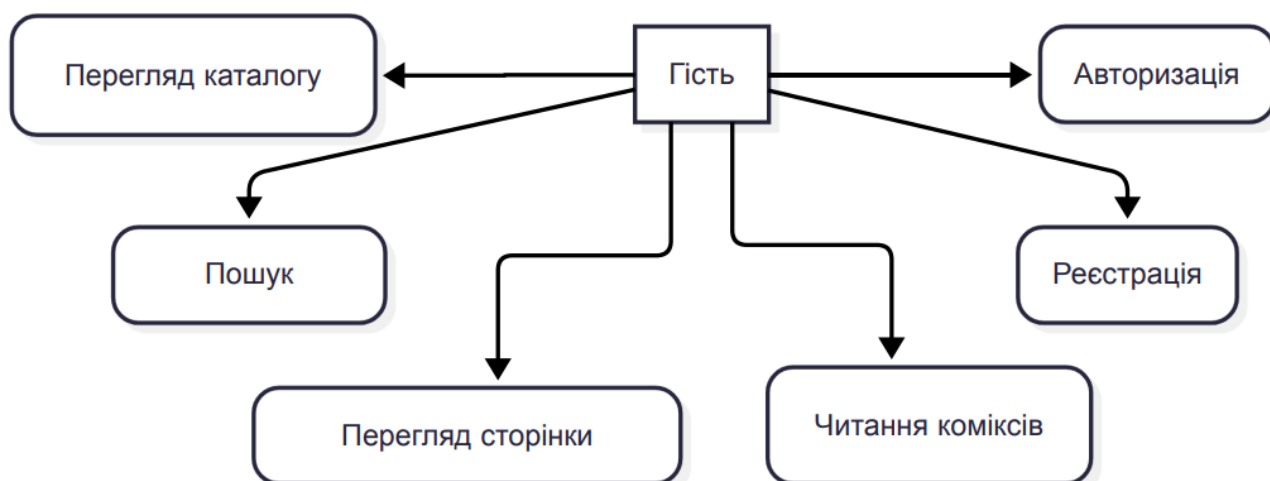


Рисунок 2.1 – Діаграма прецедентів неавторизованого користувача(Гостя)

Однією з ключових функцій системи є забезпечення процесу читання коміксів у зручному режимі перегляду сторінок. Платформа повинна підтримувати завантаження глав та сторінок коміксів, відображення графічного контенту, збереження прогресу читання та швидкий перехід між главами. Для підвищення рівня взаємодії користувачів із контентом система також повинна реалізовувати можливість залишення коментарів, оцінювання та вподобання коментарів інших користувачів.

Функціональні можливості зареєстрованого користувача наведені на діаграмі прецедентів користувача (рис. 2.2).



Рисунок 2.2 – Діаграма прецедентів авторизованого користувача(Користувач)

На діаграмі представлено взаємодію користувача із системою під час читання коміксів, створення персональних списків, додавання або видалення коміксів зі списків, коментування, оцінювання контенту та збереження прогресу читання. Важливою частиною функціоналу є створення персональних списків користувача, зокрема списків улюблених творів, контенту, який читається, або коміксів, запланованих до перегляду. Також система повинна забезпечувати розмежування прав доступу між звичайними користувачами та адміністраторами.

Адміністратор отримує розширені можливості управління платформою, включаючи додавання, редагування та видалення коміксів, завантаження нового контенту, модерацію коментарів та керування інформацією, що зберігається у базі даних.

Адміністративні функції системи представлені на діаграмі прецедентів адміністратора (рис. 2.3). Діаграма відображає процеси керування контентом платформи, зокрема додавання, редагування та видалення коміксів, завантаження й редагування глав, модерацію коментарів та керування користувачами системи.

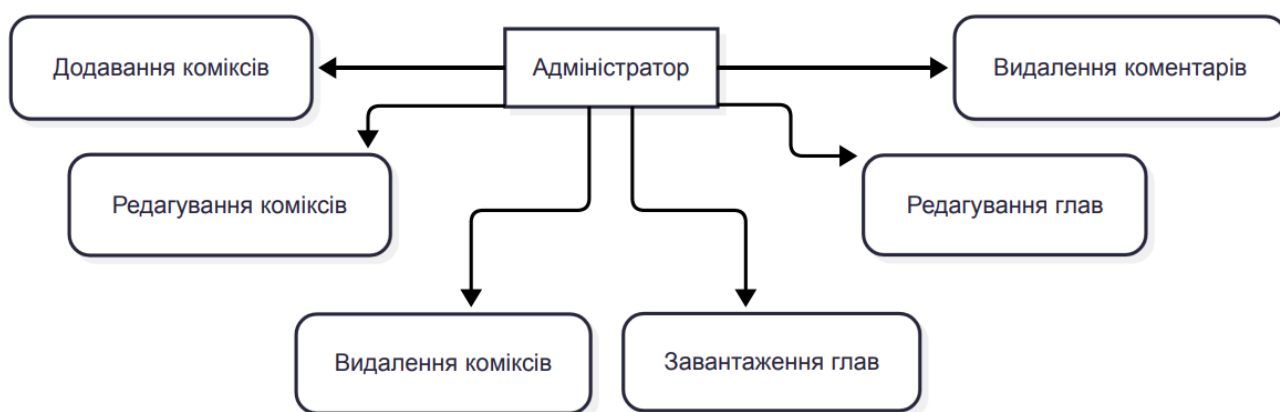


Рисунок 2.3 – Діаграма прецедентів адміністратора

Нефункціональні вимоги визначають характеристики якості програмного забезпечення та умови його експлуатації. Однією з основних вимог є забезпечення високої швидкодії системи, що передбачає швидке завантаження

сторінок, оперативну обробку запитів користувачів та ефективну взаємодію клієнтської й серверної частин застосунку. Система повинна бути стабільною та підтримувати безперервну роботу навіть при одночасній взаємодії значної кількості користувачів.

Особлива увага приділяється безпеці даних та захисту користувацької інформації. Для цього необхідно реалізувати механізми авторизації, перевірки доступу до ресурсів системи, захисту облікових записів та безпечної передачі даних через мережу за протоколом TCP/IP. Важливою вимогою є також забезпечення цілісності даних у базі та контроль коректності введеної інформації.

Оскільки вебплатформа орієнтована на використання на різних типах пристроїв, система повинна мати адаптивний інтерфейс, який коректно відображається на персональних комп'ютерах, планшетах та смартфонах. UX/UI-дизайн має забезпечувати інтуїтивно зрозумілу навігацію, зручне розташування елементів інтерфейсу та комфортний процес взаємодії користувача із системою. Крім цього, програмне забезпечення повинно бути масштабованим, що дозволить у майбутньому розширювати функціональні можливості системи без суттєвих змін її архітектури.

Оскільки система передбачає взаємодію користувача з вебінтерфейсом, обробку запитів на сервері, зберігання даних у базі даних та передачу інформації через мережу Інтернет, то було доцільно взяти за основу клієнт-серверну архітектуру. Клієнт-серверна архітектура дозволяє розділити відповідальність між окремими частинами програмного забезпечення, що спрощує розробку, тестування, підтримку та подальше масштабування системи [2].

Клієнтська частина програмного забезпечення повинна забезпечувати відображення користувацького інтерфейсу, обробку дій користувача, навігацію між сторінками та взаємодію із серверною частиною системи шляхом надсилання відповідних запитів. Інтерфейс має надавати доступ до основних функціональних можливостей платформи, зокрема перегляду каталогу коміксів, сторінок окремих творів, режиму читання, профілю користувача, системи

списків, коментарів та адміністративних функцій. Важливими вимогами до клієнтської частини є підтримка динамічного оновлення інформації, коректне відображення контенту на різних типах пристроїв і забезпечення зручної взаємодії користувача із системою. Для підвищення швидкодії та оптимізації роботи вебзастосунку доцільно використовувати технологію Virtual DOM, який дозволяє мінімізувати кількість прямих змін у структурі DOM браузера та оновлювати лише ті елементи інтерфейсу, що були змінені. Це особливо важливо для системи читання коміксів, де відбувається часте оновлення контенту, перемикання сторінок, завантаження коментарів, оновлення списків користувача та взаємодія з динамічними елементами інтерфейсу.

Архітектура інтерфейсу має передбачати поділ на окремі компоненти та модулі, що дозволяє спростити підтримку програмного забезпечення, повторне використання елементів інтерфейсу та подальше розширення функціональних можливостей системи.

Серверна частина системи повинна забезпечувати обробку HTTP-запитів, реалізацію бізнес-логіки, авторизацію та перевірку прав доступу користувачів, роботу з контентом і взаємодію з базою даних. Backend має приймати запити від клієнтської частини, обробляти їх відповідно до логіки системи, виконувати необхідні операції з даними та повертати результати у форматі, придатному для подальшого відображення в інтерфейсі користувача. Важливими вимогами є централізоване управління даними, перевірка коректності введеної інформації, забезпечення безпечної взаємодії між компонентами системи та підтримка механізмів обміну даними між клієнтською і серверною частинами за допомогою стандартних HTTP-методів.

Крім цього, серверна частина повинна підтримувати стабільну роботу при одночасному обслуговуванні кількох користувачів та забезпечувати можливість подальшого розширення функціональних можливостей системи без суттєвих змін її архітектури. Взаємодія між клієнтською та серверною частинами повинна здійснюватися через стандартизований механізм мережевого обміну даними, що

забезпечує незалежність компонентів системи та можливість їх окремого розвитку і підтримки.

Зберігання інформації в системі повинно забезпечувати централізоване та надійне управління даними користувачів і контенту платформи. База даних має підтримувати збереження інформації про користувачів, комікси, жанри, глави, сторінки, коментарі, оцінки, лайки та користувацькі списки, а також забезпечувати зв'язки між відповідними сутностями системи.

Система збереження даних повинна гарантувати цілісність, актуальність та коректність інформації під час виконання операцій створення, редагування, видалення та пошуку даних. Важливою вимогою є підтримка механізмів контролю доступу до інформації, що дозволяє обмежити прямий доступ користувачів до бази даних та виконувати всі операції із даними через серверну частину системи.

База даних повинна підтримувати ефективну обробку запитів, швидкий пошук інформації та можливість масштабування у разі збільшення кількості користувачів або обсягів контенту. Також система збереження даних має забезпечувати надійність роботи, захист інформації від несанкціонованого доступу та можливість подальшого розширення структури даних без суттєвих змін архітектури програмного забезпечення.

Проектування бази даних є одним із ключових етапів створення вебплатформи, оскільки саме структура даних визначає спосіб зберігання, обробки та взаємодії основних сутностей системи. Для даної системи було спроектовано реляційну базу даних, візуалізація якої представлена у вигляді UML-діаграми бази даних на рисунку 2.4. Створена структура дозволяє чітко організувати інформацію про користувачів, комікси, розділи, сторінки, жанри, користувацькі списки, коментарі, та статистичні показники.

Основною таблицею є Manga, у якій зберігається загальна інформація про твір: назва, автор, художник, рік випуску, статус, видавець, рейтинг, тип твору та вікове обмеження. Для уникнення дублювання даних тип твору та віковий рейтинг винесені в окремі таблиці MangaType і AgeRating, з якими таблиця

Manga пов'язана за допомогою зовнішніх ключів. Такий підхід спрощує підтримку довідкових даних і дозволяє надалі розширювати перелік типів або вікових категорій без зміни основної структури таблиці творів.

Для зберігання жанрів використовується таблиця Genre. Оскільки один твір може належати до кількох жанрів, а один жанр може бути пов'язаний із багатьма творами, між таблицями Manga і Genre реалізовано зв'язок багато-до-багатьох через проміжну таблицю MangaGenre. Це забезпечує гнучку фільтрацію та пошук творів за жанрами.

Структура контенту реалізована через таблиці Chapter і Page. Таблиця Chapter містить інформацію про розділи конкретного твору, зокрема номер, назву та дату публікації. Кожен розділ пов'язаний із відповідним записом у таблиці Manga.

Сторінки розділів зберігаються в таблиці Page, де вказується номер сторінки та шлях до зображення. Завдяки такому поділу система може окремо працювати з творами, їхніми розділами та сторінками, що є зручним для реалізації режиму читання.

Окрема частина бази даних відповідає за користувачів та їхню взаємодію з контентом. Таблиця User містить дані облікового запису користувача, зокрема ім'я, електронну пошту, роль і дату реєстрації. На основі ролі користувача система може розмежовувати доступ до звичайних та адміністративних функцій. Для збереження прогресу читання використовується таблиця ReadingProgress, яка фіксує, на якому розділі та сторінці зупинився користувач, а також дату останнього оновлення прогресу.

Для персоналізації платформи передбачено таблиці UserList і UserListManga. Таблиця UserList зберігає списки, створені користувачем, наприклад «Читаю», «Улюблене» або інші власні категорії. Проміжна таблиця UserListManga пов'язує конкретні твори зі списками користувача та зберігає дату додавання. Така структура дозволяє одному користувачу мати декілька списків, а один твір може одночасно перебувати в різних списках.

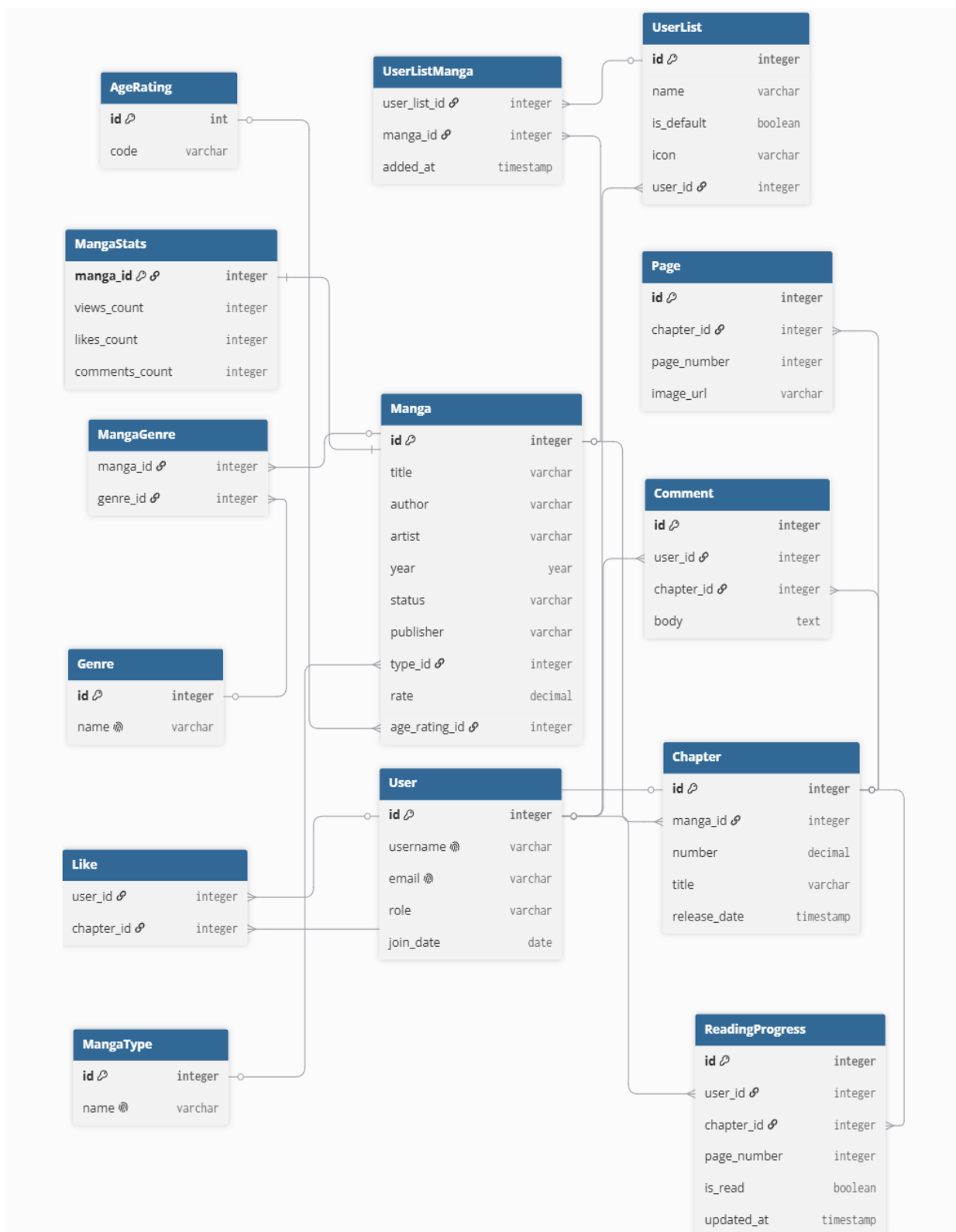


Рисунок 2.4 – Діаграма бази даних

Також база даних підтримує соціальні елементи взаємодії. Таблиця Comment використовується для збереження коментарів користувачів до розділів. Для зберігання узагальнених показників передбачено таблицю MangaStats, у якій

міститься кількість переглядів, і коментарів для конкретного твору. Для візуалізації структури було використано сервіс dbdiagram.io.

Загальна структура системи представлена у вигляді UML-діаграми послідовності на рисунку 2.5. Діаграма демонструє процес взаємодії між основними компонентами вебзастосунку. Користувач взаємодіє з клієнтською частиною системи через вебінтерфейс, виконуючи певні дії. Клієнтська частина надсилає запити до інтерфейсу обміну даними(API), яке забезпечує передачу запитів до серверної частини системи. Серверна частина обробляє отримані запити, виконує необхідну бізнес-логіку та звертається до бази даних для отримання або збереження інформації. Після обробки даних сервер формує відповідь, яка через інтерфейсу обміну даними(API) передається клієнтській частині та відображається користувачу у вебінтерфейсі.

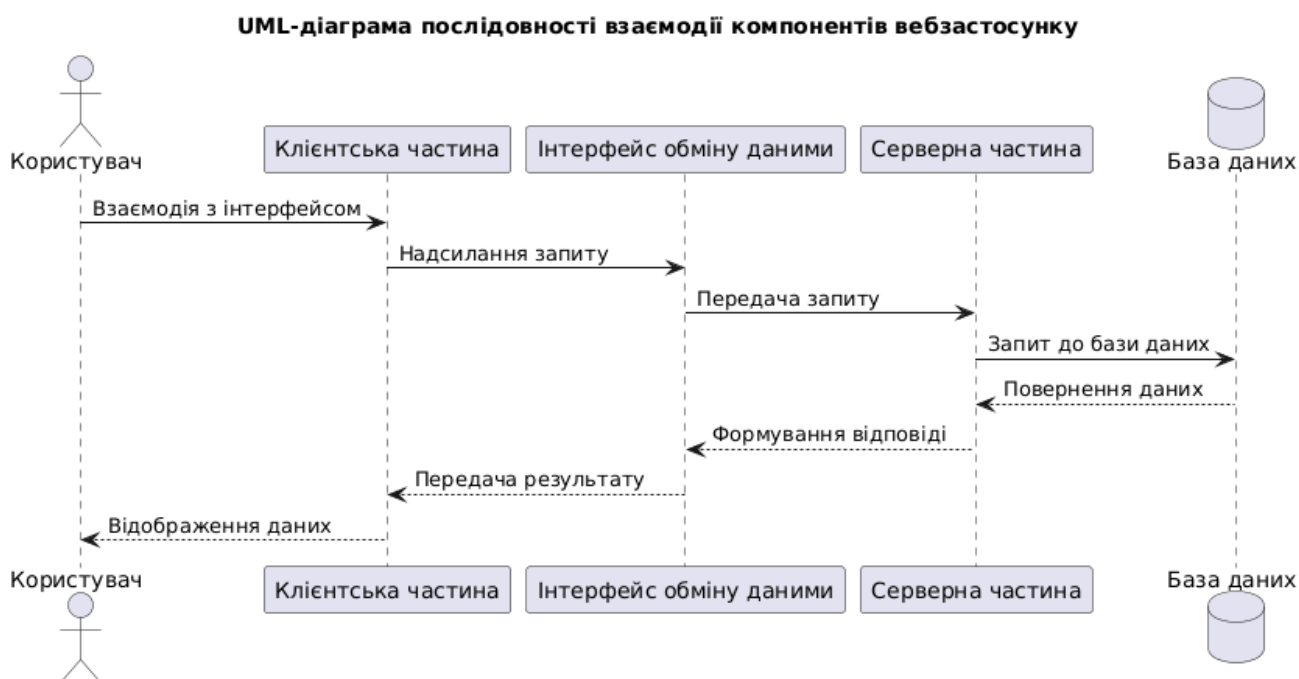


Рисунок 2.5 – Діаграма послідовності взаємодії елементів системи

Обрана архітектура забезпечує логічне розділення системи на окремі рівні, кожен з яких виконує власні функції. Клієнтська частина відповідає за зручність взаємодії користувача з платформою, серверна частина – за обробку даних і бізнес-логіку, а база даних – за надійне зберігання інформації. Такий підхід

дозволяє у майбутньому розширювати функціональні можливості системи, додавати нові модулі, оптимізувати продуктивність та підтримувати стабільну роботу онлайн-платформи.

Під час проектування онлайн-платформи для читання і зберігання коміксів особлива увага приділялася UX/UI-дизайну системи, оскільки зручність користування вебзастосунком безпосередньо впливає на комфорт взаємодії користувача з цифровим контентом. Основною метою проектування інтерфейсу було створення сучасного, інтуїтивно зрозумілого та функціонального середовища, яке забезпечує швидкий доступ до основних можливостей платформи та спрощує навігацію між її елементами.

Під час розробки інтерфейсу було створено макет системи у середовищі Figma, який використовується для візуалізації структури сторінок, розташування елементів інтерфейсу та проектування взаємодії користувача із системою.

Інтерфейс системи побудований відповідно до принципів UX-дизайну, що передбачають логічну структуру сторінок, мінімізацію кількості дій для виконання основних операцій та забезпечення швидкого доступу до ключових функцій системи. Всі сторінки платформи мають єдину стилістику оформлення, спільну навігаційну структуру та однакові принципи взаємодії з елементами інтерфейсу.

У процесі розробки системи було створено сім основних сторінок вебзастосунку: головну сторінку, сторінку каталогу, сторінку окремого твору, сторінку читання коміксів, сторінку завантаження контенту, сторінку редагування контенту та сторінку профілю користувача. Кожна зі сторінок реалізує окремий набір функціональних можливостей та забезпечує виконання відповідних задач користувача.

Головна сторінка системи призначена для ознайомлення користувача з платформою та забезпечує доступ до популярних і нових коміксів. На сторінці каталогу реалізовано функції пошуку, сортування та фільтрації контенту, що дозволяє швидко знаходити необхідні твори за різними параметрами. Сторінка

окремого коміксу містить опис твору, інформацію про жанри, рейтинг, список глав, коментарі та елементи взаємодії з користувацькими списками.

Особлива увага приділялася проектуванню сторінки читання коміксів, оскільки саме вона є основним елементом взаємодії користувача з контентом. Інтерфейс сторінки читання оптимізований для комфортного перегляду графічного контенту, підтримує швидке перемикавання між розділами та мінімізує кількість елементів, що можуть відволікати користувача від процесу читання.

Сторінка профілю користувача забезпечує доступ до персональної інформації, історії читання та користувацьких списків. Для адміністраторів системи передбачено окремі сторінки завантаження та редагування контенту, які забезпечують керування коміксами, розділами та інформацією, що зберігається у системі.

Схема навігації користувача на рисунку 2.6, демонструє основні переходи між сторінками системи та відображає логіку взаємодії користувача з платформою. Центральним елементом є головна сторінка, з якої користувач може перейти до каталогу, сторінки окремого твору або профілю.

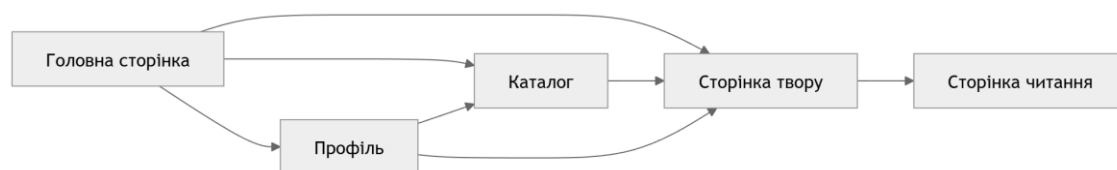


Рисунок 2.6 – Схема навігації користувача

Схема навігації адміністратора (рис. 2.7) відображає процес керування контентом системи та взаємозв'язок між адміністративними сторінками платформи.



Рисунок 2.7 – Схема навігації адміністратора

Важливою частиною проектування системи є аналіз та організація інформаційних потоків, які забезпечують взаємодію між користувачем, клієнтською частиною, серверною частиною та базою даних. Під час роботи із системою користувач через інтерфейс виконує певні дії, після чого клієнтська частина формує запити до серверної частини. Сервер обробляє отримані дані, виконує необхідні операції з базою даних та повертає результат для подальшого відображення в інтерфейсі користувача.

Основними інформаційними потоками системи є процеси авторизації користувачів, отримання інформації про комікси, завантаження сторінок для читання, робота зі списками користувача, коментарями та оцінками, а також керування контентом адміністраторами системи. Під час авторизації користувач передає облікові дані, які перевіряються системою та використовуються для визначення прав доступу. Після успішної авторизації користувач отримує доступ до персоналізованих функцій платформи.

Технологія інформаційного обслуговування системи базується на клієнт-серверній взаємодії та забезпечує доступ користувачів до контенту через вебінтерфейс. Інформаційні потреби користувачів включають швидкий пошук коміксів, зручний доступ до режиму читання, можливість організації власної бібліотеки та взаємодії з іншими користувачами системи. Інтерфейс платформи проектувався з урахуванням цих потреб, що дозволило забезпечити зручність використання та ефективну взаємодію користувача із системою. Для наочного представлення процесу обміну даними між компонентами системи було побудовано діаграму інформаційних потоків (рис. 2.8).

Під час взаємодії з каталогом система обробляє запити на пошук і перегляд коміксів, після чого отримує необхідну інформацію з бази даних та повертає результати користувачу. Аналогічним чином реалізовано режим читання, у якому здійснюється завантаження глав і сторінок коміксів для подальшого відображення в інтерфейсі користувача.

Окремий інформаційний потік пов'язаний із роботою користувачьких списків, коментарів та оцінок. Після створення або оновлення відповідної

інформації система зберігає зміни в базі даних та повертає оновлені результати для подальшого відображення в інтерфейсі.

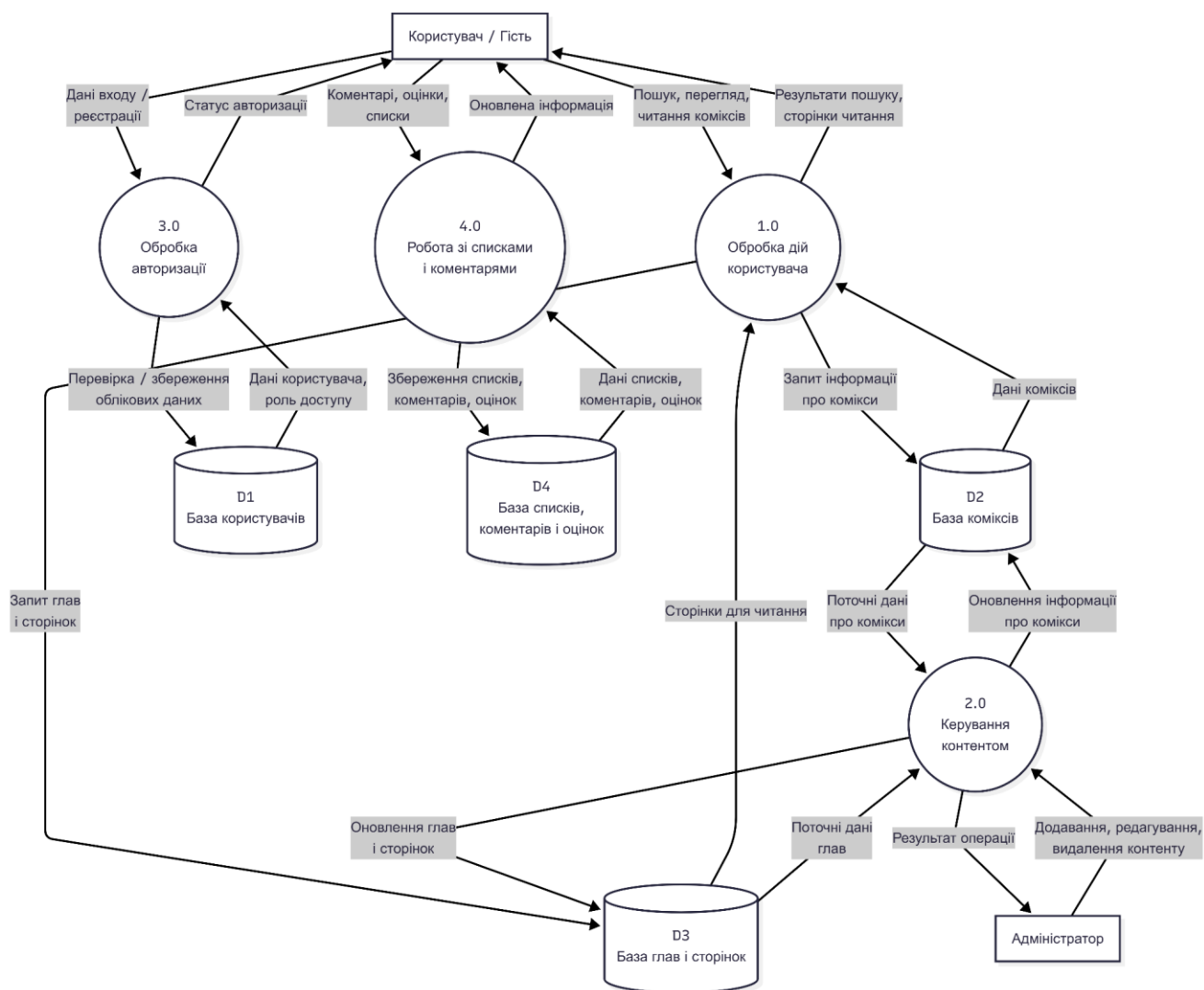


Рисунок 2.8 – Діаграма інформаційних потоків

Для адміністраторів системи передбачено потоки даних, пов'язані з керуванням контентом платформи. Адміністратор може додавати, редагувати та видаляти інформацію про комікси, глави та сторінки, після чого система оновлює відповідні дані та забезпечує їх подальше використання користувачами платформи.

Організація таких інформаційних потоків дозволяє забезпечити централізоване управління даними, підтримку цілісності інформації, контроль

доступу до функціональних можливостей системи та стабільну роботу вебзастосунку при взаємодії різних типів користувачів із платформою.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Готуючись до розробки було розглянуто велику кількість технологій та інструментів, які можуть бути використані у сучасній веб розробці. Спершу було розглянуто frontend-фреймворки, оскільки саме вони відповідають за створення користувацького інтерфейсу, швидкодію веб-застосунку та взаємодію користувача із системою. Серед найбільш популярних рішень були розглянуті Angular, Vue та React.

Angular є повноцінним frontend-фреймворком, розробленим компанією Google. Він містить значну кількість вбудованих інструментів для маршрутизації, роботи з формами, dependency injection та управління станом застосунку. Angular використовує сувору архітектуру та TypeScript як основну мову розробки, що забезпечує структурованість проекту та полегшує підтримку великих систем [3]. Водночас даний фреймворк характеризується високим порогом входження та складнішою структурою проекту. Для реалізації середніх за масштабом веб-застосунків Angular може бути надмірно громіздким, а процес розробки потребує більшої кількості коду та часу.

Vue є більш легким frontend-фреймворком, який орієнтований на простоту використання та швидке освоєння. Він дозволяє створювати інтерактивні інтерфейси з мінімальною кількістю коду та має зрозумілу документацію. Vue забезпечує хорошу продуктивність та зручний механізм роботи з компонентами. Однак екосистема Vue є меншою у порівнянні з React, а кількість доступних бібліотек та готових рішень є більш обмеженою. Крім того, Vue рідше використовується у великих комерційних проектах, що може впливати на масштабованість та підтримку системи у майбутньому.

React є JavaScript-бібліотекою для створення користувацьких інтерфейсів, розробленою компанією Meta. Основною особливістю React є компонентний підхід, при якому інтерфейс поділяється на незалежні багаторазові компоненти. React використовує технологію Virtual DOM, що полягає у створенні віртуальної копії структури DOM браузера у пам'яті застосунку.

На рисунку 2.9 показано принцип роботи Virtual DOM у React. Верхня частина зображення демонструє роботу Virtual DOM, а нижня – реального DOM браузера. Сині елементи позначають компоненти, які не змінилися, а червоні – елементи, стан яких був оновлений.

Після зміни стану компонента (State change) React створює нову версію Virtual DOM та порівнює її з попередньою (Compute diff). Під час цього процесу React визначає лише ті елементи дерева, які були змінені. На схемі видно, що зміни торкаються тільки правої гілки дерева, позначеної червоним кольором, тоді як інші елементи залишаються без змін [4].

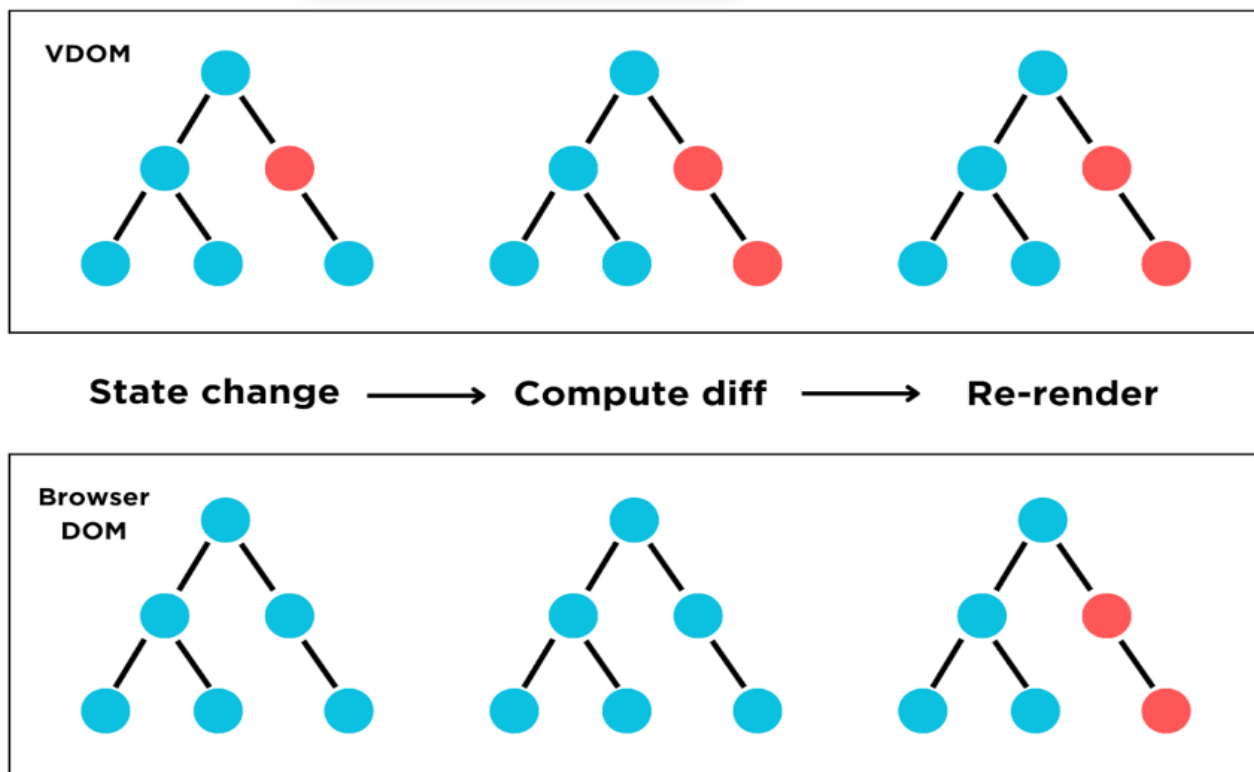


Рисунок 2.9 – Схема роботи Virtual DOM [5]

Після визначення відмінностей React виконує повторне відображення (Render) лише змінених частин інтерфейсу в реальному DOM браузера. У нижній частині схеми видно, що оновлюються тільки елементи, позначені червоним кольором, а всі інші компоненти не перерендерюються. Завдяки цьому зменшується кількість операцій взаємодії з DOM браузера, що позитивно впливає на продуктивність вебзастосунку та забезпечує швидше оновлення інтерфейсу користувача [4].

На відміну від Angular, React є більш гнучким та не нав'язує жорсткої структури проекту, що дозволяє адаптувати архітектуру системи під конкретні потреби. У порівнянні з Vue, React має значно більшу екосистему бібліотек, активнішу спільноту розробників та ширше використання у сучасній веб-розробці. Також React добре інтегрується з TypeScript та іншими сучасними інструментами frontend-розробки [6].

Для реалізації веб-платформи читання коміксів було обрано React, оскільки дана бібліотека найкраще підходить для створення динамічного та інтерактивного користувацького інтерфейсу. Компонентний підхід дозволив розділити систему на окремі незалежні елементи, зокрема картки творів, сторінки читання, систему коментарів та навігаційні компоненти. Додатковим фактором вибору React став наявний попередній досвід роботи з даною бібліотекою, що дозволило швидше освоїти процес розробки, краще зрозуміти принципи компонентного підходу та ефективніше організувати структуру проекту.

У React використовується дві мови програмування JavaScript та TypeScript, які застосовуються для створення логіки роботи компонентів, взаємодії з серверною частиною та побудови динамічного користувацького інтерфейсу. Обидві технології мають широку популярність у сучасній веб-розробці та підтримуються більшістю сучасних інструментів і бібліотек.

JavaScript є основною мовою веб-розробки та використовується для створення інтерактивних веб-застосунків. Його головними перевагами є простота використання, велика кількість доступних бібліотек та швидкість

розробки. JavaScript дозволяє оперативно створювати динамічні інтерфейси та легко інтегрується з React. Однак дана мова є динамічно типізованою, через що частина помилок може бути виявлена лише під час виконання програми [6].

TypeScript є розширенням JavaScript, яке додає підтримку статичної типізації та додаткові інструменти для розробки складних програмних систем. Завдяки використанню типів даних значна частина помилок виявляється ще на етапі написання коду, що підвищує стабільність та надійність веб-застосунку [7]. Крім того, TypeScript покращує читабельність коду, забезпечує зручніше автодоповнення та полегшує підтримку великих проектів із великою кількістю компонентів та API-запитів.

Під час розробки веб-платформи було обрано TypeScript, оскільки дана мова краще підходить для створення масштабованих React-застосунків. Використання типізації значно спростило взаємодію між клієнтською та серверною частинами веб-застосунку, та дозволило запобігти і виявити значну кількість помилок.

Для стилізації веб-застосунків сьогодні використовуються різноманітні технології та фреймворки, серед яких Bootstrap, Material UI, Tailwind CSS та звичайний CSS. Ці засоби дозволяють створювати сучасні та адаптивні користувацькі інтерфейси, однак відрізняються підходом до організації стилів, швидкістю розробки та рівнем гнучкості налаштування.

Звичайний CSS є базовим засобом стилізації веб-сторінок та забезпечує повний контроль над зовнішнім виглядом елементів інтерфейсу. Його головною перевагою є висока гнучкість та відсутність залежності від сторонніх бібліотек чи фреймворків. Використання чистого CSS дозволяє створювати повністю унікальні дизайн-системи та максимально адаптувати інтерфейс під потреби конкретного проекту. Саме тому великі технологічні компанії, такі як Google, Meta, Apple, Netflix та GitHub, часто використовують власні внутрішні дизайн-системи, побудовані переважно на основі CSS та власних компонентних бібліотек. Проте у великих проектах використання звичайного CSS може

ускладнювати підтримку стилів через значну кількість окремих файлів, дублювання коду та можливі конфлікти між класами.

Material UI є бібліотекою компонентів для React, побудованою відповідно до концепції Material Design. Вона містить готові елементи інтерфейсу та добре інтегрується з React-застосунками. Основними перевагами Material UI є велика кількість готових компонентів та зручність створення складних інтерфейсів. Проте використання даної бібліотеки значно обмежує свободу дизайну та часто призводить до перевантаження проекту зайвими стилями та компонентами.

Tailwind CSS є utility-first CSS-фреймворком, тобто фреймворком, який побудований навколо використання невеликих службових класів. Кожен такий клас відповідає за одну конкретну CSS-властивість або невелику групу властивостей [8]. Наприклад, замість створення окремого CSS-класу для кнопки розробник може одразу в JSX-кодi вказати класи для кольору фону, відступів, розміру тексту, заокруглення кутів, тіні та анімації.

Основна ідея підходу utility-first полягає в тому, що стилі описуються не через великі заздалегідь створені CSS-класи, а через набір маленьких універсальних класів. Завдяки цьому розробник може швидко комбінувати готові класи та створювати потрібний дизайн без написання великої кількості окремого CSS-коду. Наприклад у лістингу 2.1 кожен utility-клас виконує окрему функцію. Клас `text-4xl` задає розмір тексту, `md:text-6xl` змінює його для більших екранів, `font-bold` встановлює жирний шрифт, `mb-6` додає зовнішній відступ знизу, а `tracking-tight` змінює міжсимвольний інтервал. Аналогічно в описі манги класи `text-gray-600`, `max-w-lg`, `leading-relaxed` та `line-clamp-4` відповідають за колір тексту, максимальну ширину блоку, міжрядковий інтервал та обмеження кількості рядків.

Такий підхід значно пришвидшує процес розробки, оскільки стилізація елемента виконується безпосередньо в HTML або JSX-кодi компонента. Це також полегшує підтримку інтерфейсу, адже стилі конкретного елемента знаходяться поруч із його розміткою, і розробнику не потрібно постійно переходити між компонентом та окремими CSS-файлами. Крім того, utility-first

підхід зменшує ризик конфліктів між класами, оскільки більшість класів мають чітко визначене призначення та не впливають на інші елементи інтерфейсу.

Лістинг 2.1 Фрагмент коду з компоненту HeroSection.tsx

```
<h1 className="text-4xl md:text-6xl font-bold text-[#1a1a2e] mb-6
leading-[1.1] tracking-tight">
  {featuredManga.title}
</h1>
<p className="text-lg text-gray-600 mb-8 max-w-lg mx-auto md:mx-0
leading-relaxed font-normal line-clamp-4">
  {featuredManga.description}
</p>
```

кінець лістингу 2.1

У порівнянні з Bootstrap та Material UI, Tailwind CSS забезпечує значно більшу гнучкість у створенні власного дизайну інтерфейсу. Розробник не обмежений готовими шаблонами компонентів та може створювати унікальний зовнішній вигляд системи без значного перевизначення стилів. Саме тому для реалізації веб-платформи читання коміксів було обрано Tailwind CSS, оскільки даний фреймворк забезпечує швидку та гнучку стилізацію інтерфейсу. Використання utility-класів дозволило значно пришвидшити процес створення адаптивного дизайну та спростило підтримку стилів у великому проекті.

Таким чином, після визначення основних технологій для клієнтської її частини веб-застосунку, наступним етапом став вибір технології для реалізації серверної частини веб-застосунку. Серверна частина відіграє важливу роль у роботі системи, оскільки саме вона відповідає за обробку HTTP-запитів, взаємодію з базою даних, авторизацію користувачів, управління контентом та передачу даних клієнтській частині через API.

Були розглянуті такі рішення, як ASP.NET Core, Node.js та Django. Дані технології є популярними засобами для створення сучасних веб-застосунків, однак мають різні підходи до організації серверної логіки, взаємодії з базою даних та побудови архітектури системи.

Node.js є серверним середовищем виконання JavaScript, яке дозволяє використовувати одну мову програмування як для frontend-, так і для backend-

розробки. Основною перевагою Node.js є висока швидкість обробки асинхронних операцій та велика кількість доступних бібліотек. Дана технологія добре підходить для створення застосунків реального часу та невеликих веб-сервісів. Водночас при створенні великих проектів структура коду може ставати складнішою для підтримки, а відсутність суворої архітектури часто призводить до меншої організованості серверної частини [9].

Django є популярним backend-фреймворком на мові Python. Він містить значну кількість готових інструментів для роботи з базою даних, авторизацією та адміністративною панеллю [10]. Перевагами Django є швидкість створення веб-застосунків, високий рівень безпеки та зручна робота з ORM. Проте дана технологія більше орієнтована на класичні монолітні веб-застосунки та дещо поступається ASP.NET Core у продуктивності при роботі з високонавантаженими системами.

ASP.NET Core є сучасним кросплатформним фреймворком для створення веб-застосунків та API, розробленим компанією Microsoft. Даний фреймворк забезпечує високу продуктивність, підтримує багаторівневу архітектуру та має вбудовані механізми dependency injection, авторизації та маршрутизації. Однією з головних переваг ASP.NET Core є хороша інтеграція з платформою .NET та ORM Entity Framework Core, що значно спрощує роботу з базою даних та побудову складної серверної логіки [11].

Додатковим фактором на користь ASP.NET Core стала наявність попереднього досвіду з даною платформою, здобутого під час навчання в університеті. Це дозволило швидше виконати процес розробки серверної частини застосунку та ефективніше організувати структуру проекту.

Тому для реалізації серверної частини веб-платформи читання коміксів було обрано ASP.NET Core, яка поєднує високу продуктивність, зручну організацію коду та ефективну взаємодію з базою даних.

Для взаємодії серверної частини з базою даних у проекті було використано ORM Entity Framework Core. Entity Framework Core є сучасною ORM-технологією для платформи .NET, розробленою компанією Microsoft. Даний

інструмент забезпечує зручну інтеграцію з ASP.NET Core та дозволяє значно спростити роботу з базою даних під час розробки веб-застосунків [12].

ORM (Object-Relational Mapping) – це технологія, яка дозволяє працювати з базою даних через об'єкти мови програмування без необхідності постійного написання SQL-запитів вручну. Основна ідея ORM полягає у відображенні таблиць бази даних у вигляді класів, а записів таблиць – у вигляді об'єктів цих класів [13].

Однією з головних переваг Entity Framework Core є можливість працювати з даними через звичайні C#-класи. Наприклад, сутності Manga, Chapter, Comment або User описуються у вигляді моделей, після чого Entity Framework Core автоматично створює відповідні таблиці та зв'язки між ними у базі даних. Це дозволяє працювати з об'єктами та колекціями даних у коді програми замість прямої взаємодії з SQL-запитами [12].

Entity Framework Core також підтримує механізм LINQ (Language Integrated Query), який дозволяє виконувати запити до бази даних за допомогою C#-синтаксису. Замість ручного написання SQL-запитів отримання даних виконується через методи Where, Select, Include та інші LINQ-операції [12]. Такий підхід робить код більш зрозумілим, читабельним та безпечним.

Принцип взаємодії Entity Framework Core із базою даних представлений на рисунку 2.10. Основою роботи ORM є моделі сутностей, створені у вигляді C#-класів. Дані моделі об'єднуються у класі MangaDbContext, який виступає центральним компонентом взаємодії застосунку з базою даних. Під час виконання запитів розробник працює з DbSet-колекціями та LINQ-запитами, після чого Entity Framework Core автоматично генерує відповідні SQL-запити до реляційної бази даних.

Після виконання запитів база даних повертає результати у вигляді наборів записів, які Entity Framework Core автоматично перетворює у C#-об'єкти відповідних моделей. Таким чином, взаємодія серверної частини з базою даних здійснюється через ORM-рівень, що значно спрощує розробку, підтримку та масштабування програмного забезпечення.

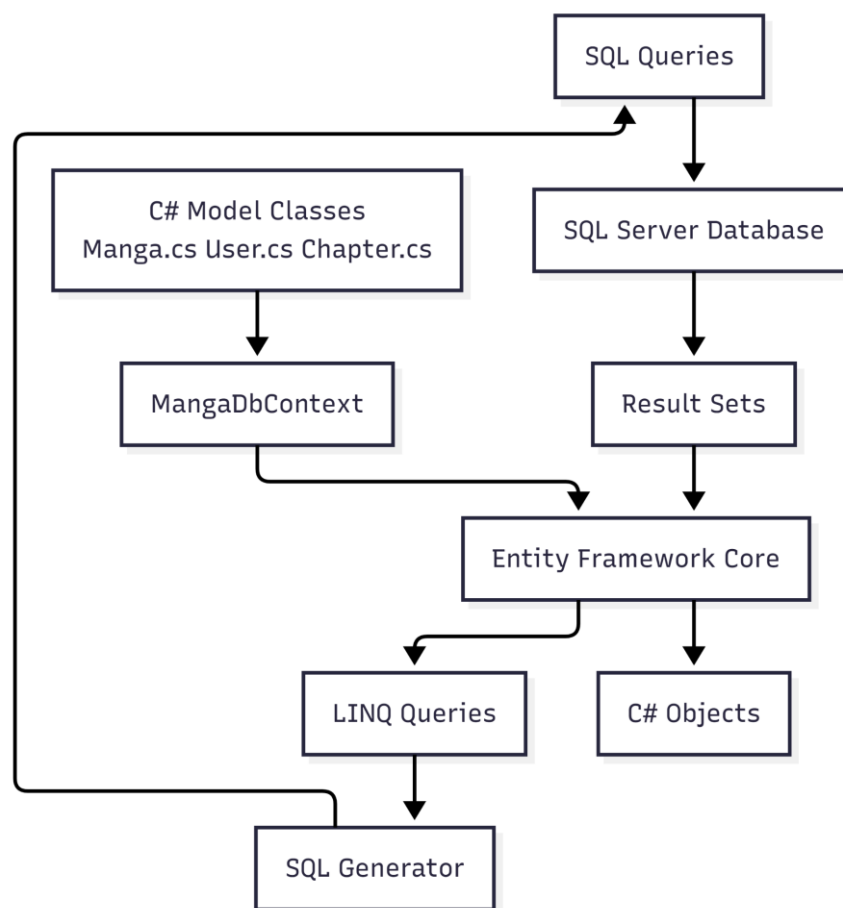


Рисунок 2.10 – Діаграма роботи ORM Entity Framework Core

Важливою перевагою Entity Framework Core є підтримка міграцій. Міграції дозволяють автоматично створювати та оновлювати структуру бази даних відповідно до змін у моделях проекту. Наприклад, при додаванні нової сутності або поля достатньо створити нову міграцію, після чого Entity Framework Core автоматично оновить структуру таблиць.

Крім того, Entity Framework Core підтримує роботу зі зв'язками між таблицями, такими як one-to-many та many-to-many. Це було особливо важливо під час реалізації системи манги, глав, жанрів, коментарів та списків користувачів, де між сутностями існує велика кількість взаємозв'язків.

Хоча використання ORM може дещо поступатися ручним SQL-запитам у максимально оптимізованих сценаріях, для розроблюваного веб-застосунка переваги у швидкості та простоті розробки, є значно важливішими.

Entity Framework Core було обрано через його тісну інтеграцію з ASP.NET Core, підтримку міграцій, зручну роботу зі зв'язками між сутностями та можливість швидко реалізовувати складну серверну логіку без необхідності постійного написання SQL-коду вручну.

Наступним етапом після вибору технології для реалізації серверної частини став вибір системи управління базою даних. Оскільки веб-платформа читання коміксів передбачає збереження значної кількості взаємопов'язаних даних, зокрема інформації про користувачів, твори, глави, сторінки, жанри, коментарі та списки користувача, база даних повинна забезпечувати надійне зберігання інформації, підтримку зв'язків між сутностями та зручну взаємодію із серверною частиною системи.

Для реалізації таких вимог доцільним є використання реляційної системи управління базами даних, оскільки структура даних у проекті має чітко визначені сутності та зв'язки між ними. Наприклад, одна манга може мати багато глав, одна глава може містити багато сторінок, користувач може створювати власні списки, залишати коментарі та взаємодіяти з контентом. Така структура добре підходить для реляційної моделі даних.

Під час аналізу були розглянуті такі системи управління базами даних, як PostgreSQL, MySQL та SQL Server. PostgreSQL є потужною об'єктно-реляційною СУБД з відкритим кодом, яка підтримує складні запити, розширення та роботу з великими обсягами даних.

MySQL є однією з найпоширеніших реляційних СУБД, яка часто використовується у веб-застосунках. Її перевагами є простота використання, висока швидкість роботи та велика кількість навчальних матеріалів. Водночас MySQL має дещо меншу гнучкість при роботі зі складними структурами даних та розширеними можливостями порівняно з PostgreSQL і SQL Server.

SQL Server є реляційною системою управління базами даних, розробленою компанією Microsoft. Її перевагою є тісна інтеграція з платформою .NET, ASP.NET Core та Entity Framework Core. Це дозволяє зручно створювати моделі даних, виконувати міграції, налаштовувати зв'язки між таблицями та працювати

з базою даних без необхідності постійного написання SQL-запитів вручну. Крім того, SQL Server забезпечує стабільну роботу зі структурованими даними, підтримує зв'язки між таблицями та має зручні інструменти для адміністрування.

У розроблюваній системі використовується SQL Server, оскільки дана СУБД найкраще поєднується з обраною серверною технологією ASP.NET Core. Важливим фактором стала підтримка Entity Framework Core, що дозволило організувати роботу з базою даних через моделі та міграції. Використання SQL Server дало змогу зручно реалізувати збереження даних про користувачів, мангу, глави, сторінки, жанри, коментарі та списки користувачів, а також забезпечити цілісність зв'язків між ними.

Таким чином, після визначення основних технологій для клієнтської та серверної частин веб-застосунку, наступним етапом став вибір інтерфейсу обміну даними між компонентами системи. API (Application Programming Interface) є механізмом, який забезпечує взаємодію між клієнтською та серверною частинами, дозволяючи передавати дані через мережеві запити та отримувати результати обробки у стандартизованому форматі.

Під час аналізу були розглянуті такі підходи до побудови API, як REST API, GraphQL та WebSocket API. Дані технології широко використовуються у сучасній веброзробці, однак мають різні принципи організації обміну даними та сфери застосування.

GraphQL є технологією для побудови API, розробленою компанією Meta. Основною особливістю GraphQL є те, що клієнтська частина самостійно визначає, які саме дані необхідно отримати від сервера. Як показано на діаграмі (рис. 2.11), клієнт формує єдиний запит (Single Query), у якому вказує лише необхідні поля (Requested Fields), після чого сервер повертає спеціально сформовану відповідь (Custom Response). Такий підхід дозволяє уникнути надлишкових даних у відповіді та зменшити кількість запитів до сервера. GraphQL добре підходить для складних систем із великою кількістю взаємопов'язаних даних, однак потребує складнішого налаштування серверної логіки, створення схем та resolver-ів [14].

WebSocket API забезпечує постійне двостороннє з'єднання між клієнтом та сервером у режимі реального часу. На діаграмі показано (рис. 2.11), що після встановлення відкритого з'єднання (Open Connection) клієнт і сервер можуть виконувати двосторонній обмін даними (Two-way Communication), забезпечуючи передачу інформації в реальному часі (Real-time Data). Даний підхід використовується у чатах, системах онлайн-сповіщень, потокових сервісах та інших застосунках, де необхідний миттєвий обмін даними. Проте для більшості стандартних вебзастосунків використання WebSocket є надлишковим та значно ускладнює серверну архітектуру [15].

REST API є одним із найпоширеніших підходів до організації взаємодії між клієнтом і сервером. Як показано на рисунку 2.11, клієнт надсилає HTTP-запит (HTTP Request) до конкретного ресурсу системи (Resource Endpoint), після чого сервер повертає відповідь у форматі JSON (JSON Response). REST базується на використанні стандартних HTTP-методів, таких як GET, POST, PUT та DELETE, для виконання операцій над ресурсами системи. Основними перевагами REST API є простота реалізації, зрозуміла структура запитів та хороша підтримка більшістю сучасних технологій і фреймворків [16].

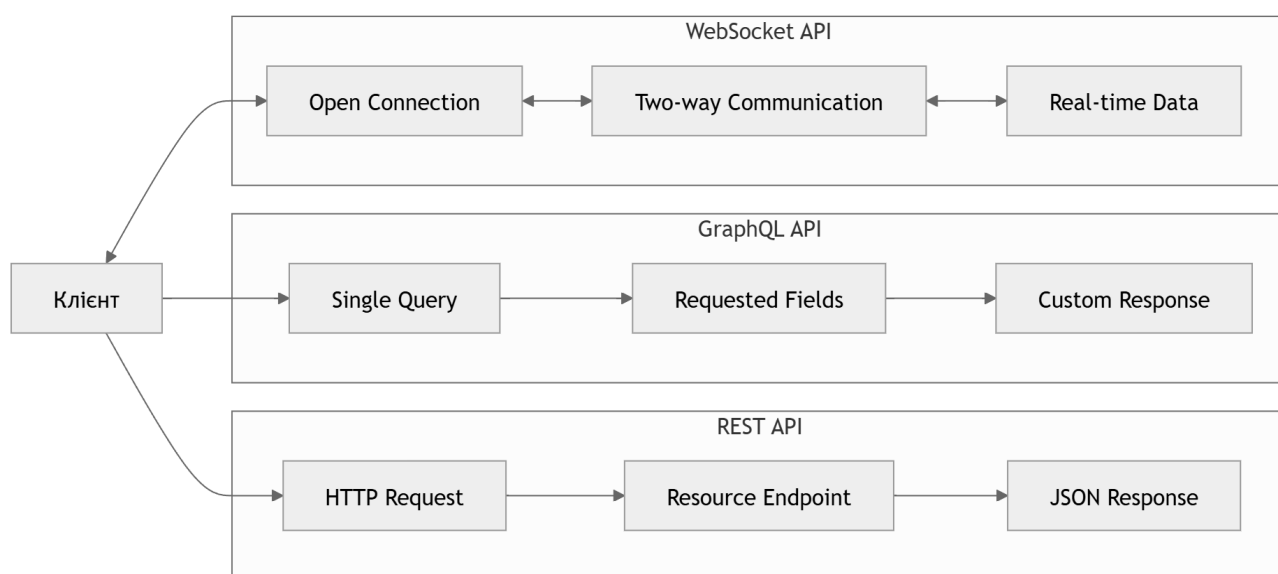


Рисунок 2.11 – Діаграма принципів роботи REST API, GraphQL API та WebSocket API

На відміну від GraphQL, REST API не потребує складного опису схем та логіки отримання даних, що значно спрощує розробку серверної частини. У порівнянні з WebSocket, REST API краще підходить для систем, де взаємодія з сервером відбувається через окремі запити користувача, а не через постійне з'єднання у реальному часі. Використання REST API дозволить забезпечити ефективну взаємодію між клієнтською та серверною частинами системи.

Після визначення основних технологій та програмних засобів, що використовуються у процесі розробки інформаційної системи, необхідно обрати середовище розробки, яке забезпечуватиме створення, тестування та налагодження програмного забезпечення.

Для розробки як серверної, так і клієнтської частин системи використовувався редактор Visual Studio Code. Дане середовище забезпечує підтримку мов програмування C#, TypeScript та JavaScript, інтеграцію з npm-пакетами, системою контролю версій Git та засобами налагодження вебзастосунків. Завдяки системі розширень Visual Studio Code підтримує роботу з ASP.NET Core, React, Entity Framework Core та іншими технологіями, що використовувалися у проєкті [17].

Серед основних переваг Visual Studio Code можна виділити швидкодію, невелике споживання системних ресурсів, підтримку автодоповнення коду, підсвічування синтаксису та інтегрований термінал. Також редактор підтримує автоматичне оновлення сторінки під час розробки клієнтської частини, що значно прискорює процес створення інтерфейсу користувача.

Альтернативами Visual Studio Code можуть виступати Visual Studio, JetBrains Rider або WebStorm. Visual Studio має ширший набір вбудованих інструментів для розробки .NET-застосунків, однак потребує значно більше системних ресурсів та орієнтована переважно на серверну частину. Rider і WebStorm також забезпечують потужні можливості для розробки, проте є комерційними програмними продуктами.

У результаті Visual Studio Code було обрано як універсальне середовище розробки, яке поєднує функціональність, швидкодію та безкоштовне поширення, забезпечуючи зручну роботу одночасно з фронтенд- та бекенд-технологіями.

РОЗДІЛ 3

РОЗРОБКА ВЕБ ПЛАТФОРМ ДЛЯ РОЗПОВСЮДЖЕННЯ ТА ЧИТАННЯ ЦИФРОВИХ КОМІКСІВ

3.1 Практична реалізація об'єкта проєктування

Практична реалізація проєкту розпочалася зі створення структури бази даних за допомогою ORM Entity Framework Core. На даному етапі були створені моделі сутностей, які описують основні компоненти системи та їх взаємозв'язки. Кожна модель реалізована у вигляді окремого C#-класу та відповідає певній таблиці у реляційній базі даних. Такий підхід дозволив реалізувати об'єктно-орієнтовану взаємодію з даними та забезпечити автоматичне формування структури бази даних на основі коду програми. Для створення таблиць, зовнішніх ключів, індексів та зв'язків між сутностями використовувалися можливості Entity Framework Core та механізм міграцій. Міграції дозволяють поступово змінювати структуру бази даних відповідно до змін у моделях програми, не створюючи базу даних заново вручну.

Після створення або зміни моделей, наприклад додавання нової таблиці, поля чи зв'язку між сутностями, у проєкті створювалася нова міграція. Вона фіксує зміни у структурі моделей і формує спеціальний файл із командами для оновлення бази даних. У цьому файлі описується, які таблиці потрібно створити, які поля додати або змінити, які зовнішні ключі чи індекси потрібно налаштувати.

Після створення міграції виконувалося оновлення бази даних. На цьому етапі Entity Framework Core застосовує згенеровані зміни до Microsoft SQL Server.

Однією з основних моделей системи є модель Manga (ліст. 3.1), яка відповідає за збереження інформації про комікси та мангу. Для прив'язки класу до конкретної таблиці бази даних використовується атрибут `[Table("Manga")]`, який визначає назву таблиці у базі даних. Первинний ключ таблиці описується через властивість `Id`, для якої використано атрибуту `[Key]` та `[DatabaseGenerated(DatabaseGeneratedOption.Identity)]`. Це дозволяє автоматично генерувати унікальний ідентифікатор для кожного запису.

Для окремих властивостей моделі використовуються атрибути валідації та обмеження довжини рядків, зокрема [Required] та [StringLength()]. Наприклад, поля Title, EnglishTitle, OriginalTitle, Author і Description є обов'язковими для заповнення, що забезпечує цілісність даних та унеможливлює створення неповних записів у базі даних. Для текстових полів також встановлюється максимальна довжина символів, що дозволяє контролювати структуру даних та оптимізувати використання пам'яті бази даних.

Модель Manga містить не лише прості поля, але й зовнішні ключі до інших таблиць. Наприклад, поле AgeRatingId використовується для зв'язку з таблицею вікових рейтингів AgeRating, а поле MangaTypeId – для зв'язку з таблицею типів творів MangaType.

Окремо для поля Rate було налаштовано точність числового типу за допомогою атрибута [Column(TypeName = "decimal(3, 2)")]. Це дозволяє зберігати рейтинг твору у форматі десяткового числа з двома знаками після коми, що є зручним для відображення оцінок користувачів.

Лістинг 3.1 – Фрагмент коду з файлу Manga.cs

```
public class Manga
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    [Required]
    [StringLength(255)]

    public string Title { get; set; } = string.Empty;

    [Required]
    public string EnglishTitle { get; set; } = string.Empty;

    [Required]
    public string OriginalTitle { get; set; } = string.Empty;
    public string AltTitle { get; set; } = string.Empty;
    public string? CoverImagePath { get; set; }

    [Required]
    [StringLength(500)]
    public string Author { get; set; } = string.Empty;
    [StringLength(100)]
    public string Artist { get; set; } = string.Empty;
```

```

[Required]
public string Description { get; set; } = string.Empty;

[Required]
public int Year { get; set; }

[Required]
public string Status { get; set; } = string.Empty
public string Publisher { get; set; } = string.Empty;
public int AgeRatingId { get; set; }
[Column(TypeName = "decimal(3, 2)")]
public decimal Rate { get; set; }

```

кінець лістингу 3.1

Для реалізації зв'язків між таблицями бази даних у моделі Manga використовуються навігаційні властивості (ліст. 3.2). Навігаційні властивості дозволяють працювати з пов'язаними даними через об'єкти C# без необхідності ручного написання SQL-запитів або самостійного виконання об'єднання таблиць.

У моделі Manga навігаційна властивість AgeRating забезпечує зв'язок із таблицею вікових обмежень AgeRating, а властивість Type – із таблицею типів творів MangaType. Завдяки цьому Entity Framework Core автоматично визначає зовнішні ключі та дозволяє отримувати пов'язану інформацію про віковий рейтинг або тип конкретного твору.

Для реалізації зв'язку один-до-багатьох використовується колекційна навігаційна властивість Chapters, яка містить список усіх розділів, що належать конкретній манзі. Це дозволяє отримувати всі глави твору через один об'єкт Manga без необхідності окремих SQL-запитів до таблиці Chapter.

Лістинг 3.2 – Навігаційні властивості моделі Manga.cs

```

public MangaType Type { get; set; } = null!;
    public AgeRating AgeRating { get; set; } = null!;

    public ICollection<Chapter> Chapters { get; set; } = new List<Chapter>();
    public ICollection<MangaGenre> MangaGenres { get; set; } = new
List<MangaGenre>();
    public MangaStats MangaStats { get; set; } = null!;
    public ICollection<UserListManga> UserListMangas { get; set; }
    = new List<UserListManga>();

```

кінець лістингу 3.2

Навігаційна властивість MangaGenres використовується для реалізації зв'язку багато-до-багатьох між творами та жанрами через проміжну таблицю MangaGenre (ліст. 3.3). Такий підхід дозволяє одному твору належати до кількох жанрів одночасно, а одному жанру – бути пов'язаним із багатьма творами.

Для роботи зі статистичною інформацією використовується навігаційна властивість MangaStats, яка реалізує зв'язок один-до-одного між коміксом та статистикою твору. Через дану властивість система може отримувати кількість переглядів, коментарів або вподобань конкретного твору.

Також у моделі реалізовано колекційну навігаційну властивість UserListMangas, яка використовується для взаємодії з користувацькими списками. Через неї один твір може одночасно належати до декількох списків різних користувачів, наприклад «Улюблене», «Читаю» або «Заплановано».

Навігаційна властивість MangaGenres використовується для реалізації зв'язку багато-до-багатьох між творами та жанрами через проміжну таблицю MangaGenre (ліст. 3.3). Такий тип зв'язку необхідний тому, що один твір може мати декілька жанрів, наприклад «Action», «Fantasy» і «Drama», а один жанр, у свою чергу, може належати багатьом різним творам.

Лістинг 3.3 – MangaGenres.cs

```
public class MangaGenre {
    public int MangaId { get; set; }
    public int GenreId { get; set; }
    public Manga Manga { get; set; } = null!;
    public Genre Genre { get; set; } = null!;}

```

кінець лістингу 3.3

У реляційній базі даних такий зв'язок не реалізується напряму між двома таблицями Manga і Genre (ліст. 3.4). Для цього створюється додаткова проміжна таблиця MangaGenre, яка зберігає пари ідентифікаторів: MangaId та GenreId. Поле MangaId вказує, до якого твору належить запис, а поле GenreId – який саме жанр пов'язаний із цим твором.

Лістинг 3.4 – Genre.cs

```
[Index(nameof(Name), IsUnique = true)]
public class Genre
{
    [Key]
    public int Id { get; set; }
    [Required]
    [StringLength(50)]
    public string Name { get; set; } = string.Empty;
    public ICollection<MangaGenre> MangaGenres { get; set; } = new
    List<MangaGenre>();}

```

кінець лістингу 3.4

Наприклад, якщо манга має три жанри, то в таблиці Manga вона буде зберігатися лише один раз, а кожен її жанр буде записаний окремим рядком у таблиці MangaGenre.

При цьому самі назви жанрів не дублюються в таблиці Manga, а зберігаються окремо в таблиці Genre (ліст. 3.4). Це дозволяє уникнути повторення однакових значень, спрощує фільтрацію творів за жанрами та забезпечує цілісність даних. Клас MangaGenre у коді виконує роль проміжної сутності. Він містить зовнішні ключі MangaId і GenreId, а також навігаційні властивості Manga та Genre, які дозволяють Entity Framework Core отримувати пов'язані об'єкти. У свою чергу, клас Genre містить колекцію MangaGenres, яка показує, з якими творами пов'язаний конкретний жанр.

Завдяки такій структурі система може легко отримати всі жанри певного твору або, навпаки, всі твори, які належать до конкретного жанру. Наприклад, для сторінки манги можна завантажити список її жанрів, а для каталогу реалізувати фільтрацію за вибраним жанром.

Ключовим файлом, який відповідає за створення бази даних та налаштування взаємозв'язків між таблицями, є MangaDbContext.cs (ліст. 3.5). Даний клас успадковується від базового класу DbContext, який є основним компонентом Entity Framework Core для взаємодії з базою даних. Через конструктор клас отримує параметри підключення та конфігурації бази даних, які передаються до базового класу DbContext.

 Лістинг 3.5 – Конструктор класу у файлі MangaDbContext.cs

```
namespace MangaHeartApi.Data;
public class MangaDbContext : DbContext
{
    public MangaDbContext(DbContextOptions<MangaDbContext> options)
    : base(options)
    {}
}
```

кінець лістингу 3.5

У класі MangaDbContext оголошуються властивості типу DbSet (ліст. 3.6), кожна з яких відповідає окремій таблиці бази даних. Наприклад, Mangas відповідає таблиці творів, Users – таблиці користувачів, Chapters – таблиці розділів, а Genres – таблиці жанрів. Завдяки DbSet Entity Framework Core розуміє, які саме сутності потрібно відобразити у вигляді таблиць у базі даних.

 Лістинг 3.6 – Оголошення DbSet-властивостей у класі MangaDbContext

```
public DbSet<User> Users { get; set; }
public DbSet<Manga> Mangas { get; set; }
public DbSet<MangaType> MangaTypes { get; set; }
public DbSet<Chapter> Chapters { get; set; }
public DbSet<Page> Pages { get; set; }
public DbSet<Genre> Genres { get; set; }
public DbSet<MangaGenre> MangaGenres { get; set; }
public DbSet<Comment> Comments { get; set; }
public DbSet<ReadingProgress> ReadingProgresses { get; set; }
public DbSet<MangaStats> MangaStats { get; set; }
```

кінець лістингу 3.6

У методі OnModelCreating() налаштовуються складніші зв'язки (ліст. 3.7) між таблицями. Наприклад зв'язок між AgeRating і Manga реалізований як один-до-багатьох, оскільки один віковий рейтинг може використовуватися для багатьох творів. Зв'язок між Manga і MangaStats є один-до-одного, адже кожен твір має один набір статистичних даних. Зв'язок між Manga і Chapter реалізований як один-до-багатьох, оскільки один твір може містити багато розділів.

 Лістинг 3.7 – Фрагмент коду з методу OnModelCreating() класу MangaDbContext

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);modelBuilder.Entity<AgeRating>()
```

```

        .HasMany(ar => ar.Mangas)
        .WithOne(m => m.AgeRating)
        .HasForeignKey(m => m.AgeRatingId);

modelBuilder.Entity<Manga>()
    .HasOne(m => m.MangaStats)
    .WithOne(s => s.Manga)
    .HasForeignKey<MangaStats>(s => s.MangaId);

modelBuilder.Entity<Chapter>()
    .HasOne(c => c.Manga)
    .WithMany(m => m.Chapters)
    .HasForeignKey(c => c.MangaId);

```

кінець лістингу 3.7

Для забезпечення цілісності даних у контексті бази даних налаштовуються унікальні індекси. Наприклад (ліст. 3.8), для таблиці ReadingProgress встановлено унікальність комбінації UserId і ChapterId, щоб один користувач мав лише один запис прогресу для конкретного розділу. Для таблиці Page унікальною є комбінація ChapterId і PageNumber, що запобігає дублюванню сторінок у межах одного розділу. Також унікальними зроблено поля UserName та Email, щоб у системі не могли існувати користувачі з однаковим логіном або електронною поштою.

Лістинг 3.8 – Фрагмент налаштування унікальних індексів у класі
MangaDbContext

```

modelBuilder.Entity<ReadingProgress>()
    .HasIndex(x => new { x.UserId, x.ChapterId })
    .IsUnique();

modelBuilder.Entity<Page>()
    .HasIndex(x => new { x.ChapterId, x.PageNumber })
    .IsUnique();

modelBuilder.Entity<User>()
    .HasIndex(x => x.UserName)
    .IsUnique();

modelBuilder.Entity<User>()
    .HasIndex(x => x.Email)
    .IsUnique();

```

кінець лістингу 3.8

Після створення та налаштування бази даних було розпочато реалізацію перших контролерів ASP.NET Core. Контролер приймає HTTP-запит від клієнтської частини, виконує необхідну обробку даних, звертається до сервісів або бази даних та повертає відповідь клієнту у форматі JSON. Приклад реалізації нище розглянутого контролера MangaController наведено у Додатку А.

На початку контролера вказуються службові атрибути [ApiController] та [Route("api/[controller]")], які визначають, що даний клас є API-контролером та задають маршрут для HTTP-запитів. Також у конструкторі контролера MangaController, фрагменти коду якого розміщені в додатку А реалізовано механізм dependency injection, за допомогою якого до контролера передаються об'єкти контексту бази даних, сервісів та AutoMapper. Це дозволяє використовувати їх у подальшій роботі методів контролера.

У серверній частині системи використовується архітектурний підхід REST API, який забезпечує взаємодію між клієнтською та серверною частинами застосунку через HTTP-запити. Основою REST API є HTTP-методи GET, POST, PUT та DELETE. Такий підхід дозволяє стандартизувати роботу API та забезпечує зрозумілу структуру взаємодії між компонентами системи. У контролері MangaController метод GetAll() реалізує отримання даних, метод Create() відповідає за створення нових записів, метод Update() – за оновлення інформації, а метод Delete() – за видалення записів із бази даних.

Однією з основних функцій контролера є отримання даних із бази даних. Для цього було реалізовано метод GetAll(), який повертає список усіх коміксів із бази даних. Метод виконує запит до таблиці Mangas, завантажує пов'язані дані про тип манги та вікове обмеження, після чого перетворює отримані об'єкти у DTO та повертає результат клієнту.

Для отримання інформації про конкретний комікс використовується метод Get(int id). Даний метод приймає ідентифікатор манги через URL-адресу, після чого виконує пошук відповідного запису у базі даних. Разом із основною інформацією також завантажуються жанри твору через проміжну таблицю MangaGenres. Якщо запис не знайдено, сервер повертає помилку NotFound().

Для створення нових записів у системі реалізовано метод `Create()`. Даний метод доступний лише адміністраторам завдяки використанню атрибута `[Authorize(Roles = "Admin")]`. Метод приймає дані форми від клієнтської частини, передає їх до `servis MangaService`, після чого створює новий запис у базі даних та повертає результат клієнту.

Оновлення та видалення інформації також реалізовані через окремі методи контролера. Метод `Update()` відповідає за зміну інформації про мангу, а метод `Delete()` – за видалення запису з бази даних. Використання окремих HTTP-методів `PUT` та `DELETE` дозволяє дотримуватись принципів `REST API` та забезпечує правильну організацію серверної логіки.

Окремо у контролері реалізовано метод `UploadFile()`, який відповідає за завантаження файлів на сервер. Під час завантаження для кожного файлу автоматично генерується унікальна назва за допомогою `Guid`, після чого файл зберігається у папці `wwwroot/uploads`. Це дозволяє уникнути конфліктів назв файлів та забезпечує коректне зберігання медіаконтенту системи.

Для реалізації пошуку манги у системі було створено метод `Search()`. Метод отримує пошуковий запит від клієнтської частини та виконує пошук назв коміксів у базі даних за допомогою функції `EF.Functions.Like`. Після цього сервер повертає список знайдених результатів користувачу.

Однією з більш складних функцій контролера є реалізація рекомендацій схожих творів. Метод `GetRelatedManga()` аналізує жанри поточного твору, після чого знаходить інші комікси зі схожими жанрами. Результати сортуються за кількістю спільних жанрів та рейтингом. Такий підхід дозволяє реалізувати базову систему рекомендацій контенту для користувачів.

Для спрощення структури серверної частини та дотримання принципу розділення відповідальності великі та складні методи обробки даних були винесені у окремі сервіси. Такий підхід дозволяє зробити код більш структурованим, зрозумілим та придатним для подальшого розширення. У даному випадку контролери виконують роль обробників HTTP-запитів та відповідають лише за

приймання запитів від клієнтської частини і повернення результату, тоді як основна бізнес-логіка реалізується у сервісах.

Для реалізації логіки роботи з творами було створено сервіс MangaService, який відповідає за виконання операцій пошуку, фільтрації, створення, оновлення та обробки медіафайлів. Фрагменти коду сервісу MangaService.cs розміщені у додатку А.

На початку сервісу через механізм dependency injection до класу передаються об'єкти контексту бази даних MangaDbContext та IMapper. Контекст бази даних використовується для взаємодії з таблицями SQL Server через Entity Framework Core, а AutoMapper забезпечує автоматичне перетворення моделей бази даних у DTO-об'єкти та навпаки. Такий підхід дозволяє уникнути дублювання коду та спрощує передачу даних між серверною та клієнтською частинами системи.

Однією з основних функцій сервісу є реалізація пошуку творів. Для цього було створено метод SearchMangaAsync(), який виконує пошук за декількома полями одночасно. Пошук здійснюється не лише за основною назвою твору, але й за англійською назвою, альтернативними назвами, автором, художником, видавцем та жанрами. Для реалізації пошуку використовується функція EF.Functions.Like, яка дозволяє виконувати SQL-подібний пошук за частковим збігом тексту. Отримані результати додатково сортуються за рейтингом манги, що дозволяє відображати найбільш популярні твори першими. Після цього дані перетворюються у DTO-об'єкти та повертаються клієнтській частині.

Окремо у сервісі реалізовано метод GetMangaByGenresAsync(), який відповідає за фільтрацію манги за декількома жанрами одночасно. Метод отримує список жанрів від клієнтської частини та виконує складний LINQ-запит до бази даних. У процесі роботи система аналізує жанри кожного твору та повертає лише ті записи, які містять усі жанри, обрані користувачем. Така функціональність дозволяє реалізувати розширену систему фільтрації контенту на платформі.

Для створення нових записів манги використовується метод CreateAsync(). Даний метод реалізує одразу декілька важливих етапів обробки даних. Спочатку відбувається очищення назви манги від недопустимих символів файлової системи,

після чого формується структура директорій для зберігання обкладинок та глав. Далі система створює відповідні папки у файловій системі сервера та виконує завантаження файлу обкладинки. Для уникнення конфліктів назв використовується генерація унікального ідентифікатора Guid. Після завершення завантаження формується шлях до зображення, який зберігається у базі даних.

Після обробки файлів сервіс створює об'єкт манги та формує зв'язки між мангою та жанрами через проміжну таблицю MangaGenres. Далі новий запис додається до бази даних та зберігається за допомогою методу SaveChangesAsync(). Після завершення створення запису система повторно отримує повну інформацію про мангу разом із пов'язаними сутностями та повертає результат клієнтській частині.

Для оновлення інформації про мангу реалізовано метод UpdateAsync(). Метод виконує пошук відповідного запису у базі даних, після чого оновлює всі необхідні поля, зокрема назву, опис, автора, рік випуску, рейтинг та інші характеристики твору. Також у процесі оновлення може змінюватися обкладинка манги. У такому випадку файл повторно завантажується на сервер, а шлях до нового зображення оновлюється у базі даних. Окремо виконується перебудова зв'язків між мангою та жанрами, оскільки користувач може змінювати набір жанрів твору.

Використання окремих сервісів для реалізації бізнес-логіки дозволило зробити архітектуру серверної частини більш модульною та зрозумілою.

Для передачі даних між серверною та клієнтською частинами у проєкті використовуються DTO (Data Transfer Object) – спеціальні об'єкти передачі даних. DTO застосовуються для того, щоб відокремити внутрішню структуру моделей бази даних від даних, які безпосередньо передаються через API. Такий підхід дозволяє підвищити безпеку системи, зменшити кількість зайвих даних у відповіді сервера та зробити структуру API більш зрозумілою й стабільною.

У даному проєкті DTO використовуються у всіх методах контролерів та сервісів. Наприклад, при отриманні інформації про мангу сервер не повертає пряму модель бази даних Manga, а спочатку перетворює її у об'єкт MangaDto, який містить лише необхідні для клієнтської частини поля. Аналогічно для

створення та оновлення записів використовуються окремі DTO-об'єкти `CreateMangaDto` та `UpdateMangaDto` (ліст. 3.9), які містять лише ті дані, які користувач може надсилати на сервер.

Використання DTO також дозволило уникнути передачі службових полів, внутрішніх зв'язків Entity Framework Core та навігаційних властивостей, які не повинні бути доступними клієнтській частині. Крім цього, DTO спрощують контроль структури JSON-відповідей API та дозволяють змінювати внутрішню структуру моделей бази даних без необхідності змінювати клієнтську частину застосунку.

Лістинг 3.9 – UpdateMangaDto.cs

```
public class UpdateMangaDto
{
    public string Title { get; set; } = string.Empty;
    public string EnglishTitle { get; set; } = string.Empty;
    public string OriginalTitle { get; set; } = string.Empty;
    public string AltTitle { get; set; } = string.Empty;
    public string CoverImagePath { get; set; } = string.Empty;
    public string Author { get; set; } = string.Empty;
    public string Artist { get; set; } = string.Empty;

    public int Year { get; set; }
    public string Status { get; set; } = string.Empty;
    public string Publisher { get; set; } = string.Empty;
    public decimal Rate { get; set; }
    public int MangaTypeId { get; set; }
    public int AgeRatingId { get; set; }
    public string Description { get; set; } = string.Empty;
    public List<int> GenreIds { get; set; } = new();
    public IFormFile? CoverFile { get; set; }
}
```

кінець лістингу 3.9

Для спрощення роботи з DTO у проєкті використовується бібліотека `AutoMapper`. Основним призначенням `AutoMapper` є автоматичне перетворення об'єктів одного типу в інший. У даному випадку `AutoMapper` використовується для перетворення моделей бази даних у DTO-об'єкти та навпаки. Це дозволяє уникнути ручного копіювання великої кількості полів між об'єктами та значно зменшує обсяг однотипного коду.

Налаштування правил перетворення виконується за допомогою профілів (Profile). Профіль є окремим класом, у якому описуються всі правила мапінгу між моделями та DTO. У проєкті для роботи з мангою було створено профіль MangaProfile, приклад якого наведено у лістингу 3.10.

У наведеному профілі за допомогою методу CreateMap створюються правила перетворення між різними типами об'єктів. Перший мапінг CreateMap<Manga, MangaDto>() використовується для перетворення моделі бази даних Manga у DTO-об'єкт MangaDto. У більшості випадків AutoMapper автоматично співставляє поля з однаковими назвами, однак у деяких ситуаціях необхідно явно вказувати спосіб отримання даних.

Для цього використовується метод ForMember. Наприклад, поле Type у DTO отримує значення з навігаційної властивості src.Type.Name, а поле AgeRating – із властивості src.AgeRating.Code. Аналогічно список жанрів формується через вибірку назв жанрів із проміжної таблиці MangaGenres. Це дозволяє передавати клієнтській частині вже готові дані у зручному форматі без необхідності додаткової обробки на фронтенді.

Наступний мапінг CreateMap<CreateMangaDto, Manga>() використовується під час створення нової манги. У даному випадку всі поля автоматично копіюються з DTO у модель бази даних, окрім поля CoverImagePath. Для нього використовується метод Ignore(), оскільки шлях до зображення формується окремо після завантаження файлу на сервер і не повинен надходити напряму від користувача.

Останній мапінг CreateMap<UpdateMangaDto, Manga>() використовується для оновлення інформації про мангу. Завдяки AutoMapper система може автоматично переносити значення з DTO у модель бази даних без необхідності вручну присвоювати кожне поле окремо.

Лістинг 3.10 – Профіль MangaProfile

```
public class MangaProfile : Profile
{
    public MangaProfile()
    {
```

```

        CreateMap<Manga, MangaDto>()
            .ForMember(dest => dest.Type, opt => opt.MapFrom(src =>
src.Type.Name))
            .ForMember(dest => dest.AgeRating, opt => opt.MapFrom(src =>
src.AgeRating.Code))
            .ForMember(dest => dest.Genres, opt => opt.MapFrom(src =>
src.MangaGenres.Select(mg => mg.Genre.Name).ToList()));

        CreateMap<CreateMangaDto, Manga>()
            .ForMember(dest => dest.CoverImagePath, opt => opt.Ignore());

        CreateMap<UpdateMangaDto, Manga>();
    }
}

```

кінець лістингу 3.10.

Використання AutoMapper та профілів дозволило значно спростити серверну логіку, зменшити кількість дубльованого коду та зробити процес передачі даних між компонентами системи більш зручним і безпечним. Крім цього, централізоване налаштування мапінгу у профілях полегшує підтримку проєкту та спрощує внесення змін у структуру DTO або моделей бази даних.

Завершальним етапом реалізації серверної частини стало налаштування основних служб застосунку у файлі Program.cs. Даний файл є точкою входу в ASP.NET Core-застосунок і відповідає за підключення необхідних сервісів, налаштування бази даних, авторизації, роботи з файлами, CORS-політики та маршрутизації API-запитів.

У файлі Program.cs наведеного у додатку А, було налаштовано підключення до бази даних за допомогою Entity Framework Core. Для цього сервіс MangaDbContext реєструється в контейнері залежностей, а рядок підключення отримується з конфігураційного файлу через DefaultConnection. Це дозволяє серверній частині взаємодіяти з Microsoft SQL Server та виконувати всі необхідні операції з даними.

Також у конфігурації застосунку були зареєстровані основні сервіси системи, зокрема MangaService, ChapterService та UserService. Завдяки цьому контролери можуть отримувати доступ до сервісів через механізм dependency injection і використовувати їх для виконання бізнес-логіки. Окрім цього, було підключено

AutoMapper для автоматичного перетворення моделей бази даних у DTO-об'єкти та навпаки.

Оскільки система працює з великою кількістю зображень, у Program.cs було налаштовано обмеження розміру HTTP-запитів та multipart-форм до 200 МБ. Це необхідно для коректного завантаження обкладинок, глав і сторінок манги. Також було підключено статичну роздачу файлів із фізичної директорії E:\MangaData, яка доступна клієнтській частині за шляхом /manga-files. Завдяки цьому фронтенд може отримувати зображення напряму через URL-адреси, що зберігаються в базі даних.

Для взаємодії клієнтської та серверної частин було налаштовано CORS-політику AllowFrontend. Вона дозволяє приймати запити з адреси клієнтської частини, на якій під час розробки запускається клієнтська частина React-застосунку. Політика дозволяє використовувати різні HTTP-методи та заголовки, що необхідно для повноцінної роботи REST API.

Крім цього, у серверній частині було підключено Swagger, який використовується для перегляду, тестування та документування API-методів. Це спростило процес перевірки роботи контролерів під час розробки та дозволило швидко тестувати запити до серверної частини без використання клієнтського інтерфейсу.

Після налаштування всіх сервісів у конвеєр обробки запитів було додано middleware для роботи зі статичними файлами, CORS, Swagger, маршрутизацією контролерів та HTTPS-перенаправленням. Завдяки цьому серверна частина отримала повністю налаштоване середовище для обробки запитів від клієнтської частини, роботи з базою даних, авторизації користувачів та передачі медіафайлів.

Файл Program.cs наведений у додатку А, об'єднує всі основні налаштування серверної частини інформаційної системи та забезпечує її коректну взаємодію з клієнтською частиною, базою даних і файловою системою. Саме після виконання цих налаштувань серверна частина стала готовою до приймання запитів від React-застосунку та повноцінного обміну даними між усіма компонентами системи.

Розробка клієнтської частини розпочалася з створення головної сторінки застосунку HomePage. Вона виконує роль початкової сторінки вебплатформи, з якої користувач може перейти до основних розділів системи. Структура сторінки побудована на основі компонентного підходу React, що дозволяє розділити інтерфейс на окремі незалежні частини.

У компоненті HomePage (ліст. 3.11) використовується загальний контейнер сторінки, який задає мінімальну висоту екрана, ширину, фоновий колір та основний стиль тексту.

Головна сторінка має модульну структуру та складається з окремих компонентів, що відповідають за різні частини інтерфейсу. Усередині сторінки розміщуються основні компоненти інтерфейсу: Navbar, HeroSection, WhatsHotSection та Footer. Компонент Navbar відповідає за верхню навігаційну панель, HeroSection – за головний вітальний блок сторінки, а Footer – за нижню частину сторінки.

Лістинг 3.11 – HomePage.tsx

```
export function HomePage() {
  return (
    <div className="min-h-screen w-full bg-brand-beige font-dm-sans text-brand-navy">
      <Navbar />
      <main className="p-8">
        <HeroSection />
        <WhatsNewSection />
        <WhatsHotSection />
        <ContinueReading />
      </main>
      <Footer />
    </div>
  )
}
```

кінець лістингу 3.11

Загальний вигляд навігаційної панелі та головного інформаційного блоку сторінки наведено на рисунку 3.1. Крім основних секцій на сторінці також реалізовано блок WhatsNewSection, який використовується для відображення нових творів, доданих до системи. Для авторизованих користувачів додатково

відображається секція ContinueReading, яка дозволяє швидко продовжити читання з останнього відкритого розділу. Такий підхід дозволяє розділити інтерфейс на незалежні компоненти та спрощує подальше розширення функціональності клієнтської частини без необхідності змінювати загальну структуру сторінки.

Одним із ключових компонентів клієнтської частини системи є навігаційна панель Navbar, яка забезпечує швидкий доступ користувача до основних функцій вебзастосунку. Даний компонент відображається у верхній частині сторінки та залишається закріпленим під час прокручування сторінки завдяки використанню стилю sticky top-0.

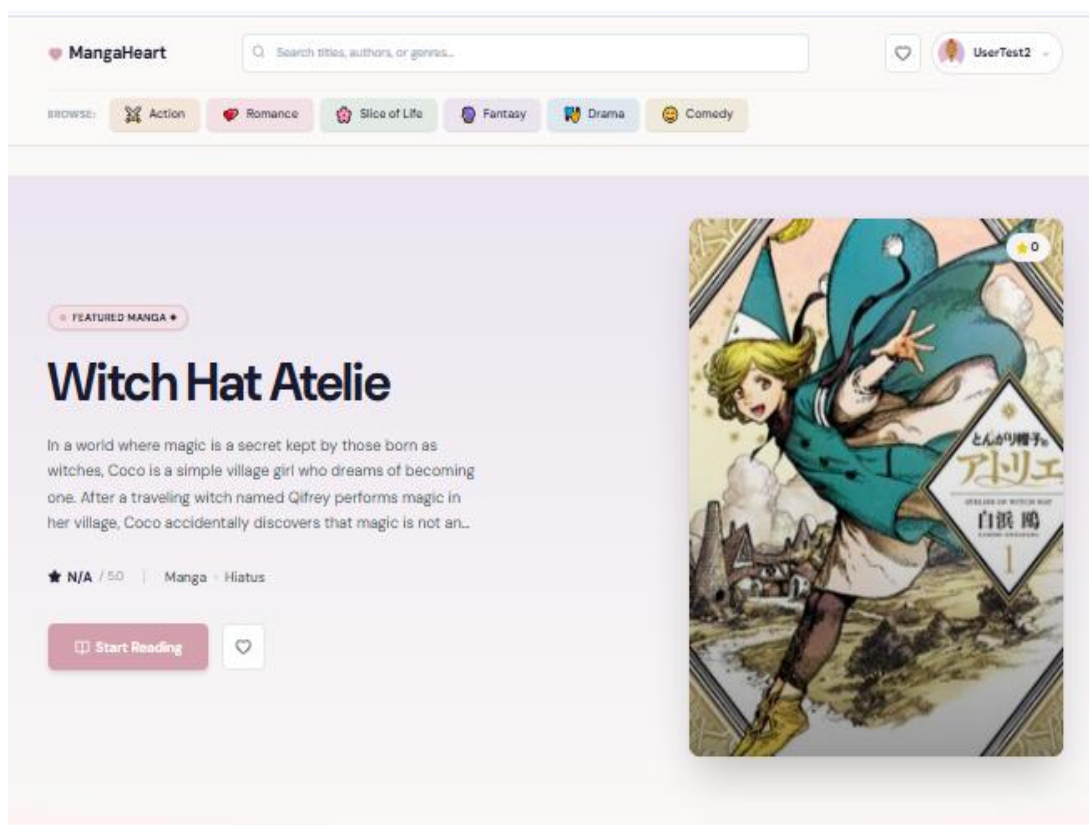


Рисунок 3.1 – Скріншот головної сторінки

У структурі навігаційної панелі реалізовано логотип платформи, рядок пошуку, систему навігації за жанрами та блок авторизації користувача. Для побудови інтерфейсу використовувався компонентний підхід React, а стилізація виконувалася за допомогою Tailwind CSS. Це дозволило створити адаптивний та сучасний інтерфейс із мінімальною кількістю окремих CSS-файлів.

У верхній частині навігаційної панелі розміщується логотип платформи MangaHeart, який одночасно є посиланням на головну сторінку застосунку. Для реалізації переходів між сторінками використовується бібліотека React Router та компонент Link. Також для мобільних пристроїв реалізовано кнопку меню з використанням іконки Menu з бібліотеки lucide-react.

Однією з основних функцій Navbar є система пошуку манги. Для цього було реалізовано окремий рядок пошуку, який працює у реальному часі. Під час введення тексту користувачем за допомогою хука useEffect автоматично виконується HTTP-запит до серверної частини через метод fetch. Запит надсилається до API-маршруту /api/Manga/searchHeader, а сервер повертає список знайдених творів у форматі JSON.

Отримані результати зберігаються у стані компонента через useState та відображаються у випадаючому списку під рядком пошуку. Для кожного знайденого твору виводиться обкладинка, назва, жанри, тип манги, рейтинг та короткий опис. Завдяки цьому користувач може швидко знайти потрібний твір без переходу на окрему сторінку пошуку.

Для покращення взаємодії з користувачем у компоненті також реалізовано механізм автоматичного закриття випадаючого списку при натисканні за межами області пошуку. Для цього використовується хук useRef, який дозволяє отримати доступ до DOM-елемента, та обробник події mousedown.

Окрему частину навігаційної панелі займає система авторизації користувачів. Під час завантаження компонента через localStorage перевіряється наявність інформації про авторизованого користувача. Якщо користувач увійшов у систему, у Navbar відображається компонент UserMenu; інакше користувачу показуються кнопки Register та Sign In. Для анімації кнопок використовується бібліотека framer-motion, що дозволяє створити плавні ефекти наведення та натискання.

У нижній частині навігаційної панелі реалізовано горизонтальний список жанрів. Для цього використовується окремий компонент GenreBadge, який відображає жанр, іконку та колір оформлення. Після натискання на жанр

користувач автоматично переходить до каталогу з відповідним фільтром через систему маршрутизації React Router.

Компонент Navbar розміщується на кожній сторінці вебплатформи і де виконує одразу декілька важливих функцій: забезпечує навігацію між сторінками, реалізує пошук контенту, взаємодію з системою авторизації та швидкий доступ до жанрових фільтрів.

Важливою складовою навігаційної панелі є рядок пошуку, який дозволяє користувачам швидко знаходити необхідний контент. Пошук реалізовано таким чином, щоб одночасно виконувати пошук за назвами творів, іменами авторів та жанрами. Під час введення тексту система надсилає запит до серверної частини та відображає список найбільш релевантних результатів у випадаючому меню. Такий підхід значно спрощує навігацію по великій бібліотеці контенту та дозволяє користувачеві швидко перейти до потрібного твору без необхідності ручного перегляду каталогу.

Після створення головної сторінки та навігаційної панелі було реалізовано сторінку каталогу. Вона призначена для перегляду доступних творів, фільтрації контенту за жанрами й статусом, а також переходу до сторінки окремої манги. Загальний вигляд сторінки каталогу наведено на рисунку 3.2.

Сторінка CatalogPage лістинг якої наведений у додатку Б, використовує компоненти Navbar, Footer, MangaCard та CatalogFilters, що дозволяє розділити інтерфейс на окремі логічні частини. У верхній частині сторінки розміщено навігаційну панель, нижче – службову навігацію Home > Catalog, заголовковий блок сторінки та основний блок із фільтрами й сіткою творів.

Для зберігання стану сторінки використовуються хуки useState. Зокрема, у стані зберігаються вибрані жанри, статус твору, мінімальний рейтинг, список жанрів для фільтрів, список отриманої манги та стан завантаження. Це дозволяє сторінці динамічно реагувати на дії користувача без повного перезавантаження сторінки. Взаємодія з серверною частиною реалізована через HTTP-запити до API. При першому відкритті сторінки виконується запит до /api/Genres, щоб отримати

список жанрів для бокової панелі фільтрів. Після цього, залежно від вибраного жанру, сторінка або завантажує повний список манги, або звертається до спеціального API-методу для отримання творів за конкретним жанром.

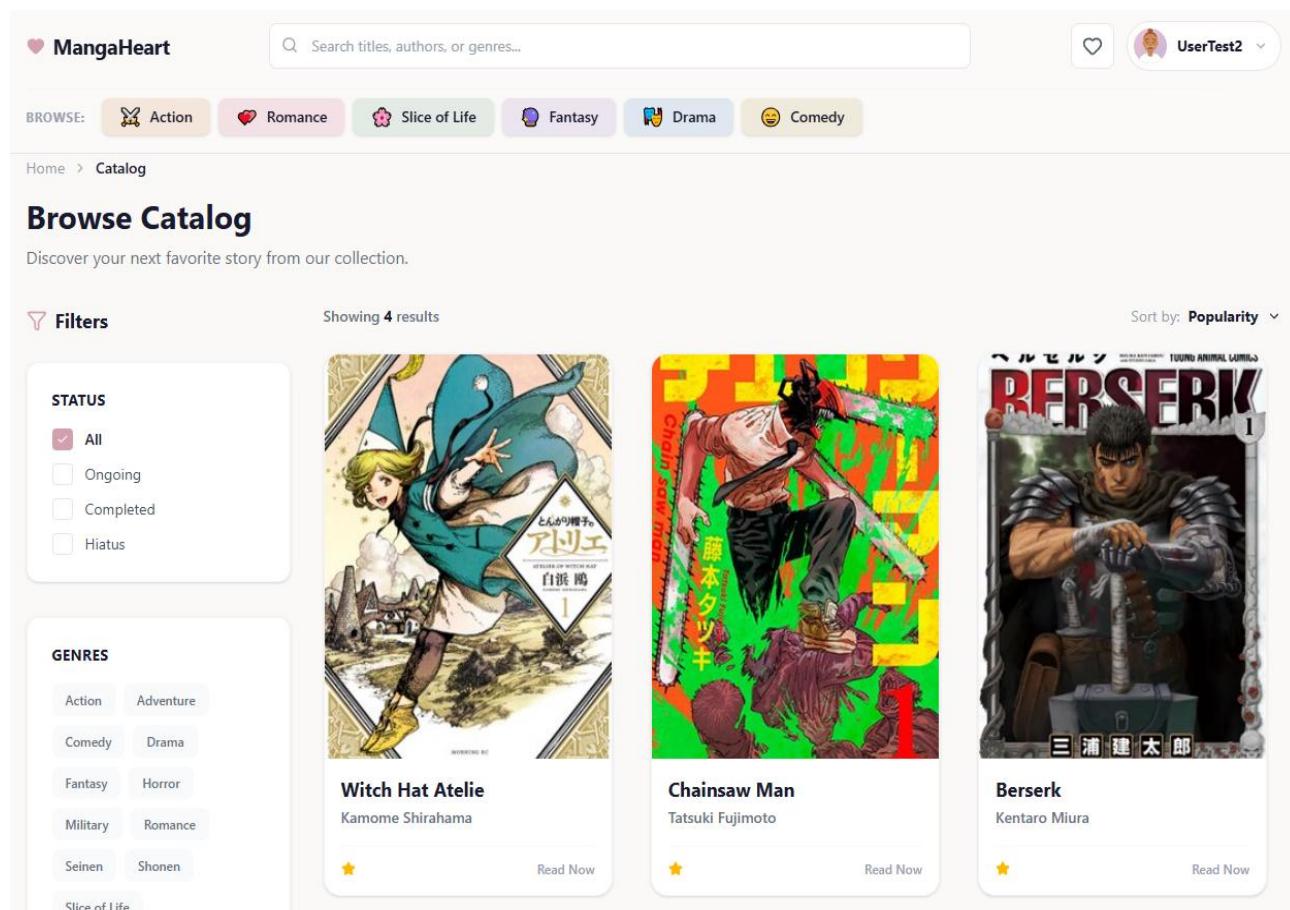


Рисунок 3.2 – Скріншот сторінки каталогу

Взаємодія з серверною частиною реалізована через HTTP-запити до API. При першому відкритті сторінки виконується запит до `/api/Genres`, щоб отримати список жанрів для бокової панелі фільтрів. Після цього, залежно від вибраного жанру, сторінка або завантажує повний список манги, або звертається до спеціального API-методу для отримання творів за конкретним жанром.

Також на сторінці реалізовано роботу з параметрами URL за допомогою `useSearchParams`. Завдяки цьому при переході з навігаційної панелі або виборі жанру у фільтрах жанр додається до адреси сторінки у вигляді параметра `genres`.

Це робить фільтрацію більш зручною, оскільки стан каталогу частково зберігається в URL.

Окремо реалізовано клієнтську фільтрацію за рейтингом і статусом. Після отримання даних від сервера масив манги додатково обробляється на клієнтській частині, після чого на сторінці відображаються лише ті твори, які відповідають вибраним параметрам.

Основний контент каталогу виводиться у вигляді адаптивної сітки карток. Для кожного твору використовується компонент `MangaCard`, а для плавної появи елементів застосовується бібліотека `framer-motion`.

Натиснувши на компонент `MangaCard`, користувач переходить на сторінку окремого тайтлу `MangaTitlePage`, яка є основною сторінкою взаємодії з конкретним твором та містить детальну інформацію про твір, список глав, коментарі та інші функціональні можливості системи. Саме на цій сторінці користувач отримує повну інформацію про твір, список глав, рекомендації схожої коміксів, систему коментарів та можливість взаємодії з власними списками. Приклад реалізації сторінки наведено у додатку Б, а скриншот сторінки представлено на рисунку 3.3.

На початку компонента за допомогою `React Hooks` створюються стани для керування вкладками, списками користувача, коментарями, завантаженням даних та іншими інтерактивними елементами сторінки. Для навігації між сторінками використовується `useNavigate`, а для отримання ідентифікатора манги з URL – `useParams`.

Після відкриття сторінки через `useEffect` виконується одночасне завантаження інформації про мангу та схожі твори за допомогою `Promise.all`. Серверна частина повертає JSON-об'єкти через API-запити `/api/manga/{id}` та `/api/manga/{id}/related`. Після отримання відповіді дані зберігаються у стані компонента та використовуються для побудови інтерфейсу сторінки.

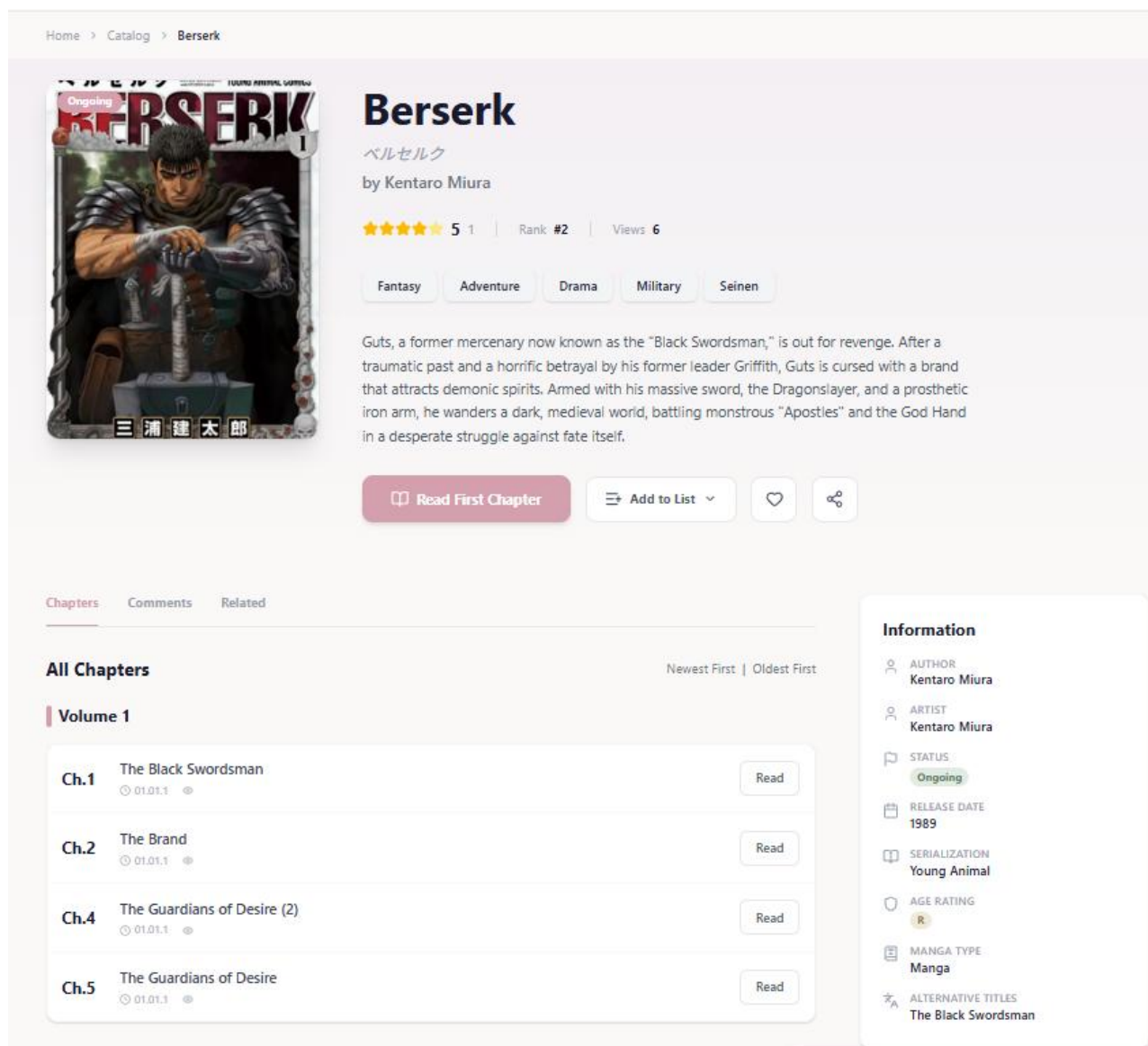


Рисунок 3.3 – Скриншот сторінки MangaTitlePage

Під час завантаження сторінки виконуються запити для отримання списків користувача, перевірки доданих списків та завантаження коментарів. Таким чином сторінка отримує всю необхідну інформацію для повноцінної взаємодії користувача з контентом.

У верхній частині сторінки реалізовано систему breadcrumbs-навігації, яка дозволяє користувачу швидко повернутися на головну сторінку або до каталогу. Назва поточної манги формується динамічно на основі отриманих даних.

Основний блок сторінки складається з обкладинки твору, інформації про автора, жанри, опису та елементів взаємодії користувача із системою. Для анімації появи елементів використовується бібліотека framer-motion, що робить інтерфейс

більш плавним та сучасним. Також реалізовано адаптивну структуру сторінки, яка коректно працює як на комп'ютерах, так і на мобільних пристроях.

На сторінці реалізовано систему додавання манги до користувацьких списків. Після натискання кнопки Add to List відкривається випадаюче меню зі списками користувача, отриманими із серверної частини. Система автоматично визначає, чи вже додано комікс до певного списку, та відповідно змінює вигляд інтерфейсу.

Окрему частину сторінки займає система вкладок (Tabs), яка дозволяє перемикатися між главами, коментарями та рекомендаціями схожої манги. Для цього використовується стан activeTab, який визначає активний блок контенту.

У вкладці Chapters реалізовано відображення списку глав через окремий компонент ChapterList. Користувач може переглядати доступні глави та переходити безпосередньо до режиму читання.

Вкладка Comments реалізує систему коментарів користувачів. Користувач може залишати нові коментарі, вподобувати їх або видаляти коментарі за наявності адміністративних прав. Для взаємодії із сервером використовуються API-запити до контролера коментарів. Також реалізовано сортування коментарів за популярністю або новизною.

У вкладці Related реалізовано систему рекомендацій схожої манги. Дані для цієї секції завантажуються із серверної частини на основі спільних жанрів та рейтингу творів. Відображення рекомендацій виконується за допомогою компонентів MangaCard.

Натиснувши на кнопку Read у компоненті ChapterList, користувач переходить на сторінку читання ReadingPage, яка призначена для перегляду графічних матеріалів коміксів у повноекранному режимі. На даній сторінці реалізовано систему навігації між сторінками та главами, бокову панель зі списком глав, індикатор прогресу читання та адаптивний інтерфейс для різних типів пристроїв. Загальний вигляд сторінки читання наведено на рисунку 3.4.

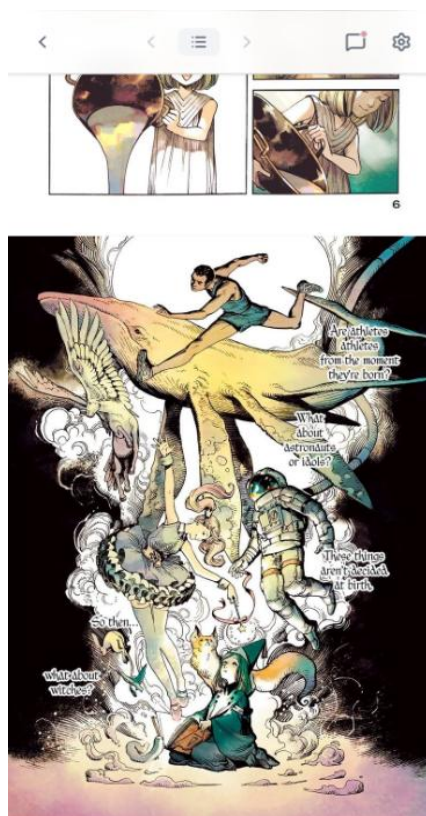


Рисунок 3.4 –Скриншот мобільної версії сторінки ReadingPage

Після відкриття сторінки через `useParams` із URL отримуються ідентифікатор коміксу, назва твору та номер поточної глави. Далі за допомогою `useEffect` виконується завантаження сторінок глави, інформації про мангу та списку доступних глав через API-запити до серверної частини. Отримані дані зберігаються у стані компонента та використовуються для побудови інтерфейсу сторінки.

Для відображення сторінок манги використовується масив `mangaPages`, який містить інформацію про зображення та номери сторінок. Кожна сторінка завантажується окремо через тег `img`, а для оптимізації продуктивності використовується атрибут `loading="lazy"`, що дозволяє відкладати завантаження зображень до моменту їх появи у видимій області екрана.

Завантаження та відображення сторінок здійснюється відповідно до їхніх порядкових номерів, які встановлюються на етапі додавання глави до системи. Під час завантаження зображень кожній сторінці автоматично присвоюється номер, що визначає її позицію у главі. Отримані значення зберігаються у базі даних та надалі використовуються для сортування сторінок під час їх відображення користувачу.

Однією з ключових функцій сторінки є система навігації між сторінками. Для цього реалізовано механізм автоматичного визначення поточної сторінки через `IntersectionObserver`, який відстежує, яке зображення знаходиться у видимій області екрана. На основі цього автоматично оновлюється індикатор прогресу читання та активна сторінка у навігаційній панелі.

У правій частині сторінки реалізовано фіксовану панель навігації між сторінками, яка містить кнопки переходу вперед і назад, індикатор поточної сторінки та інтерактивні позначки сторінок. Користувач може швидко перейти до будь-якої сторінки глави без необхідності прокручування всього контенту.

Для покращення користувацького досвіду на сторінці реалізовано систему приховування елементів керування під час прокручування контенту. Верхня панель `ReadingControls` автоматично зникає при прокручуванні вниз і з'являється при прокручуванні вгору. Це дозволяє збільшити доступний простір для перегляду графічного контенту.

Окрему частину клієнтської частини системи складають адміністративні сторінки, які призначені для керування контентом платформи. Доступ до таких сторінок мають лише користувачі з роллю адміністратора. Основною адміністративною сторінкою є `AddContentPage`, лістинг якої представлений в додатку Б, яка використовується для додавання нових творів, завантаження розділів та ознайомлення з правилами публікації контенту.

Сторінка `AddContentPage` побудована за принципом вкладок. За допомогою стану `activeTab` визначається, який саме розділ адміністративної панелі відображається в поточний момент: форма додавання нової манги, форма завантаження глави або сторінка з правилами публікації. Для перемикання між вкладками використовується бічна панель із кнопками `Add Manga`, `Add Chapter` та `Guidelines`.

Форма `AddMangaForm` призначена для створення нового запису коміксу у системі. У ній адміністратор заповнює основну інформацію про твір: назву, оригінальну та англійську назви, автора, художника, рік випуску, статус, видавця, тип манги, вікове обмеження, опис та жанри. Також у формі реалізовано

завантаження обкладинки з попереднім переглядом зображення. Після натискання кнопки Publish Manga дані збираються у FormData та надсилаються на сервер через HTTP-запит POST до маршруту /api/manga.

Окремо у формі реалізовано роботу з жанрами. Після відкриття форми список жанрів завантажується із серверної частини через запит до /api/genres. Адміністратор може вибрати один або декілька жанрів для нового тайтлу, а також створити новий жанр безпосередньо з форми. Це робить процес додавання контенту більш гнучким і не потребує окремого переходу до іншої сторінки для керування жанрами.

Форма AddChapterForm використовується для додавання нового розділу до вже наявного твору. Спочатку адміністратор знаходить потрібний комікс через поле пошуку. Пошук працює із затримкою 300 мс, що зменшує кількість зайвих запитів до сервера під час введення тексту. Після вибору манги адміністратор вказує номер тому, номер розділу, назву розділу та завантажує зображення сторінок.

Для завантаження сторінок глави використовується множинний вибір файлів. Перед додаванням файлів у стан компонента виконується перевірка їхнього формату: дозволяються лише зображення у форматах PNG, JPG, JPEG та WebP. Адміністратор може бачити список вибраних файлів, їхній розмір та порядок завантаження, а також видаляти окремі сторінки перед публікацією.

Після заповнення форми розділ надсилається на сервер у вигляді FormData, де передаються ідентифікатор твору, номер тому, номер розділу, назва та масив файлів сторінок. Для цього використовується HTTP-запит POST до маршруту /api/Chapter. Такий підхід дозволяє передавати на сервер не лише текстові дані, а й файли зображень у межах одного запиту.

Також в адміністративній частині системи було реалізовано сторінку редагування окремого твору TitleEditPage фрагменти коду якої розміщені в додатку Б. Перехід до цієї сторінки здійснюється зі сторінки конкретної манги за допомогою кнопки Edit, яка доступна лише користувачам з роллю адміністратора.

Дана сторінка призначена для зміни основної інформації про твір, редагування жанрів, оновлення обкладинки, а також додавання, редагування та видалення глав.

Усі зміни, внесені адміністратором, передаються до серверної частини через REST API та після успішного збереження автоматично відображаються в інтерфейсі, що забезпечує актуальність даних та цілісність інформації в системі. Такий підхід дозволяє ефективно організувати процес керування контентом платформи та створює зручне робоче середовище для адміністраторів вебзастосунку.

Після відкриття сторінки `TitleEditPage` за допомогою `useParams` з URL отримується ідентифікатор манги. Далі через `Promise.all` одночасно виконуються запити до серверної частини для отримання даних про тайтл, списку його глав та повного переліку жанрів. Отримані дані зберігаються у станах `titleData`, `chaptersData` та `genresData`, після чого використовуються для заповнення форми редагування.

Для збереження змін основної інформації про мангу використовується метод `handleSaveManga`. У ньому дані збираються у `FormData`, оскільки разом із текстовими полями може передаватися файл нової обкладинки. Після цього виконується HTTP-запит `PUT` до маршруту `/api/Manga/{id}`. Для захисту адміністративної дії у заголовок запиту додається JWT-токен користувача.

Візуально сторінка редагування (рис. 3.5) побудована подібно до сторінки перегляду тайтлу, але замість статичного відображення інформації використовуються поля введення, випадаючі списки та інтерактивні елементи. Компонент `TitleEditHero` дозволяє змінювати обкладинку, назву, оригінальну назву, автора, статус, жанри та опис твору. Для вибору жанрів використовується компонент `GenrePicker`, у якому вже вибрані жанри відображаються окремо, а невибрані можна додати натисканням. Праворуч на сторінці розміщується інформаційна панель `EditableInfoSidebar`, у якій адміністратор може змінювати додаткові параметри тайтлу: автора, художника, статус, рік випуску, видавця, вікове обмеження, тип манги, англійську та альтернативну назви.

Editing: Berserk

Last saved 2 min ago

Cancel

Save Changes

Home

Catalog

Berserk

Edit



Berserk

ベルセルク

By Kentaro Miura

★★★★★ 4.8 (12.5k reviews) | Rank: #5 | Views: 2.4M

Action × Fantasy × Adventure × Drama × Military × Seinen × Comedy + Horror + Romance

+ Shonen + Slice of Life

Guts, a former mercenary now known as the "Black Swordsman," is out for revenge. After a traumatic past and a horrific betrayal by his former leader Griffith, Guts is cursed with a brand that attracts demonic spirits. Armed with his massive sword, the Dragonslayer, and a prosthetic iron arm, he wanders a dark, medieval world, battling monstrous "Apostles" and the God Hand in a desperate struggle against fate itself.

419 / 2990

Chapters

+ Add Chapter

CHAPTER NUMBER	VOLUME	TITLE	RELEASE DATE
1	1	The Black Swordsman	28.05.2026
2		The Brand Volume 1 • 28.05.2026	
3		The Guardians of Desire Volume 1 • 28.05.2026	

Cancel Save

Information

AUTHOR
Kentaro Miura

ARTIST
Kentaro Miura

STATUS
Ongoing

YEAR
1989

PUBLISHER
Young Animal

Рисунок 3.5 – Скріншот сторінки редагування

. Такий поділ дозволяє не перевантажувати основну частину сторінки та зробити редагування зручнішим.

Окремим блоком реалізовано редагування глав через компонент ChapterEditList. У ньому адміністратор може додати нову главу, змінити номер глави, том, назву та дату релізу, а також видалити вже наявну главу. Для створення використовується запит POST, для оновлення – PUT, а для видалення – DELETE,

що відповідає CRUD-підходу REST API. Загальний режим редагування додатково підкреслюється компонентом EditModeBanner, який розміщується у верхній частині сторінки та містить кнопку збереження змін.

Усі основні користувацькі сторінки системи, зокрема головна сторінка, каталог, сторінка тайтлу, режим читання та адміністративні сторінки, були оптимізовані для коректної роботи на різних типах пристроїв – персональних комп'ютерах, планшетах і смартфонах.

Для реалізації адаптивності використовувалися можливості Tailwind CSS та система responsive-класів. Завдяки цьому структура сторінок автоматично змінюється залежно від розміру екрана користувача. Наприклад, багатоколонкові блоки на великих екранах відображаються у вигляді однієї колонки на мобільних пристроях, а навігаційні елементи та бічні панелі адаптуються під сенсорне керування.

Особливо важливою адаптивність стала для сторінки читання коміксу, оскільки значна частина користувачів переглядає графічний контент саме зі смартфонів. Для цього було реалізовано мобільну навігацію між розділами, автоматичне масштабування зображень та оптимізацію елементів керування для невеликих екранів. Аналогічно адміністративні сторінки також адаптуються під мобільні пристрої: бічні панелі переходять у горизонтальне меню, а форми автоматично перебудовуються у вертикальну структуру.

Використання адаптивного дизайну дозволило забезпечити зручну взаємодію користувача із системою незалежно від типу пристрою та роздільної здатності екрана, що є важливою вимогою для сучасних вебзастосунків.

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

Тестування клієнтської частини інформаційно-комп'ютерної системи виконувалося з метою перевірки коректності роботи користувацького інтерфейсу, навігації між сторінками та взаємодії клієнтської частини із сервером. Оскільки

фронтенд був реалізований за допомогою React та TypeScript, основна увага приділялася роботі компонентів, обробці станів, виконанню HTTP-запитів та правильному відображенню отриманих даних.

Під час тестування перевірялася коректність переходів між основними сторінками системи: головною сторінкою, каталогом, сторінкою окремого тайтлу, сторінкою читання, профілем користувача та адміністративними сторінками. Особлива увага приділялася роботі React Router, який забезпечує навігацію між сторінками без повного перезавантаження застосунку.

Також тестувалася робота основних React-компонентів, зокрема навігаційної панелі, карток манги, форм додавання та редагування контенту, списку глав, системи коментарів і сторінки читання. Перевірялося коректне оновлення станів компонентів після взаємодії користувача та правильність відображення даних, отриманих із серверної частини.

Виконувалося тестування форм авторизації, додавання манги, завантаження глав, редагування тайтлів та системи коментарів. Для форм, що працюють із файлами, додатково тестувалося завантаження обкладинок і сторінок манги. За допомогою інструментів розробника перевірялися HTTP-відповіді сервера, структура JSON-даних, передача JWT-токенів у захищених запитах. Для налагодження клієнтської частини використовувалися Chrome DevTools та React Developer Tools. Chrome DevTools застосовувався для аналізу мережевих запитів, перевірки помилок у консолі, структури DOM-дерева та адаптивності інтерфейсу. React Developer Tools дозволяв аналізувати структуру компонентів, їхні стани та пропси, що значно спростило пошук помилок у логіці застосунку.

Перевірка адаптивності інтерфейсу проводилася у responsive-режимі браузера. Сторінки тестувалися на різних розмірах екрана для перевірки коректного відображення елементів інтерфейсу на комп'ютерах, планшетах і мобільних пристроях.

Тестування серверної частини інформаційно-комп'ютерної системи виконувалося з метою перевірки роботи REST API, взаємодії з базою даних, авторизації користувачів та обробки HTTP-запитів від клієнтської частини.

Під час тестування перевірялася робота основних API-маршрутів, зокрема отримання списку манги, завантаження окремого тайтлу, створення та редагування записів, додавання глав, робота зі списками користувачів, системою коментарів та пошуком контенту. Для кожного API-методу перевірялися HTTP-відповіді, структура JSON-даних та відповідність результатів очікуваній логіці роботи системи.

Особлива увага приділялася тестуванню CRUD-операцій REST API. Перевірялися HTTP-методи GET, POST, PUT та DELETE, які використовуються для отримання, створення, оновлення та видалення інформації. Також контролювалася правильність передачі DTO-об'єктів між серверною та клієнтською частинами системи.

Окремо тестувалася система авторизації та автентифікації на основі JWT-токенів. Перевірялися створення токена після входу в систему, передача токена у заголовках HTTP-запитів та обмеження доступу до адміністративних API-методів. Також проводилося тестування взаємодії серверної частини з базою даних через Entity Framework Core. Перевірялися створення зв'язків між сутностями, виконання LINQ-запитів, робота міграцій та збереження даних у Microsoft SQL Server.

Важливу роль у тестуванні серверної частини відігравав Swagger, який використовувався для автоматичної генерації документації REST API та тестування HTTP-запитів. За допомогою Swagger перевірялася робота контролерів, CRUD-операцій, маршрутизації API та структура JSON-відповідей без необхідності запуску клієнтської частини. Це значно прискорило процес налагодження та інтеграції фронтенду з бекендом.

Swagger використовувався як основний інструмент тестування REST API під час розробки серверної частини системи. Через Swagger виконувалися HTTP-запити типу GET, POST, PUT та DELETE, перевірялася робота контролерів, DTO-об'єктів та взаємодія з базою даних. Приклад інтерфейсу Swagger із доступними API-маршрутами наведено на рисунку 3.6.

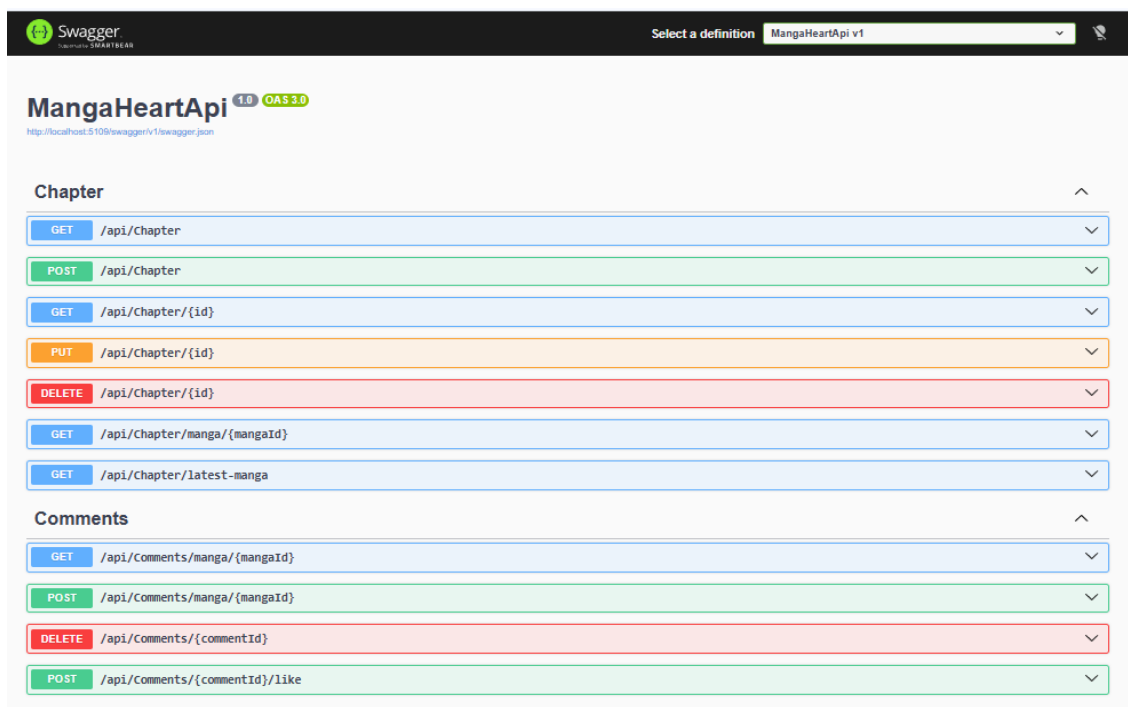


Рисунок 3.6 – Інтерфейс Swagger для тестування REST API

3.3 Захист інформаційно-комп'ютерної системи

Під час розробки інформаційної системи було забезпечено захист даних, контролю доступу до функцій системи та безпечній взаємодії між клієнтською та серверною частинами вебзастосунку. Основними напрямками забезпечення захисту стали реалізація механізмів автентифікації та авторизації користувачів, захист REST API, перевірка вхідних даних та контроль роботи із файловою системою.

Автентифікація користувачів у системі реалізована за допомогою JWT-токенів (JSON Web Token). Після успішного входу користувач отримує токен, який використовується для подальшої взаємодії із захищеними API-маршрутами.

Для доступу до захищених API-маршрутів клієнтська частина передає JWT-токен у заголовок Authorization кожного запиту. Приклад такого запиту наведено у лістингу 3.12. Під час редагування інформації про мангу клієнтська частина надсилає HTTP-запит типу PUT, до якого додається токен у форматі Bearer. Після отримання запиту сервер перевіряє дійсність токена та права користувача, і лише після успішної перевірки дозволяє виконати зміну даних. Такий механізм забезпечує захист адміністративних функцій від несанкціонованого доступу. Токен

передається у заголовках HTTP-запитів та перевіряється серверною частиною під час кожного звернення до API.

Лістинг 3.12 – Передача JWT-токена в захищеному HTTP-запиті

```
const response = await fetch(`${API_BASE}/api/Manga/${id}`, {  
  method: 'PUT',  
  headers: {  
    Authorization: `Bearer ${token}`,  
  },  
  body: formData,  
})
```

кінець лістингу 3.12

У системі реалізовано розподіл ролей користувачів на звичайних користувачів та адміністраторів. Звичайні користувачі мають доступ до перегляду контенту, режиму читання, системи коментарів та власних списків манги. Адміністратори, окрім базових можливостей, отримують доступ до адміністративної панелі, функцій додавання нових тайтлів, завантаження глав, редагування контенту та керування даними системи. На серверній частині адміністративні API-маршрути додатково захищені перевіркою ролі користувача, що запобігає несанкціонованому доступу навіть у випадку спроби прямого звернення до API.

Для захисту REST API використовується JWT Bearer Authentication та CORS-політика. Налаштування CORS дозволяє обмежити перелік джерел, з яких дозволено надсилати HTTP-запити до серверної частини. Це забезпечує контроль взаємодії між клієнтською та серверною частинами застосунку та зменшує ризик небажаних зовнішніх запитів.

Важливою складовою захисту системи є валідація даних, які надходять від користувачів. На серверній частині перевіряються DTO-об'єкти, обов'язкові поля, типи даних та коректність вхідної інформації. Також перевіряються права користувача на виконання певних дій. Це дозволяє запобігти передачі некоректних або неповних даних до серверної частини та бази даних.

Окрему увагу приділено захисту файлової системи. Під час завантаження обкладинок манги та сторінок глав виконується перевірка форматів файлів, а також контроль максимально допустимого розміру завантаження. Для уникнення конфліктів імен використовується генерація унікальних назв файлів, а самі медіафайли зберігаються в окремій директорії серверної частини.

Додатково у серверному застосунку використовується перенаправлення HTTP-запитів на HTTPS-з'єднання. Це забезпечує захищену передачу даних між клієнтською та серверною частинами системи, зокрема під час авторизації користувачів та виконання захищених API-запитів.

Таким чином, у розробленій інформаційній системі реалізовано базові механізми захисту вебзастосунків, які забезпечують контроль доступу до функцій системи, безпечну взаємодію між компонентами та захист даних користувачів і контенту платформи.

3.4 SEO інформаційно-комп'ютерної системи

Під час розробки платформи було враховано основні принципи SEO-оптимізації. Для кожного тайтлу формується унікальна адреса сторінки (URL), яка містить назву твору. Такий підхід робить посилання більш зрозумілими як для користувачів, так і для пошукових систем. Наприклад, замість використання лише ідентифікатора запису сторінка може мати адресу, що містить назву конкретного твору.

Для сторінок використовуються заголовки різних рівнів (h1, h2, h3), що дозволяє пошуковим системам коректно визначати структуру інформації. Крім цього, текстові описи творів, жанри, назви глав та інші метадані можуть використовуватися для формування індексованого контенту.

Для підвищення швидкодії вебзастосунку реалізовано відкладене завантаження зображень (lazy loading), що дозволяє зменшити обсяг даних, які завантажуються під час відкриття сторінки. Швидкість завантаження є одним із

факторів ранжування сайтів у пошукових системах, тому оптимізація графічного контенту позитивно впливає на SEO-показники ресурсу.

Важливим фактором також є адаптивність інтерфейсу. Платформа коректно працює на персональних комп'ютерах, планшетах та мобільних пристроях. Оскільки пошукові системи враховують мобільну оптимізацію під час ранжування сайтів, адаптивний дизайн сприяє покращенню позицій вебресурсу у результатах пошуку.

3.5 Розгортання та хостинг інформаційної системи

Після завершення розробки та тестування інформаційної системи було виконано її розгортання у хмарному середовищі. Використання хмарних сервісів дозволило забезпечити доступність вебзастосунку через мережу Інтернет, спростити процес оновлення програмного забезпечення та централізувати керування серверною інфраструктурою.

Оскільки розроблена система складається з клієнтської, серверної частин та бази даних, для їх розміщення були використані окремі сервіси хмарного хостингу. Клієнтська частина, реалізована за допомогою React та TypeScript, була розгорнута на платформі Vercel. Даний сервіс спеціалізується на розміщенні сучасних вебзастосунків та забезпечує автоматичне розгортання проєкту безпосередньо з репозиторію GitHub. Після внесення змін до репозиторію Vercel автоматично виконує процес збірки та публікації нової версії клієнтської частини.

Серверна частина, розроблена на платформі ASP.NET Core, а також база даних Microsoft SQL Server були розміщені на платформі Railway. Для серверного застосунку було налаштовано автоматичне розгортання з GitHub, що дозволяє швидко оновлювати API після внесення змін до вихідного коду. Крім того, Railway забезпечує зберігання конфігураційних параметрів, змінних середовища та рядків підключення до бази даних.

Під час розгортання було налаштовано взаємодію між клієнтською та серверною частинами через REST API. Для забезпечення коректного обміну

даними було виконано налаштування CORS-політики та конфігурацію адрес віддалених сервісів. Також було забезпечено підтримку HTTPS-з'єднання, що дозволяє захистити передачу даних між користувачем і сервером.

Використання Vercel та Railway дозволило реалізувати повністю хмарну архітектуру системи, у якій фронтенд, бекенд та база даних працюють незалежно один від одного та можуть оновлюватися окремо.

ВИСНОВКИ

У кваліфікаційній роботі було досягнуто поставлену мету, яка полягала у Розробка онлайн-платформи для читання і зберігання коміксів використовуючи React.

У процесі виконання роботи було проаналізовано сучасні онлайн-платформи для читання манги та коміксів, серед яких MangaDex, MangaPlus та manga.in.ua. Проведений аналіз дозволив визначити їхні переваги та недоліки, а також сформувати перелік функціональних можливостей, які доцільно реалізувати у власній інформаційній системі.

Було визначено функціональні та нефункціональні вимоги до програмного забезпечення. Сформовано перелік основних можливостей системи, серед яких перегляд каталогу творів, режим читання, система коментарів, персональні списки користувачів, авторизація та адміністрування контенту. Також визначено вимоги до продуктивності, безпеки, адаптивності та зручності використання вебзастосунку.

Обґрунтовано вибір технологій та інструментальних засобів розробки. Для реалізації клієнтської частини було використано React та TypeScript, для серверної частини – платформу ASP.NET Core, для роботи з базою даних – Microsoft SQL Server та Entity Framework Core. Використання зазначених технологій дозволило створити сучасний вебзастосунок із розподілом клієнтської та серверної логіки.

У роботі виконано проєктування архітектури системи, структури бази даних та інформаційних потоків. Побудовано UML-діаграми, розроблено структуру бази даних та визначено взаємозв'язки між основними сутностями системи. Це дозволило сформувати цілісну архітектуру програмного забезпечення та забезпечити ефективну взаємодію між його компонентами.

Було реалізовано клієнтську та серверну частини вебзастосунку. Створено каталог творів, сторінку окремого тайтлу, режим читання, систему коментарів, персональні списки користувачів та адміністративну панель для керування

контентом. Для взаємодії між компонентами системи реалізовано REST API, що забезпечує обмін даними між клієнтською та серверною частинами.

У системі реалізовано механізми автентифікації та авторизації на основі JWT-токенів, розподіл ролей користувачів, захист адміністративних функцій, валідацію вхідних даних та використання HTTPS-з'єднання. Це дозволило забезпечити безпечну взаємодію користувачів із платформою та захистити дані від несанкціонованого доступу.

Проведено тестування та налагодження клієнтської і серверної частин вебзастосунку. Перевірено коректність роботи основних функцій системи, взаємодію між фронтендом і бекендом, роботу REST API, механізмів авторизації та операцій із базою даних. У ході тестування було виявлено та усунено помилки, що дозволило забезпечити стабільну роботу програмного забезпечення.

Після завершення розробки виконано розгортання системи у хмарному середовищі. Клієнтська частина була розміщена на платформі Vercel, а серверна частина та база даних – на платформі Railway, що забезпечило доступність вебплатформи через мережу Інтернет.

Таким чином, усі поставлені завдання було виконано у повному обсязі, а розроблена онлайн-платформа може бути використана для читання, зберігання та управління графічним контентом, забезпечуючи користувачам сучасний та зручний інструмент для роботи з коміксами та мангою.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. DataLife engine (DLE) official site. DataLife Engine (DLE) Official site. URL: <https://dle-news.com/> (date of access: 30.04.2026).
2. What is Client-Server Architecture?. URL: <https://www.theknowledgeacademy.com/>.
<https://www.theknowledgeacademy.com/blog/client-server-architecture/> (date of access: 20.05.2026).
3. Angular. URL: <https://angular.dev/guide/> (date of access: 21.05.2026).
4. Початок роботи – React. React – JavaScript-бібліотека для створення користувацьких інтерфейсів. URL: <https://uk.legacy.reactjs.org/docs/getting-started.html> (дата звернення: 22.05.2026).
5. Схема роботи Virtual DOM. medium.com. URL: https://miro.medium.com/v2/resize:fit:720/format:webp/1*IR0pcWncnQ4J-Z_KOZrNrQ.png (дата звернення: 22.05.2026).
6. Ефективність застосування бібліотеки react. Інформаційні технології та суспільство. Наукова періодика Міжрегіональної Академії управління персоналом. URL: <https://journals.maup.com.ua/index.php/it/article/view/2096> (дата звернення: 21.05.2026).
7. Documentation - TypeScript for JavaScript Programmers. TypeScript: JavaScript With Syntax For Types. URL: <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html> (date of access: 23.05.2026).
8. Tailwind CSS Documentation. Tailwind CSS - Rapidly build modern websites without ever leaving your HTML. URL: <https://tailwindcss.com/docs/> (date of access: 23.05.2026).
9. Team A. E. Pros and Cons of Node.js Web App Development. AltexSoft. URL: <https://www.altexsoft.com/blog/the-good-and-the-bad-of-node-js-web-app-development/> (date of access: 24.05.2026).

10. Django – Вікіпедія. Вікіпедія. URL: <https://uk.wikipedia.org/wiki/Django> (дата звернення: 24.05.2026).
11. ASP.NET documentation. Microsoft Learn: Build with answers in reach. URL: <https://learn.microsoft.com/en-us/aspnet/core/> (date of access: 25.05.2026).
12. Entity Framework documentation hub. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/ef/> (date of access: 25.05.2026).
13. ORM (Object-Relational Mapping). medium.com. URL: https://medium.com/@amit.s_15142/orm-object-relational-mapping-ef49916a1a84 (date of access: 25.05.2026).
14. Schemas and Types. GraphQL. GraphQL. The query language for modern APIs. URL: <https://graphql.org/learn/schema/> (date of access: 25.05.2026).
15. O'Riordan M. WebSocket API: Events, Methods & Properties. WebSocket.org. URL: <https://websocket.org/reference/websocket-api/> (date of access: 25.05.2026).
16. What is RESTful API? - RESTful API Explained - AWS. Amazon Web Services, Inc. URL: <http://aws.amazon.com/what-is/restful-api/> (date of access: 25.05.2026).
17. Microsoft. Documentation for Visual Studio Code. Visual Studio Code - The open source AI code editor. Your home for multi-agent development. URL: <https://code.visualstudio.com/docs> (date of access: 25.05.2026).

ДОДАТКИ

Додаток А

Фрагменти програмного коду серверної частини MangaHeart

Лістинг А.1 – MangaController.cs

```

[ApiController]
[Route("api/[controller]")]
public class MangaController : ControllerBase
{
    private readonly MangaDbContext _context;
    private readonly IMapper _mapper;
    private readonly MangaService _mangaService;

    public MangaController(
        MangaDbContext context,
        IMapper mapper,
        MangaService mangaService)
    {
        _context = context;
        _mapper = mapper;
        _mangaService = mangaService;
    }

    [HttpGet]
    public async Task<ActionResult<IEnumerable<MangaDto>>> GetAll()
    {
        var mangas = await _context.Mangas
            .Include(m => m.Type)
            .Include(m => m.AgeRating)
            .ToListAsync();

        return Ok(_mapper.Map<List<MangaDto>>(mangas));
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<MangaDto>> Get(int id)
    {
        var manga = await _context.Mangas
            .Include(m => m.Type)
            .Include(m => m.AgeRating)
            .Include(m => m.MangaGenres)
            .ThenInclude(mg => mg.Genre)
            .FirstOrDefaultAsync(m => m.Id == id);

        if (manga == null)
            return NotFound();

        return Ok(_mapper.Map<MangaDto>(manga));
    }

    [Authorize(Roles = "Admin")]
    [HttpPost]
    public async Task<ActionResult<MangaDto>> Create(
        [FromBody] CreateMangaDto dto)
    {
        var result = await _mangaService.CreateAsync(dto);
    }
}

```

```

        return CreatedAtAction(
            nameof(Get),
            new { id = result.Id },
            result);
    }

    [Authorize(Roles = "Admin")]
    [HttpPut("{id}")]
    public async Task<IActionResult> Update(
        int id,
        [FromForm] UpdateMangaDto dto)
    {
        var result = await _mangaService.UpdateAsync(id, dto);

        if (!result)
            return NotFound();

        return NoContent();
    }

    [Authorize(Roles = "Admin")]
    [HttpDelete("{id}")]
    public async Task<IActionResult> Delete(int id)
    {
        var manga = await _context.Mangas.FindAsync(id);

        if (manga == null)
            return NotFound();

        _context.Mangas.Remove(manga);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    [Authorize(Roles = "Admin")]
    [HttpPost("upload")]
    public async Task<IActionResult> UploadFile(IFormFile file)
    {
        var url = await _mangaService.UploadFileAsync(file);

        return Ok(new { url });
    }

    [HttpGet("search")]
    public async Task<IActionResult> Search(string query)
    {
        var result = await _mangaService.SearchAsync(query);

        return Ok(result);
    }

    [HttpGet("{id}/related")]
    public async Task<ActionResult<IEnumerable<object>>> GetRelatedManga(
        int id)
    {
        var result = await _mangaService.GetRelatedMangaAsync(id);
    }

```

```

        return Ok(result);
    }
}

```

кінець лістингу A.1

Лістинг A.2 – MangaService.cs

```

public class MangaService
{
    private readonly MangaDbContext _context;
    private readonly IMapper _mapper;

    public MangaService(
        MangaDbContext context,
        IMapper mapper)
    {
        _context = context;
        _mapper = mapper;
    }

    public async Task<IEnumerable<MangaSearchDto>>
        SearchMangaAsync(string query)
    {
        query = $"%{query.Trim()}%";

        return await _context.Mangas
            .Where(m =>
                EF.Functions.Like(m.Title, query) ||
                EF.Functions.Like(m.EnglishTitle, query) ||
                EF.Functions.Like(m.Author, query) ||
                m.MangaGenres.Any(g =>
                    EF.Functions.Like(g.Genre.Name, query)))
            .OrderByDescending(m => m.Rate)
            .Select(m => new MangaSearchDto
            {
                Id = m.Id,
                Title = m.Title,
                Rate = m.Rate
            })
            .ToListAsync();
    }

    public async Task<List<MangaDto>>
        GetMangaByGenresAsync(List<string> genreNames)
    {
        var genreCount = genreNames.Distinct().Count();

        return await _context.Mangas
            .Where(m => m.MangaGenres
                .Where(g => genreNames.Contains(g.Genre.Name))
                .Select(g => g.Genre.Name)
                .Distinct()
                .Count() == genreCount)
            .Select(m => new MangaDto
            {
                Id = m.Id,
                Title = m.Title
            })

```

```

        .ToListAsync();
    }

    public async Task<MangaDto>
    CreateAsync(CreateMangaDto dto)
    {
        string cleanTitle =
            string.Concat(dto.Title.Split(
                Path.GetInvalidFileNameChars()));

        Directory.CreateDirectory(
            Path.Combine(@"E:\MangaData", cleanTitle));

        string fileName =
            Guid.NewGuid() +
            Path.GetExtension(dto.CoverFile.FileName);

        var manga = _mapper.Map<Manga>(dto);

        _context.Mangas.Add(manga);

        await _context.SaveChangesAsync();

        return _mapper.Map<MangaDto>(manga);
    }

    public async Task<bool>
    UpdateAsync(int id, UpdateMangaDto dto)
    {
        var manga = await _context.Mangas
            .Include(m => m.MangaGenres)
            .FirstOrDefaultAsync(m => m.Id == id);

        if (manga == null)
            return false;

        manga.Title = dto.Title;
        manga.Description = dto.Description;
        manga.Author = dto.Author;
        manga.Year = dto.Year;
        manga.Rate = dto.Rate;

        _context.MangaGenres.RemoveRange(
            manga.MangaGenres);

        await _context.SaveChangesAsync();

        return true;
    }
}

```

кінець лістингу А.

Лістинг А.3– Program.cs

```

using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity;
using MangaHeartApi.Data.Seed;
using MangaHeartApi.Data;

```

```

using MangaHeartApi.Models;
using Microsoft.Extensions.FileProviders;
using Microsoft.AspNetCore.Http.Features;
using MangaHeartApi.Services;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;
using System.Security.Claims;

var builder = WebApplication.CreateBuilder(args);

builder.WebHost.ConfigureKestrel(options =>
{
    options.Limits.MaxRequestBodySize = 200 * 1024 * 1024; // 200 MB
});

builder.Services.Configure<FormOptions>(options =>
{
    options.ValueLengthLimit = int.MaxValue;
    options.MultipartBodyLengthLimit = 200 * 1024 * 1024; // 200 MB
    options.MultipartHeadersLengthLimit = int.MaxValue;
});
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = false,
            ValidateAudience = false,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]!)
            ),
            RoleClaimType = ClaimTypes.Role
        };
    });

builder.Services.AddAuthorization();
builder.Services.AddScoped<MangaService>();
builder.Services.AddScoped<ChapterService>();
builder.Services.AddScoped<UserService>();
builder.Services.AddOpenApi();
builder.Services.AddDbContext<MangaDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
builder.Services.AddSwaggerGen();
builder.Services.AddControllers();
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowFrontend",
        policy =>
        {
            policy.WithOrigins("http://localhost:5173")
                .AllowAnyHeader()
                .AllowAnyMethod();
        });
});

```

```
var app = builder.Build();
var mangaPath = @"E:\MangaData";

if (!Directory.Exists(mangaPath)) Directory.CreateDirectory(mangaPath);

app.UseStaticFiles(new StaticFileOptions
{
    FileProvider = new PhysicalFileProvider(mangaPath),
    RequestPath = "/manga-files"
});
app.UseCors("AllowFrontend");
app.UseSwagger();
app.UseSwaggerUI();
app.MapControllers();

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}
app.UseHttpsRedirection();
app.Run();
```

кінець лістингу A.3

Додаток Б

Лістинг сторінок вебдодатку MangaHear

Лістинг Б.1 – CatalogPage.tsx

```

export function CatalogPage() {
  const [searchParams, setSearchParams] = useSearchParams()

  const [selectedGenres, setSelectedGenres] = useState<string[]>([])
  const [selectedStatus, setSelectedStatus] = useState('All')
  const [genres, setGenres] = useState<any[]>([])
  const [minRating, setMinRating] = useState(0)
  const [mangaList, setMangaList] = useState<any[]>([])
  const [loading, setLoading] = useState(true)

  useEffect(() => {
    fetch('/api/Genres')
      .then((res) => res.json())
      .then((data) => setGenres(data))
  }, [])

  useEffect(() => {
    const genreFromUrl = searchParams.get('genres')

    if (genreFromUrl) {
      setSelectedGenres([genreFromUrl])
    } else {
      setSelectedGenres([])
    }
  }, [searchParams])

  useEffect(() => {
    const fetchMangaData = async () => {
      setLoading(true)

      let url = '/api/manga'

      if (selectedGenres.length > 0) {
        url = `/api/Manga/manga/genre/${selectedGenres[0]}`
      }

      const response = await fetch(url)
      const data = await response.json()

      setMangaList(data)
      setLoading(false)
    }

    fetchMangaData()
  }, [selectedGenres])

  const toggleGenre = (genreName: string) => {
    if (selectedGenres.includes(genreName)) {
      setSelectedGenres([])
      setSearchParams({})
    } else {
      setSelectedGenres([genreName])
    }
  }
}

```

```

    setSearchParams({ genres: genreName })
  }
}
const displayManga = mangaList.filter((manga) => {
  const rating = manga.rate ?? manga.rating
  return (
    rating >= minRating &&
    (selectedStatus === 'All' ||
      manga.status === selectedStatus)
  )
})
return (
  <>
    <CatalogFilters
      genres={genres}
      selectedGenres={selectedGenres}
      toggleGenre={toggleGenre}
      selectedStatus={selectedStatus}
      setSelectedStatus={setSelectedStatus}
      minRating={minRating}
      setMinRating={setMinRating}
    />
    {loading ? (
      <div>Loading...</div>
    ) : (
      displayManga.map((manga) => (
        <motion.div key={manga.id}>
          <MangaCard {...manga} />
        </motion.div>
      ))
    )}
  )}
}

```

кінець лістингу Б.1

Лістинг Б.2 – MangaTitlePage.tsx

```

export function MangaTitlePage() {
  const { id } = useParams()
  const navigate = useNavigate()

  const [activeTab, setActiveTab] = useState('chapters')
  const [showListMenu, setShowListMenu] = useState(false)
  const [userLists, setUserLists] = useState<UserList[]>([])
  const [addedLists, setAddedLists] = useState<number[]>([])
  const [comments, setComments] = useState<Comment[]>([])
  const [commentBody, setCommentBody] = useState('')
  const [commentSort, setCommentSort] = useState<'top' | 'newest'>('top')
  const [manga, setManga] = useState<any>(null)
  const [relatedManga, setRelatedManga] = useState<any[]>([])
  const [loading, setLoading] = useState(true)

  useEffect(() => {
    const fetchData = async () => {
      const [mangaResponse, relatedResponse] = await Promise.all([
        fetch(`/api/manga/${id}`),
        fetch(`/api/manga/${id}/related`),
      ])

      const mangaData = await mangaResponse.json()
    }
  })
}

```

```

    const relatedData = await relatedResponse.json()

    setManga(mangaData)
    setRelatedManga(relatedData)

    await loadUserLists()
    await loadAddedLists(mangaData.id)
    await loadComments(mangaData.id)

    setLoading(false)
  }

  fetchData()
}, [id])

const loadUserLists = async () => {
  const token = localStorage.getItem('token')

  const response = await fetch('/api/List/lists', {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  })

  setUserLists(await response.json())
}

const loadAddedLists = async (mangaId: number) => {
  const token = localStorage.getItem('token')

  const response = await fetch(`/api/List/check-manga/${mangaId}`, {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  })

  setAddedLists(await response.json())
}

const toggleList = async (listId: number) => {
  const token = localStorage.getItem('token')
  const alreadyAdded = addedLists.includes(listId)

  await fetch(`/api/List/${listId}/manga/${manga.id}`, {
    method: alreadyAdded ? 'DELETE' : 'POST',
    headers: {
      Authorization: `Bearer ${token}`,
    },
  })

  setAddedLists((prev) =>
    alreadyAdded
      ? prev.filter((id) => id !== listId)
      : [...prev, listId]
  )
}

const loadComments = async (mangaId: number) => {
  const token = localStorage.getItem('token')

```

```

    const response = await fetch(`/api/Comments/manga/${mangaId}`, {
      headers: token ? { Authorization: `Bearer ${token}` } : {},
    })

    setComments(await response.json())
  }

  const addComment = async () => {
    const token = localStorage.getItem('token')

    await fetch(`/api/Comments/manga/${manga.id}`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        Authorization: `Bearer ${token}`,
      },
      body: JSON.stringify({
        body: commentBody,
      }),
    })

    setCommentBody('')
    await loadComments(manga.id)
  }

  const toggleCommentLike = async (commentId: number) => {
    const token = localStorage.getItem('token')

    await fetch(`/api/Comments/${commentId}/like`, {
      method: 'POST',
      headers: {
        Authorization: `Bearer ${token}`,
      },
    })

    await loadComments(manga.id)
  }

  const deleteComment = async (commentId: number) => {
    const token = localStorage.getItem('token')

    await fetch(`/api/Comments/${commentId}`, {
      method: 'DELETE',
      headers: {
        Authorization: `Bearer ${token}`,
      },
    })

    await loadComments(manga.id)
  }

  const sortedComments = [...comments].sort((a, b) => {
    if (commentSort === 'newest') {
      return (
        new Date(b.createdAt).getTime() -
        new Date(a.createdAt).getTime()
      )
    }
  })

```

```

    return b.likesCount - a.likesCount
  })

  if (loading) {
    return <div>Loading...</div>
  }

  return (
    <div>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/catalog">Catalog</Link>
        <span>{manga.title}</span>
      </nav>

      <motion.section>
        <img src={manga.coverImagePath} alt={manga.title} />

        <h1>{manga.title}</h1>
        <p>{manga.originalTitle}</p>
        <p>by {manga.author}</p>

        <div>
          {manga.genres.map((genre: string) => (
            <GenreBadge key={genre} label={genre} />
          ))}
        </div>

        <p>{manga.description}</p>

        <button
          onClick={() =>
            navigate(`/manga/${manga.id}/${manga.title}/read/1`)
          }
        >
          Read First Chapter
        </button>

        <button onClick={() => setShowListMenu(!showListMenu)}>
          {addedLists.length > 0
            ? `In ${addedLists.length} List`
            : 'Add to List'}
        </button>

        {showListMenu && (
          <div>
            {userLists.map((list) => (
              <button
                key={list.id}
                onClick={() => toggleList(list.id)}
              >
                {list.name}
              </button>
            ))}
          </div>
        )}
      </motion.section>
    </div>
  )
}

```

```

<div>
  {[ 'chapters', 'comments', 'related'].map((tab) => (
    <button
      key={tab}
      onClick={() => setActiveTab(tab)}
    >
      {tab}
    </button>
  )]}
</div>

{activeTab === 'chapters' && (
  <ChapterList
    mangaId={manga.id}
    mangaTitle={manga.title}
  />
)}

{activeTab === 'comments' && (
  <div>
    <button onClick={() => setCommentSort('top')}>
      Top
    </button>

    <button onClick={() => setCommentSort('newest')}>
      Newest
    </button>

    <textarea
      value={commentBody}
      onChange={(e) => setCommentBody(e.target.value)}
    />

    <button onClick={addComment}>
      Send
    </button>

    {sortedComments.map((comment) => (
      <div key={comment.id}>
        <b>{comment.userName}</b>
        <p>{comment.body}</p>

        <button onClick={() => toggleCommentLike(comment.id)}>
          Like {comment.likesCount}
        </button>

        <button onClick={() => deleteComment(comment.id)}>
          Delete
        </button>
      </div>
    )]}
  </div>
)}

{activeTab === 'related' && (
  <div>
    {relatedManga.map((manga) => (
      <MangaCard key={manga.id} {...manga} />
    ))}
  </div>
)}

```

```

        </div>
      )}

      <MangaInfoSidebar mangaId={id || ''} />
    </div>
  )
}

```

кінець лістингу Б.1

Лістинг Б.3 – ReadingPage.tsx

```

interface Page {
  id: number
  imageUrl: string
  pageNumber: number
}

interface Chapter {
  id: number
  number: number
  title: string
}

export function ReadingPage() {
  const { id, title, chapterNumber } = useParams<{
    id: string
    title: string
    chapterNumber: string
 }>()

  const navigate = useNavigate()
  const pageRefs = useRef<(HTMLDivElement | null)[]>([])

  const [currentPage, setCurrentPage] = useState(0)
  const [showControls, setShowControls] = useState(true)
  const [lastScrollY, setLastScrollY] = useState(0)
  const [mangaPages, setMangaPages] = useState<Page[]>([])
  const [chaptersData, setChaptersData] = useState<Chapter[]>([])
  const [mangaData, setMangaData] = useState<any>(null)

  useEffect(() => {
    fetch(`/api/Page/manga/${id}/chapter/${chapterNumber}`)
      .then((response) => response.json())
      .then((data) => setMangaPages(data))

    fetch(`/api/Chapter/manga/${id}`)
      .then((response) => response.json())
      .then((data) => setChaptersData(data))

    fetch(`/api/manga/${id}`)
      .then((response) => response.json())
      .then((data) => setMangaData(data))
  }, [id, chapterNumber])

  useEffect(() => {
    const observer = new IntersectionObserver((entries) => {
      entries.forEach((entry) => {
        if (!entry.isIntersecting) return

```

```

    const index = pageRefs.current.findIndex(
      (ref) => ref === entry.target
    )

    if (index !== -1) {
      setCurrentPage(index)
    }
  })
})

pageRefs.current.forEach((ref) => {
  if (ref) observer.observe(ref)
})

return () => observer.disconnect()
}, [mangaPages])

useEffect(() => {
  const handleScroll = () => {
    const currentScrollY = window.scrollY

    setShowControls(
      !(currentScrollY > lastScrollY && currentScrollY > 100)
    )

    setLastScrollY(currentScrollY)
  }

  window.addEventListener('scroll', handleScroll)

  return () => window.removeEventListener('scroll', handleScroll)
}, [lastScrollY])

const scrollToPage = (index: number) => {
  pageRefs.current[index]?.scrollIntoView({
    behavior: 'smooth',
    block: 'center',
  })
}

const handlePrevPage = () => {
  if (currentPage > 0) {
    scrollToPage(currentPage - 1)
  }
}

const handleNextPage = () => {
  if (currentPage < mangaPages.length - 1) {
    scrollToPage(currentPage + 1)
  }
}

const handlePrevChapter = () => {
  navigate(`/manga/${id}/${title}/read/${Number(chapterNumber) - 1}`)
  window.scrollTo(0, 0)
}

const handleNextChapter = () => {

```

```

    navigate(`/manga/${id}/${title}/read/${Number(chapterNumber) + 1}`)
    window.scrollTo(0, 0)
  }

  return (
    <div>
      {showControls && (
        <ReadingControls
          mangaId={id}
          mangaTitle={mangaData?.title}
          currentChapter={Number(chapterNumber)}
          totalChapters={chaptersData.length}
          onPrev={handlePrevChapter}
          onNext={handleNextChapter}
        />
      )}

      <aside>
        {chaptersData.map((chapter) => (
          <button
            key={chapter.id}
            onClick={() =>
              navigate(`/manga/${id}/${title}/read/${chapter.number}`)
            }
          >
            Chapter {chapter.number}: {chapter.title}
          </button>
        ))}
      </aside>

      <main>
        <div>
          Page {currentPage + 1} of {mangaPages.length}
        </div>

        {mangaPages.map((page, index) => (
          <div
            key={page.id}
            ref={(element) => {
              pageRefs.current[index] = element
            }}
          >
            <img
              src={page.imageUrl}
              alt={`Page ${page.pageNumber}`}
              loading="lazy"
            />
          </div>
        ))}

        <button onClick={handlePrevPage}>Previous Page</button>
        <button onClick={handleNextPage}>Next Page</button>

        <button onClick={handlePrevChapter}>Previous Chapter</button>
        <button onClick={handleNextChapter}>Next Chapter</button>
      </main>
    </div>
  )
}

```

кінець лістингу Б.3

Лістинг Б.4 – AddContentPage.tsx

```

export function AddContentPage() {
  const [activeTab, setActiveTab] = useState<
    'manga' | 'chapter' | 'guidelines'
  >('manga')

  return (
    <div>
      <aside>
        <SidebarButton
          label="Add Manga"
          isActive={activeTab === 'manga'}
          onClick={() => setActiveTab('manga')}
        />

        <SidebarButton
          label="Add Chapter"
          isActive={activeTab === 'chapter'}
          onClick={() => setActiveTab('chapter')}
        />

        <SidebarButton
          label="Guidelines"
          isActive={activeTab === 'guidelines'}
          onClick={() => setActiveTab('guidelines')}
        />
      </aside>

      <main>
        <h1>
          {activeTab === 'manga' && 'Add New Series'}
          {activeTab === 'chapter' && 'Upload Chapter'}
          {activeTab === 'guidelines' && 'Content Guidelines'}
        </h1>

        {activeTab === 'manga' && <AddMangaForm />}

        {activeTab === 'chapter' && (
          <AddChapterForm setActiveTab={setActiveTab} />
        )}

        {activeTab === 'guidelines' && (
          <div>
            <h3>Submission Guidelines</h3>
            <ul>
              <li>Ensure you have permission to upload the content.</li>
              <li>Do not upload duplicate chapters or series.</li>
              <li>High quality images are preferred.</li>
              <li>Adult content must be correctly flagged.</li>
            </ul>
          </div>
        )}
      </main>
    </div>
  )
}

```

```

}

function SidebarButton({
  label,
  isActive,
  onClick,
}): {
  label: string
  isActive: boolean
  onClick: () => void
}) {
  return (
    <button onClick={onClick}>
      {label}
    </button>
  )
}

```

кінець лістингу Б.4

Лістинг Б.5 – Основна структура сторінки TitleEditPage.tsx

```

export function TitleEditPage() {
  const { id } = useParams()

  const [titleData, setTitleData] = useState<TitleData | null>(null)
  const [chaptersData, setChaptersData] = useState<Chapter[]>([])
  const [genresData, setGenresData] = useState<{ id: number; name: string }[]>([])

  useEffect(() => {
    const fetchData = async () => {
      const [titleResponse, chaptersResponse, genresResponse] =
        await Promise.all([
          fetch(`/api/manga/${id}`),
          fetch(`/api/Chapter/manga/${id}`),
          fetch(`/api/genres`),
        ])

      setTitleData(await titleResponse.json())
      setChaptersData(await chaptersResponse.json())
      setGenresData(await genresResponse.json())
    }

    fetchData()
  }, [id])

  const handleSaveManga = async () => {
    if (!titleData) return

    const formData = new FormData()

    formData.append('Title', titleData.title)
    formData.append('EnglishTitle', titleData.englishTitle)
    formData.append('OriginalTitle', titleData.originalTitle)
    formData.append('Author', titleData.author)
    formData.append('Description', titleData.description)

    const selectedGenreIds = genresData
      .filter((genre) => titleData.genres.includes(genre.name))

```

```

        .map((genre) => genre.id)

selectedGenreIds.forEach((id) =>
  formData.append('GenreIds', String(id))
)

const token = localStorage.getItem('token')

await fetch(`/api/Manga/${id}`, {
  method: 'PUT',
  headers: {
    Authorization: `Bearer ${token}`,
  },
  body: formData,
})
}

return (
  <>
    <EditModeBanner
      title={titleData?.title}
      onSave={handleSaveManga}
    />

    <TitleEditHero
      titleData={titleData}
      setTitleData={setTitleData}
      genresList={genresData}
    />

    <ChapterEditList
      mangaId={Number(id)}
      chapters={chaptersData}
      setChapters={setChaptersData}
    />

    <EditableInfoSidebar
      titleData={titleData}
      setTitleData={setTitleData}
    />
  </>
)
}

```

кінець лістингу Б.5