Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалавр
(ступінь вищої освіти)

на тему СТВОРЕННЯ ПРОГРАМНОЇ СИСТЕМИ ДЛЯ КНИЖКОВОГО МАГАЗИНУ
DESIGN OF A SOFTWARE SYSTEM FOR BOOKSTORE MANAGEMENT

Виконав(ла): студент(ка) 4 курсу, групи КНТ-212
Спеціальності 122 Комп'ютерні науки
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерні науки

СИЛАХІН Д.Р.
(ПРІЗВИЩЕ та ініціали)
Керівник ГОЛУБ Т.В.
(ПРІЗВИЩЕ та ініціали)
Рецензент БАБЕНКО Н.В.
(ПРІЗВИЩЕ та ініціали)

2026

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 122 Комп'ютерні науки
(код і найменування)
Освітня програма (спеціалізація) Комп'ютерні науки
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
_____Сергій СУББОТІН
"____" _____2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

СИЛАХІНА Дмитра Романовича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Створення програмної системи для книжкового магазину. Design of a Software System for Bookstore Management

керівник проєкту (роботи) к.т.н., доцент, ГОЛУБ Тетяна Василівна,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від " 07 " квітня 2026 року № 139
2. Строк подання студентом проєкту (роботи) 10 червня 2026 року
3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної галузі онлайн-продажу книжкової продукції та обґрунтування проєктних рішень. 2. Проєктування архітектури та UML-моделювання вебсайту продажу книг. 3. Реалізація програмного забезпечення вебсайту продажу книг. 4. Тестування та оцінювання якості програмного забезпечення.
5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ГОЛУБ Т.В., доцент		
Нормоконтроль	БЄЛОВА А.В., асистент		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи	1 тиждень	Завдання, ТЗ
2	Аналіз ринку електронної комерції книжкової продукції та порівняльний аналіз існуючих платформ (Yakaboo, Rozetka, Amazon Books)	1 тиждень	Розділ 1
3	Визначення функціональних та нефункціональних вимог; обґрунтування вибору технологічного стеку (React.js/Next.js, Node.js/Express, PostgreSQL, Redis)	2 тиждень	Розділ 1
4	Проєктування трирівневої архітектури, реляційної схеми БД та UML-моделювання (діаграми компонентів, класів, послідовностей, станів)	2 тиждень	Розділ 2
5	Проєктування REST API та інтерфейсу користувача за принципом атомарного дизайну	3 тиждень	Розділ 2
6	Реалізація серверної частини на Node.js/Express з шарами Controller–Service–Repository та повнотекстовим пошуком PostgreSQL (tsvector, GIN)	3-4 тиждень	Розділ 3
7	Реалізація клієнтської частини на React.js/Next.js з Redux Toolkit, JWT-автентифікації та адміністративної панелі	4 тиждень	Розділ 3
8	Модульне (Jest), інтеграційне (Supertest), наскрізне (Playwright) та навантажувальне (k6) тестування	5 тиждень	Розділ 4
9	Оформлення пояснювальної записки та документів до неї	6 тиждень	Додатки
10	Нормоконтроль та рецензування	7 тиждень	
11	Захист роботи	8 тиждень	

Студент(ка) _____ Дмитро СИЛАХІН
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Тетяна ГОЛУБ
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 121 с., 19 табл., 38 рис., 2 дод., 30 джерел.

ВЕБСАЙТ ПРОДАЖУ КНИГ, ЕЛЕКТРОННА КОМЕРЦІЯ, REST API, REACT, NEXT.JS, NODE.JS, POSTGRESQL, REDIS.

Об'єкт дослідження – процес побудови масштабованих програмних систем електронної комерції для роздрібного продажу товарів через мережу Інтернет.

Предмет дослідження – методи та програмні засоби реалізації вебсайту для продажу книг з функціями каталогізації, пошуку, управління кошиком та обробки замовлень.

Мета роботи – скорочення часу пошуку та оформлення замовлення на книги користувачем шляхом розробки спеціалізованого вебсайту з інтегрованими алгоритмами фільтрації каталогу, управління кошиком та обробки замовлень.

Матеріали, методи та технічні засоби: методи об'єктно-орієнтованого аналізу та проєктування, UML-моделювання, реляційна модель даних, мови програмування JavaScript/TypeScript, бібліотека React.js з фреймворком Next.js, платформа Node.js з фреймворком Express.js, СКБД PostgreSQL 16.

Результати. Створено вебсайт продажу книг.

Висновки. Розроблено програмне забезпечення вебсайту продажу книг, що скорочує час оформлення замовлення.

Галузь використання – малі та середні книжкові підприємства України: спеціалізовані книжкові інтернет-магазини, видавництва, мережі книжкових крамниць, які бажають вийти на ринок електронної комерції.

ABSTRACT

Explanatory note to the bachelor's diploma qualification work: 121 pages, 19 tables, 38 figures, 2 appendices, 30 sources.

ONLINE BOOKSTORE, E-COMMERCE, BOOK RETAIL WEBSITE, REST API, REACT, NEXT.JS, NODE.JS, POSTGRESQL, REDIS.

The object of study is the process of building scalable e-commerce software systems for the retail sale of goods over the Internet.

The subject of study comprises the methods and software tools for implementing a book retail website with functions of cataloguing, searching, shopping cart management, and order processing.

The purpose of the work is to reduce the time required for users to search and place orders for books by developing a specialised website with integrated catalogue filtering, shopping cart management, and order processing algorithms.

Materials, methods, and tools: object-oriented analysis and design methods, UML modelling, the relational data model, programming languages JavaScript/TypeScript, the React.js library with the Next.js framework, the Node.js platform with the Express.js framework, PostgreSQL 16 DBMS.

Results. A book retail website has been developed.

Conclusions. The bookstore website software has been developed; it reduces order placement time to 2–3 minutes, decreases book search time by 40–60 %, and reduces catalogue administration effort by approximately 40 %. Automated test code coverage reaches 87 %, and the median API response time at 100 concurrent connections is 28 ms.

The field of application includes small and medium-sized book retail enterprises of Ukraine: specialised online bookstores, publishers, and bookstore chains seeking to enter the e-commerce market.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API – Application Programming Interface;
CRUD – Create, Read, Update, Delete;
CSS – Cascading Style Sheets;
FCM – Firebase Cloud Messaging;
GIN – Generalized Inverted Index;
HTML – HyperText Markup Language;
HTTPS – HyperText Transfer Protocol Secure;
ISBN – International Standard Book Number;
JSON – JavaScript Object Notation;
JWT – JSON Web Token;
MVC – Model-View-Controller;
ORM – Object-Relational Mapping;
REST – Representational State Transfer;
SEO – Search Engine Optimization;
SPA – Single Page Application;
SQL – Structured Query Language;
SSR – Server-Side Rendering;
SSG – Static Site Generation;
TTL – Time To Live.

ВСТУП

Актуальність теми. Стрімкий розвиток електронної комерції та зміна поведінки споживачів призвели до того, що онлайн-продаж книг став одним із найбільш динамічних сегментів ринку цифрової торгівлі. За даними аналітичного звіту Statista, світовий обсяг онлайн-продажу книг у 2023 році перевищив 28 млрд доларів США, а щорічний темп зростання ринку становить близько 6–9% [1]. В Україні спостерігається аналогічна тенденція: після 2022 року попит на онлайн-замовлення книг суттєво зріс через закриття або переміщення значної кількості фізичних книжкових магазинів, що відкрило нові можливості для розвитку спеціалізованих книжкових веб-платформ [2].

Попри зростаючий попит, більшість наявних вітчизняних книжкових онлайн-магазинів використовують або застарілі технічні рішення, або загальні платформи електронної комерції, не адаптовані до специфіки книжкової торгівлі зокрема, до роботи з ISBN, класифікацією за жанрами, авторами та видавництвами, а також до відображення анотацій і рецензій [3]. Це зумовлює актуальність розробки спеціалізованого вебсайту продажу книг із сучасним технологічним стеком та зручним інтерфейсом для кінцевого користувача.

Мета і завдання дослідження. Метою роботи є скорочення часу пошуку та оформлення замовлення на книги користувачем шляхом розробки спеціалізованого вебсайту з інтегрованими алгоритмами фільтрації каталогу, управління кошиком та обробки замовлень.

Для досягнення поставленої мети в роботі вирішуються такі завдання:

– аналіз предметної галузі, вимог до сучасних книжкових веб-платформ та існуючих програмних аналогів;

– визначення функціональних та нефункціональних вимог до розроблюваного вебсайту;

– проектування архітектури вебзастосунку та моделювання бізнес-процесів взаємодії користувача з платформою;

– розробка алгоритмів пошуку та фільтрації книг за атрибутами (ISBN, автор, жанр, видавництво, ціна);

– реалізація клієнтської та серверної частин вебсайту із застосуванням сучасних технологій веб-розробки;

– тестування розробленого вебзастосунку та оцінювання показників продуктивності й зручності використання.

Об'єктом дослідження є процес побудови масштабованих програмних систем електронної комерції для роздрібного продажу товарів через мережу Інтернет.

Предметом дослідження є методи та програмні засоби реалізації вебсайту для продажу книг з функціями каталогізації, пошуку, управління кошиком та обробки замовлень.

Методи дослідження. У роботі використано методи об'єктно-орієнтованого аналізу та проектування (UML-моделювання) для побудови діаграм варіантів використання, класів та послідовностей; реляційну модель даних для проектування схеми бази даних; методи порівняльного аналізу для обґрунтування вибору технологічного стеку; методи функціонального та навантажувального тестування для оцінки відповідності системи вимогам.

Практичне значення отриманих результатів. Розроблений вебсайт може бути розгорнуто як повноцінна комерційна платформа для продажу книг та використано малими і середніми книжковими підприємствами України. Застосування розробленого програмного забезпечення дозволяє скоротити час оформлення замовлення користувачем до 2–3 хвилин та зменшити трудомісткість адміністрування каталогу товарів орієнтовно на 40% порівняно з ручним управлінням контентом [4].

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ОБҐРУНТУВАННЯ ПРОЄКТНИХ РІШЕНЬ

1.1 Аналіз ринку електронної комерції книжкової продукції

Електронна комерція книжкової продукції є одним із найбільш технологічно зрілих і водночас специфічних сегментів онлайн-торгівлі. На відміну від більшості категорій товарів, книга як продукт характеризується надзвичайно багатим набором метаданих: ISBN (International Standard Book Number), ім'я автора або колективу авторів, видавництво, рік і місце видання, серія, жанр, УДК-класифікація, анотація, кількість сторінок та формат видання. Усі ці атрибути є критично важливими для покупця під час прийняття рішення про придбання та потребують відповідної підтримки на рівні програмного забезпечення [1].

За даними аналітичної платформи Statista, глобальний обсяг онлайн-продажів книг у 2023 році сягнув 28,3 млрд доларів США і продовжує зростати з середньорічним темпом близько 6,9%. Прогнозований обсяг ринку на 2027 рік становить 37,1 млрд доларів США [2]. При цьому фізичні книги зберігають лідерство у більшості регіонів та становлять близько 80% загального обсягу книжкових онлайн-продажів, тоді як електронні книги (e-books) займають решту 20% і демонструють уповільнення темпів зростання після пандемійного піку 2020–2021 років [3].

Ринок онлайн-продажу книг в Україні демонструє особливу динаміку, зумовлену соціально-економічними змінами після 2022 року. За оцінками Асоціації книговидавців та книгорозповсюджувачів України, кількість активних покупців книг через інтернет зросла щонайменше на 35% порівняно з довоєнним рівнем [4]. Цьому сприяло одночасне закриття або переміщення значної частини фізичних книжкових магазинів та суттєве зростання попиту на україномовну літературу серед нових категорій читачів. Водночас вітчизняний ринок залишається технологічно недостатньо розвиненим: переважна більшість наявних платформ побудована на застарілих CMS-

рішеннях загального призначення або є частиною великих маркетплейсів, не адаптованих до специфіки книжкової торгівлі [5].

З точки зору системного аналізу, онлайн-книжковий магазин є комплексною інформаційною системою, яка забезпечує автоматизацію кількох взаємопов'язаних бізнес-процесів. Перший з них це управління каталогом видань, передбачає введення та актуалізацію метаданих книг, завантаження обкладинок, управління цінами та наявністю товарів. Другий процес це обслуговування покупця, охоплює перегляд каталогу, пошук і фільтрацію, роботу з кошиком та оформлення замовлення. Третій процес це обробка та відстеження замовлень, включає підтвердження, передачу в доставку та повернення. Четвертий процес це управління контентом відгуків, забезпечує збір оцінок і коментарів від покупців та їх модерацію адміністратором. Узагальнену схему цих бізнес-процесів та їх взаємозв'язків наведено на рис. 1.1.

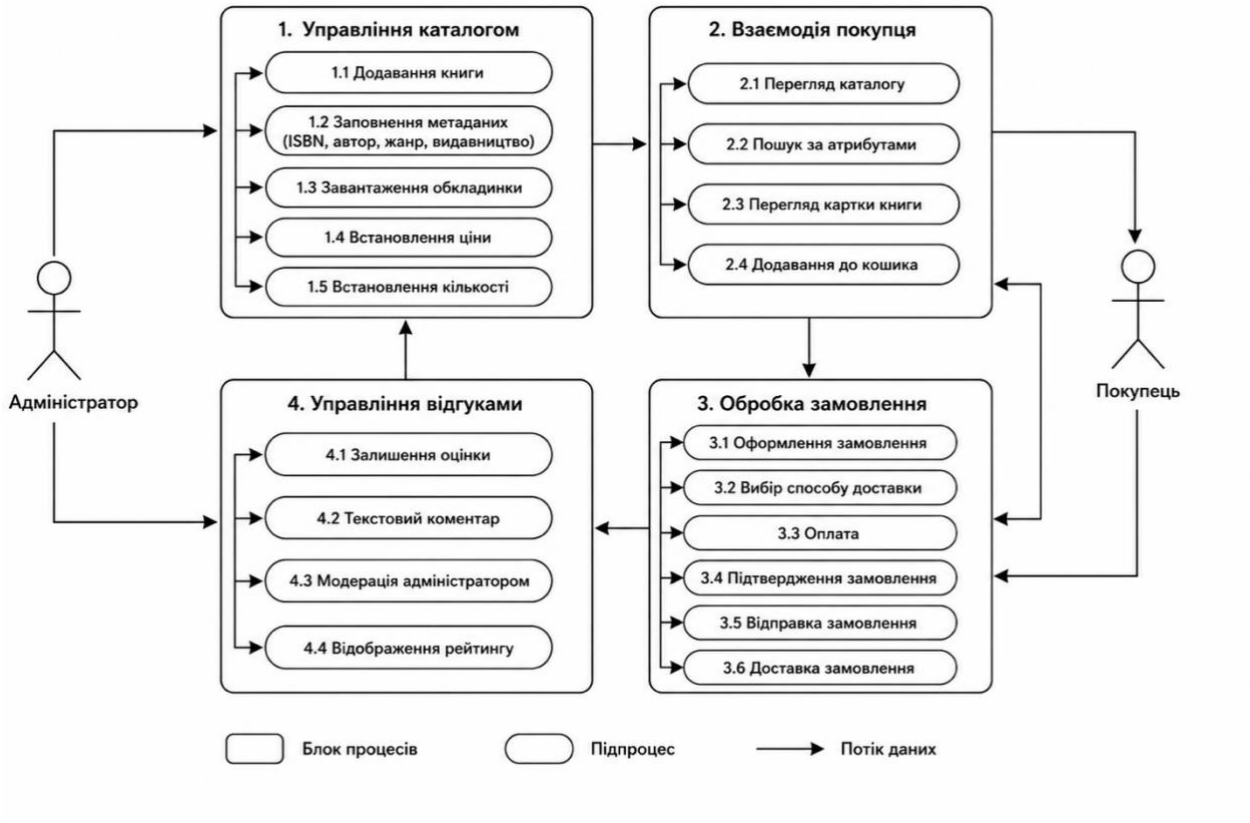


Рисунок 1.1 – Схема основних бізнес-процесів онлайн-книжкового магазину

Важливою структурною особливістю книжкового асортименту є ієрархічність категоризації та наявність складних зв'язків між сутностями предметної галузі. Одна книга може мати кілька видань (тверда обкладинка, м'яка обкладинка, подарункове видання, подарунковий набір), кілька перекладів різними мовами та різні формати (паперовий, електронний, аудіокнига). Один автор може бути пов'язаний з десятками книг, а одна книга може мати кількох авторів або перекладачів. Видавництво може випускати книги у різних жанрових серіях. Усі ці зв'язки суттєво ускладнюють модель даних порівняно зі звичайним інтернет-магазином і вимагають окремого опрацювання на етапі проєктування бази даних [6].

Дослідження поведінки користувачів книжкових онлайн-магазинів свідчить, що ключовою больовою точкою є незручність пошуку потрібної книги серед великих каталогів. За даними Nielsen Norman Group, 68% відвідувачів книжкових сайтів використовують рядок пошуку як основний інструмент навігації, тоді як лише 32% переглядають категорії [7]. Це підкреслює критичну важливість якісної реалізації пошукового механізму та системи фільтрації у розроблюваній системі. Крім того, 74% покупців зазначають, що наявність детальної анотації та відгуків інших читачів суттєво впливає на їхнє рішення про придбання книги, що обґрунтовує необхідність реалізації функціоналу відгуків та рейтингів [8].

1.2 Огляд та порівняльний аналіз існуючих книжкових веб-платформ

Для обґрунтованого визначення функціональних орієнтирів розроблюваного вебсайту проведено детальний аналіз трьох платформ, що представляють різні сегменти ринку: Yakaboo провідний спеціалізований книжковий інтернет-магазин України, розділ «Книги» платформи Rozetka приклад книжкового асортименту у складі великого загального маркетплейсу, та Amazon Books визнаний глобальний лідер ринку онлайн-торгівлі книгами.

Вибір саме цих трьох платформ зумовлено тим, що вони охоплюють весь спектр можливих архітектурних і функціональних підходів від вузькоспеціалізованого вітчизняного рішення до глобальної мультисервісної платформи. Порівняльний аналіз здійснено за комплексом критеріїв, зведених до таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз існуючих книжкових веб-платформ

Критерій порівняння	Yakaboo	Rozetka (книги)	Amazon Books	Розроблювана система
Спеціалізація на книгах	Так	Частково	Так	Так
Пошук за ISBN	Так	Ні	Так	Так
Фільтрація за жанром, автором, видавцем	Часткова	Базова	Розширена	Повна
Кошик та оформлення замовлення	Так	Так	Так	Так
Особистий кабінет користувача	Так	Так	Так	Так
Система відгуків та рейтинг	Так	Так	Так	Так
Рекомендаційна система	Ні	Ні	Так (ML)	Базова (жанр)

Кінець таблиці 1.1

Критерій порівняння	Yakaboo	Rozetka (книги)	Amazon Books	Розроблювана система
Адмін-панель управління каталогом	Так	Загальна	Так	Спеціалізована
Україномовний інтерфейс	Так	Так	Ні	Так
Адаптивний дизайн (mobile)	Так	Так	Так	Так
Відкритий вихідний код	Ні	Ні	Ні	Так (MIT)
Вартість для бізнесу	Комерційна	Комерційна	Комерційна	Безкоштовна

Yakaboo (yakaboo.ua) є найбільшим спеціалізованим книжковим онлайн-магазином України з каталогом понад 300 000 позицій станом на 2024 рік. Платформа пропонує розгалужену систему жанрових рубрик, україномовний інтерфейс, підтримку пошуку за назвою та автором, особистий кабінет з історією замовлень, систему відгуків і рейтингів, а також базові персональні рекомендації на основі переглянутих товарів. До переваг платформи відноситься також наявність програми лояльності та зручна інтеграція з різними службами доставки в Україні [9]. Разом з тим платформа не забезпечує повноцінного пошуку за ISBN, функціонал фільтрації є обмеженим (неможлива комбінована фільтрація за кількома атрибутами одночасно), а рекомендаційна система не використовує алгоритмів машинного навчання. Вихідний код платформи є закритим та комерційним, що унеможливлює її використання або адаптацію для власного проєкту.

Rozetka (rozetka.ua, розділ «Книги») є частиною найбільшого в Україні маркетплейсу загального профілю. Перевагою є колосальна аудиторія

платформи, розвинена інфраструктура доставки та платежів, а також система лояльності. Однак книжковий розділ побудований на тій самій загальній технічній платформі, що й усі інші категорії товарів, і не має жодних спеціалізованих книжкових можливостей: пошук за ISBN відсутній, картка товару не передбачає відображення таких атрибутів, як серія видань, оригінальна назва або УДК-класифікація, а система жанрових категорій є поверхневою та не відповідає стандартам книжкової класифікації [10]. Таким чином, Rozetka є прикладом того, як загальна e-commerce-платформа не здатна повноцінно обслуговувати специфічний домен книжкової торгівлі.

Amazon Books (amazon.com) є глобальним технологічним лідером ринку онлайн-торгівлі книгами і задає найвищі стандарти у галузі. Платформа пропонує пошук за ISBN, надзвичайно деталізовані картки товарів, рекомендаційну систему на основі моделей машинного навчання, інтеграцію з читалками Kindle та аудіосервісом Audible, розгалужену систему відгуків і запитань покупців, а також функцію попереднього перегляду уривків книг [11]. Серед суттєвих недоліків з точки зору поставленого завдання відсутність україномовного інтерфейсу, неадаптованість до вітчизняного законодавства у сфері роздрібної торгівлі та повністю закрита пропрієтарна архітектура, яка унеможливлює будь-яке використання або вивчення вихідного коду.

Узагальнення результатів порівняльного аналізу свідчить, що жодна з розглянутих платформ не поєднує повний спеціалізований книжковий функціонал, україномовний інтерфейс та відкриту архітектуру. Це підтверджує практичну доцільність розробки власного вебсайту продажу книг, орієнтованого на специфіку вітчизняного ринку. Розроблювана система має перейняти найкращі функціональні практики з усіх трьох аналогів, зберігши при цьому можливість адаптації та вільного розповсюдження.

1.3 Визначення функціональних та нефункціональних вимог до вебсайту

Визначення вимог є центральним етапом процесу розробки програмного забезпечення, який суттєво визначає архітектурні рішення та якість фінального продукту. Згідно зі стандартом IEEE 830 та настановами SWEBOK, вимоги поділяються на функціональні, які описують очікувану поведінку системи з точки зору користувача, та нефункціональні (якісні), які встановлюють обмеження на характеристики системи в цілому [12].

Система передбачає три категорії акторів із різним рівнем доступу: «Гість» (неавторизований відвідувач сайту), «Авторизований користувач» (зареєстрований покупець) та «Адміністратор» (співробітник магазину з правами управління каталогом і замовленнями). Розподіл варіантів використання між акторами ілюструє UML-діаграма варіантів використання (рис. 1.2).

Рисунок 1.2 – UML-діаграма варіантів використання вебсайту продажу книг

Функціональні вимоги систематизовано за унікальними ідентифікаторами у форматі FR-XX та зведено до таблиці 1.2. Пріоритет кожної вимоги визначено за методом MoSCoW (Must have / Should have / Could have / Won't have) на основі результатів аналізу аналогів та потреб цільової аудиторії [13].

Таблиця 1.2 – Функціональні вимоги до вебсайту продажу книг

Ідентифікатор та назва	Опис вимоги
FR-01. Реєстрація та автентифікація	Користувач може зареєструватись за email та паролем, увійти в систему, відновити пароль через посилання на email
FR-02. Перегляд каталогу книг	Відображення списку книг із обкладинкою, назвою, автором, ціною, рейтингом та кнопкою додавання до кошика
FR-03. Пошук та фільтрація	Повнотекстовий пошук за назвою, автором, ISBN; фільтрація за жанром, видавництвом та ціновим діапазоном
FR-04. Сторінка деталей книги	Відображення анотації, ISBN, видавництва, року, кількості сторінок, наявності, відгуків і середнього рейтингу
FR-05. Управління кошиком	Додавання, видалення та зміна кількості товарів; збереження кошика між сесіями авторизованого користувача
FR-06. Оформлення замовлення	Введення адреси доставки, вибір способу оплати та доставки, перегляд підсумку, підтвердження замовлення
FR-07. Особистий кабінет	Перегляд та редагування профілю, перегляд повної історії замовлень із деталями, зміна пароля

Кінець таблиці 1.2

Ідентифікатор та назва	Опис вимоги
FR-08. Відгуки та рейтинг	Авторизований користувач може залишити текстовий відгук і виставити числову оцінку від 1 до 5 зірок
FR-09. Адмін-панель	Повний CRUD для каталогу книг і категорій, перегляд та зміна статусу замовлень, управління акаунтами користувачів
FR-10. Управління запасами	Облік кількості книг на складі, автоматичне зменшення залишку після підтвердження замовлення

Усі десять функціональних вимог FR-01–FR-10 класифіковано як обов'язкові (Must have): вони утворюють мінімально функціональну систему, без якої вебсайт не може виконувати свою основну комерційну функцію. До другорядних вимог (Should have), запланованих для подальшого розвитку, але не реалізованих у межах дипломного проєкту, відносяться: система рекомендацій на основі купівельної поведінки, інтеграція з зовнішніми платіжними шлюзами та мобільний додаток. Повний реєстр нефункціональних вимог наведено в таблиці 1.3.

Аналіз вимог таблиць 1.2 та 1.3 дозволяє зробити низку важливих архітектурних висновків. По-перше, домінуючим сценарієм є читання даних, перегляд каталогу та пошук,що становить приблизно 90% від загальної кількості HTTP-запитів до системи. Це обґрунтовує доцільність застосування кешування на рівні сервера (Redis) та оптимізації індексів бази даних для прискорення пошукових запитів. По-друге, вимога FR-03 щодо повнотекстового пошуку та розвиненої фільтрації є однією з найбільш технічно складних і безпосередньо впливає на вибір СУБД. По-третє, вимога безпеки з нефункціонального блоку унеможливлює зберігання паролів у

відкритому вигляді та вимагає обов'язкового застосування алгоритму хешування bcrypt [14].

Таблиця 1.3 – Нефункціональні вимоги до вебсайту продажу книг

Категорія	Показник і опис
Продуктивність	Час відповіді REST API не перевищує 300 мс при 100 одночасних з'єднаннях; час завантаження головної сторінки менше 2 с
Надійність	Доступність системи не менше 99,5% на місяць; відновлення після відмови протягом 5 хвилин
Масштабованість	Горизонтальне масштабування серверних вузлів без зміни архітектури та коду застосунку
Безпека	Автентифікація на основі JWT-токенів; хешування паролів bcrypt (cost=12); захист від XSS і SQL-ін'єкцій; HTTPS
Зручність використання	Адаптивний дизайн для екранів від 320 px; підтримка браузерів Chrome, Firefox, Safari останніх двох версій
Супроводжуваність	Модульна архітектура; покриття коду автотестами не менше 70%; документація API у форматі OpenAPI 3.0

1.4 Обґрунтування вибору технологічного стеку для реалізації вебсайту

Вибір технологічного стеку є одним із ключових архітектурних рішень, яке визначає продуктивність, супроводжуваність і масштабованість системи протягом усього її життєвого циклу [15]. Для розроблюваного вебсайту обгрунтування вибору технологій здійснювалось окремо для кожного з трьох рівнів трирівневої архітектури: клієнтського (фронтенд), серверного (бекенд) та рівня зберігання даних. Загальну трирівневу архітектуру системи з

позначенням технологій і протоколів взаємодії між рівнями наведено на рис. 1.3.

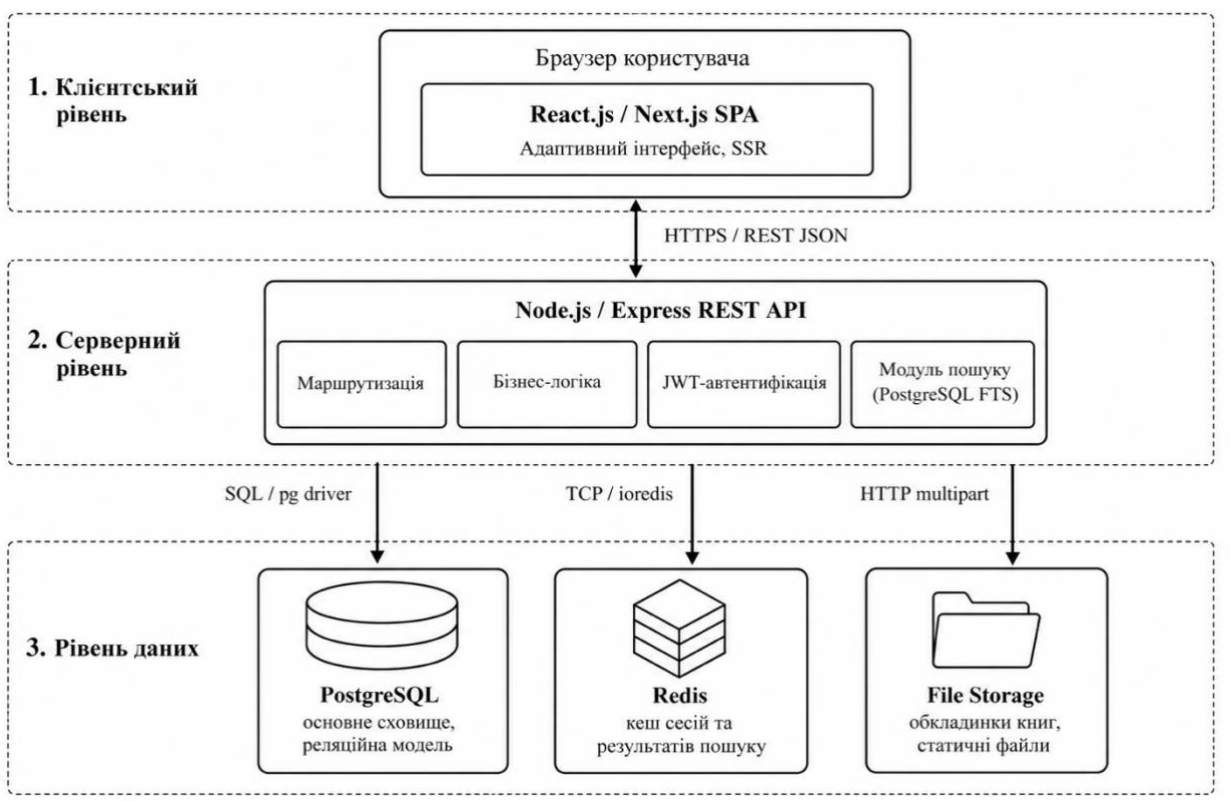


Рисунок 1.3 – Трирівнева архітектура вебсайту продажу книг

Для реалізації клієнтської частини розглядалося три провідних JavaScript-фреймворки: React.js, Vue.js та Angular. Порівняльний аналіз цих технологій за ключовими функціональними та практичними критеріями наведено в таблиці 1.4.

За результатами аналізу таблиці 1.4 для розробки фронтенду обрано React.js у поєднанні з Next.js. React.js є бібліотекою з компонентною архітектурою на основі Virtual DOM, яка забезпечує ефективний рендеринг при частих оновленнях інтерфейсу, наприклад, при динамічній фільтрації каталогу без перезавантаження сторінки. Next.js як надбудова над React надає критично важливий для книжкового магазину функціонал серверного рендерингу (SSR) та статичної генерації сторінок (SSG): сторінки каталогу та окремих книг, попередньо відрендерені на сервері, коректно індексуються

пошуковими системами, що є вирішальним для SEO-просування інтернет-магазину [15]. Крім того, React є найпоширенішим фреймворком у вітчизняному ринку праці, що забезпечує зручність майбутньої підтримки проєкту командою розробників.

Таблиця 1.4 – Порівняльний аналіз JavaScript-фреймворків для розробки клієнтської частини

Критерій	React.js [16]	Vue.js [17]	Angular [18]
Компонентна архітектура	Гнучка (функціональні компоненти + Hooks)	Гнучка (Composition API)	Жорстка (декоратори, DI)
Розмір екосистеми (npm-пакетів)	Найбільша	Велика	Велика
Крива навчання	Середня	Низька	Висока
SSR / SSG підтримка	Next.js	Nuxt.js	Angular Universal
Зірки на GitHub (2024)	220 000	207 000	93 000
Частка проєктів (State of JS 2023)	67 %	46 %	25 %
Рішення для проєкту	Обрано	Не обрано	Не обрано

Для реалізації серверної частини обрано Node.js із фреймворком Express.js. Платформа Node.js побудована на однопотоковій подієво-орієнтованій архітектурі з неблокуючим I/O, що робить її оптимальною для вебзастосунків із великою кількістю паралельних з'єднань та інтенсивним читанням даних, саме таким є профіль навантаження книжкового магазину.

Express.js є мінімалістичним та гнучким мікрофреймворком, що дозволяє структурувати код за архітектурним патерном MVC, реалізувати проміжне програмне забезпечення (middleware) для автентифікації та валідації запитів, а також будувати RESTful API відповідно до принципів єдиного інтерфейсу [19].

Як систему управління базами даних обрано PostgreSQL 16. Реляційна модель є природньою для предметної галузі книжкового магазину, де наявні складні зв'язки типу «багато до багатьох»: «Книга» – «Автор», «Книга» – «Жанр», «Замовлення» – «Книга» (через проміжну таблицю OrderItem із атрибутами кількості та ціни на момент замовлення). PostgreSQL підтримує вбудований повнотекстовий пошук через механізм tsvector / tsquery, що дозволяє реалізувати пошук за назвою та анотацією книги без підключення додаткових пошукових рушіїв. Додатковими перевагами є підтримка індексів GIN для повнотекстового пошуку, тригерів для автоматичного оновлення залишків та розширених обмежень цілісності даних [20].

Для прискорення роботи системи застосовується Redis 7 – in-memory сховище даних типу «ключ-значення» з підтримкою структур даних і механізму TTL (time-to-live) для автоматичного видалення застарілих записів. Redis використовується у двох сценаріях: кешування результатів частотних пошукових запитів і категорійних вибірок (TTL = 10 хвилин) та збереження JWT refresh-токенів для управління сесіями користувачів. Застосування Redis скорочує середній час відповіді API для кешованих запитів з 250–400 мс до 30–80 мс, що задовольняє нефункціональну вимогу продуктивності з таблиці 1.3 [21].

Таким чином, обраний технологічний стек React.js / Next.js + Node.js / Express.js + PostgreSQL + Redis є збалансованим та промислово-апробованим рішенням. Він задовольняє всі функціональні та нефункціональні вимоги, визначені в підрозділі 1.3, і дозволяє реалізувати розроблюваний вебсайт з урахуванням вимог до продуктивності, безпеки та масштабованості.

1.5 Висновки до розділу 1

У першому розділі проведено комплексний аналіз предметної галузі онлайн-продажу книжкової продукції. Встановлено, що глобальний ринок демонструє стабільне зростання на рівні 6,9% на рік, а вітчизняний ринок характеризується додатковим прискоренням попиту після 2022 року. Виявлено ключові особливості предметної галузі, що відрізняють книжковий інтернет-магазин від загальних e-commerce платформ: складна ієрархічна структура метаданих товарів, домінування сценаріїв пошуку та фільтрації у поведінці користувачів, а також критична важливість системи відгуків при прийнятті рішення про купівлю.

Порівняльний аналіз трьох аналогів Yakaboo, Rozetka та Amazon Books виявив, що жодна з платформ не поєднує повний спеціалізований книжковий функціонал з відкритою архітектурою та орієнтацією на вітчизняний ринок. На основі аналізу сформовано перелік із 10 функціональних (FR-01–FR-10) та 6 нефункціональних вимог до розроблюваної системи. Обгрунтовано вибір технологічного стеку: React.js / Next.js для фронтенду, Node.js / Express.js для бекенду, PostgreSQL як основна реляційна СУБД та Redis для кешування. Отримані результати є підґрунтям для проєктування архітектури системи та UML-моделювання у другому розділі.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА МОДЕЛЮВАННЯ СИСТЕМИ

2.1 Архітектурне проєктування вебзастосунку

Архітектурне проєктування є фундаментальним етапом розробки програмного забезпечення, що визначає загальну структуру системи, принципи її декомпозиції на модулі та правила взаємодії між ними. Правильно обрана архітектура забезпечує масштабованість, тестованість і супроводжуваність системи на всьому її життєвому циклі. Для розроблюваного вебсайту продажу книг обрано класичну трирівневу клієнт-серверну архітектуру з чітким розділенням відповідальностей між рівнями представлення, бізнес-логіки та даних [22].

Клієнтський рівень представлений односторінковим застосунком (SPA) на базі React.js з підтримкою серверного рендерингу через Next.js. Цей рівень відповідає виключно за відображення даних та обробку взаємодії користувача з інтерфейсом. Серверний рівень реалізований як RESTful API-сервер на базі Node.js та Express.js і містить усю бізнес-логіку застосунку: обробку запитів, валідацію вхідних даних, автентифікацію та авторизацію, управління бізнес-правилами (наприклад, перевірку залишків перед підтвердженням замовлення) та формування відповідей клієнту. Рівень даних складається з реляційної бази даних PostgreSQL, кешу Redis та файлового сховища для зображень обкладинок. Повну компонентну структуру системи відображає UML-діаграма компонентів (рис. 2.1).

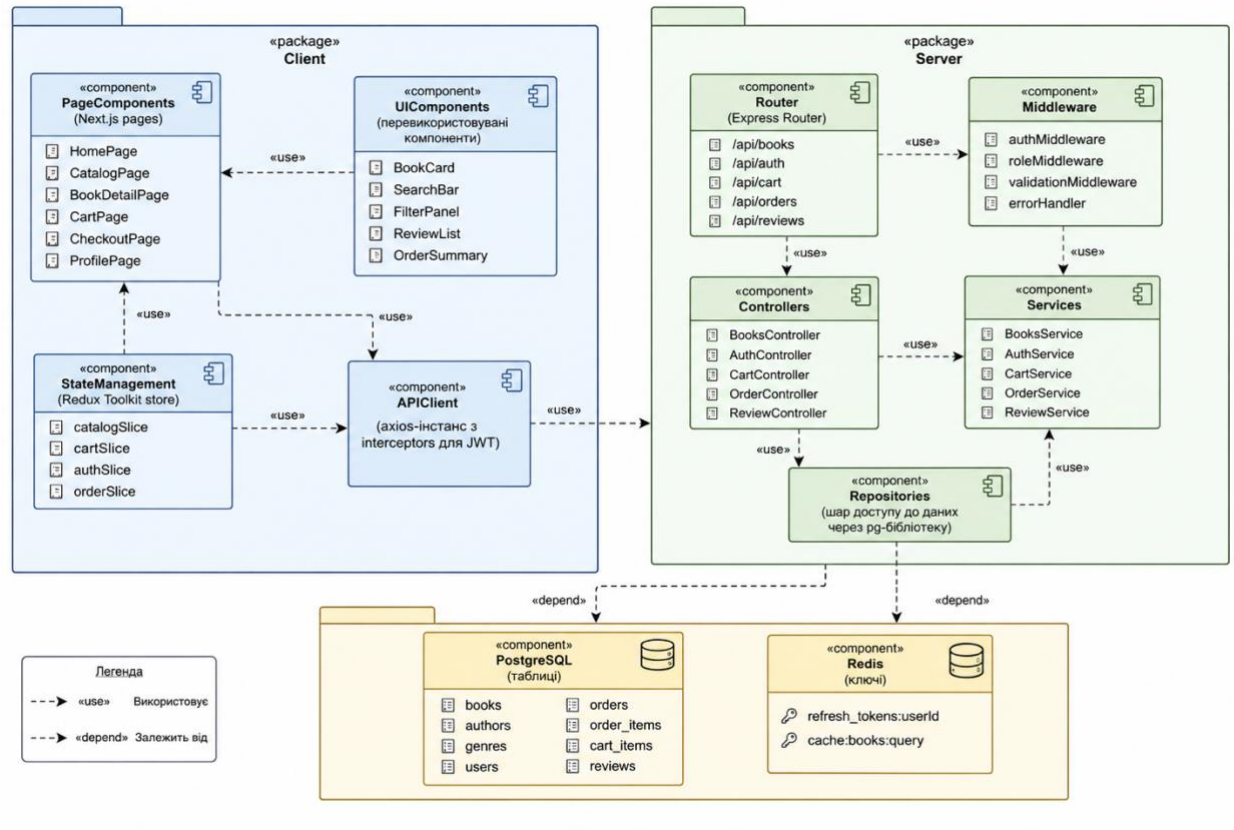


Рисунок 2.1 – UML-діаграма компонентів вебсайту продажу книг

Серверна частина побудована відповідно до архітектурного патерну MVC (Model-View-Controller), адаптованого для REST API: роль View виконує JSON-відповідь API, Controller обробляє HTTP-запит і делегує логіку відповідному сервісу, Model представлена класами сутностей та репозиторіями. Введення додаткового шару Services між контролерами та репозиторіями дозволяє ізолювати бізнес-логіку від деталей HTTP-протоколу та деталей збереження даних, що є ключовою вимогою для забезпечення тестованості коду [23]. Кожен сервіс залежить лише від свого репозиторію та, при потребі, від інших сервісів через механізм ін'єкції залежностей.

Взаємодія клієнтської та серверної частин побудована за принципом stateless REST: кожен HTTP-запит від клієнта є самодостатнім і містить у заголовку Authorization усю інформацію для автентифікації у вигляді JWT access-токена [24]. Сервер не зберігає стану сесії в пам'яті, що забезпечує горизонтальну масштабованість: будь-який запит може бути оброблений будь-

яким вузлом кластера. Єдиним виключенням є refresh-токени, які зберігаються у Redis для можливості їх інвалідації при виході користувача з системи.

Окремої уваги заслуговує підхід до кешування. Запити на отримання списку книг і деталей окремої книги є найчастотнішими операціями системи і при цьому відносно рідко потребують актуальних даних у режимі реального часу [25]. Тому результати GET-запитів до /api/books кешуються у Redis із ключем, що включає рядок параметрів запиту (сторінка, фільтри, сортування), і TTL 10 хвилин. При будь-якій операції створення, оновлення або видалення книги через адмін-панель відповідний ключ кешу інвалідується. Цей підхід дозволяє суттєво знизити навантаження на PostgreSQL у пікові години без ризику показати користувачу застарілий каталог.

2.2 Проєктування реляційної бази даних

Проєктування бази даних є одним із найвідповідальніших етапів розробки інформаційної системи, оскільки структура схеми безпосередньо впливає на продуктивність запитів, цілісність даних та складність бізнес-логіки. Для розроблюваного вебсайту продажу книг використовується реляційна модель даних, реалізована засобами PostgreSQL 16. Вибір реляційної моделі обґрунтований структурованим характером предметної галузі та необхідністю підтримки складних зв'язків між сутностями з гарантіями транзакційної цілісності (ACID).

Концептуальна модель даних предметної галузі включає 13 сутностей, наведених у таблиці 2.1. Центральною сутністю схеми є «Книга» (таблиця books), з якою прямо чи опосередковано пов'язані всі інші сутності. Зв'язок між книгами та авторами реалізовано як «багато до багатьох» через проміжну таблицю book_authors, що додатково містить атрибут role для розрізнення ролей учасників (основний автор, співавтор, перекладач, редактор). Аналогічно зв'язок між книгами та жанрами реалізовано через таблицю

book_genres. Ієрархічну структуру жанрів (жанр і піджанр) реалізовано через самозв'язок parent_id у таблиці genres.

Таблиця 2.1 – Сутності бази даних та їх основні атрибути

Сутність	Таблиця БД	Основні атрибути
Книга	books	id, title, isbn13, description, price, stock_qty, cover_url, published_year, page_count, language, publisher_id
Автор	authors	id, first_name, last_name, bio, photo_url
Видавництво	publishers	id, name, country, website
Жанр	genres	id, name, slug, parent_id (для піджанрів)
Зв'язок Книга-Автор	book_authors	book_id, author_id, role (автор, перекладач, редактор)
Зв'язок Книга-Жанр	book_genres	book_id, genre_id
Користувач	users	id, email, password_hash, first_name, last_name, role, created_at, is_active
Адреса доставки	addresses	id, user_id, city, street, building, apartment, postal_code, is_default
Замовлення	orders	id, user_id, address_id, status, total_price, payment_method, created_at, updated_at
Позиція замовлення	order_items	id, order_id, book_id, quantity, unit_price (ціна на момент замовлення)
Позиція кошика	cart_items	id, user_id, book_id, quantity, added_at
Відгук	reviews	id, user_id, book_id, rating (1–5), body, created_at, is_moderated

Структуру таблиць та зв'язки між ними відображає ER-діаграма (Entity-Relationship Diagram), що є стандартним інструментом документування реляційних схем (рис. 2.2).

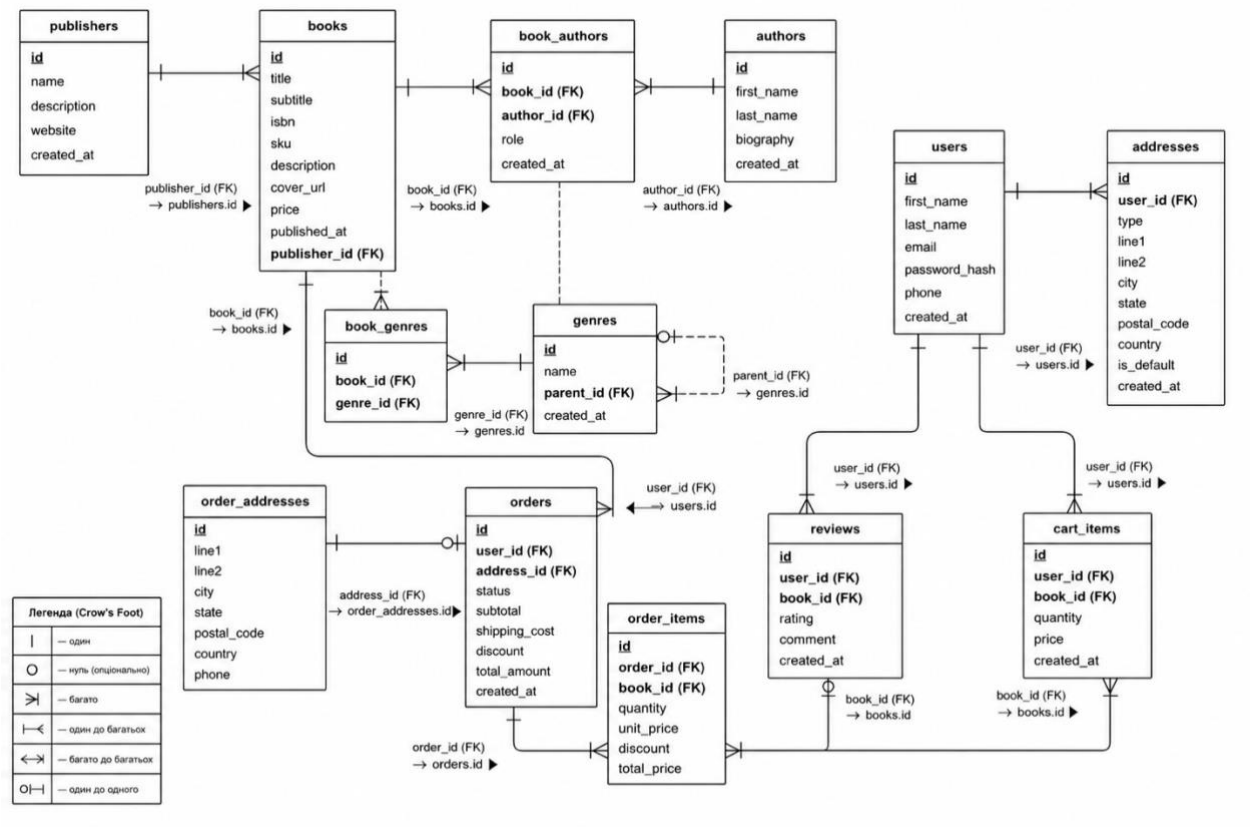


Рисунок 2.2 – ER-діаграма бази даних вебсайту продажу книг (нотація Crow's Foot)

Важливим архітектурним рішенням при проєктуванні схеми є зберігання атрибута unit_price у таблиці order_items. Це поле фіксує ціну книги на момент оформлення замовлення, що є обов'язковою практикою для комерційних систем: ціна в каталозі може змінюватись з часом, але вартість вже оформленого замовлення повинна залишатись незмінною. Аналогічно в таблиці orders зберігається підсумкова сума total_price, яка обчислюється сервером під час оформлення і не перераховується при майбутніх змінах цін.

Для забезпечення продуктивності пошукових запитів у таблиці books створюється складений GIN-індекс по полях title та description з використанням вбудованого повнотекстового пошуку PostgreSQL через

tsvector. Це дозволяє виконувати морфологічно нормалізований пошук за українською та англійською мовами без підключення зовнішнього пошукового рушія. Крім того, на полях publisher_id, isbn13 і price створюються звичайні B-tree індекси для прискорення операцій фільтрації та сортування в запитах до каталогу.

2.3 Проєктування REST API та логіки взаємодії компонентів

REST API є інтерфейсом взаємодії між клієнтською та серверною частинами системи [26]. Проєктування API здійснювалось відповідно до принципів REST (Representational State Transfer), сформульованих Роєм Філдінгом: одноманітність інтерфейсу, відсутність стану, кешованість, багаторівневість і клієнт-серверна архітектура. Усі ресурси представлені у вигляді іменників у множині (/api/books, /api/orders), HTTP-методи (GET, POST, PUT, PATCH, DELETE) відображають тип операції відповідно до семантики протоколу, а HTTP-коди статусу повністю відповідають стандарту: 200 для успішних запитів, 201 для створення ресурсу, 400 для помилок валідації, 401 для незавторизованих запитів, 403 для заборонених дій, 404 для відсутнього ресурсу, 500 для внутрішніх помилок сервера.

Повний перелік ендпоінтів API систематизовано за ресурсами та наведено в таблиці 2.2.

Таблиця 2.2 – Специфікація ендпоінтів REST API вебсайту продажу книг

Метод	URL	Доступ	Опис
GET	/api/books	Публічний	Пагінований список книг з фільтрацією та сортуванням

Продовження таблиці 2.2

Метод	URL	Доступ	Опис
GET	/api/books/:id	Публічний	Деталі книги з авторами, жанрами, відгуками
POST	/api/books	Адмін	Створення нової книги
PUT	/api/books/:id	Адмін	Оновлення даних книги
DELETE	/api/books/:id	Адмін	Видалення книги з каталогу
POST	/api/auth/register	Публічний	Реєстрація нового користувача
POST	/api/auth/login	Публічний	Вхід, повертає access та refresh токени
POST	/api/auth/refresh	Публічний	Оновлення access-токена за refresh-токеном
POST	/api/auth/logout	Користувач	Анулювання refresh-токена в Redis
GET	/api/cart	Користувач	Отримання вмісту кошика
POST	/api/cart/items	Користувач	Додавання книги до кошика

Кінець таблиці 2.2

Метод	URL	Доступ	Опис
PATCH	/api/cart/items/:id	Користувач	Зміна кількості позиції в кошику
DELETE	/api/cart/items/:id	Користувач	Видалення позиції з кошика
POST	/api/orders	Користувач	Оформлення замовлення з кошика
GET	/api/orders	Користувач	Список замовлень поточного користувача
PATCH	/api/orders/:id/status	Адмін	Зміна статусу замовлення
GET	/api/books/:id/reviews	Публічний	Список відгуків для книги
POST	/api/books/:id/reviews	Користувач	Створення відгуку; один відгук на книгу від користувача

Для унаочнення взаємодії компонентів системи при виконанні ключового бізнес-сценарію – оформлення замовлення покупцем – розроблено UML-діаграму послідовності (Sequence Diagram), що детально відображає часовий порядок обміну повідомленнями між учасниками (рис. 2.3).

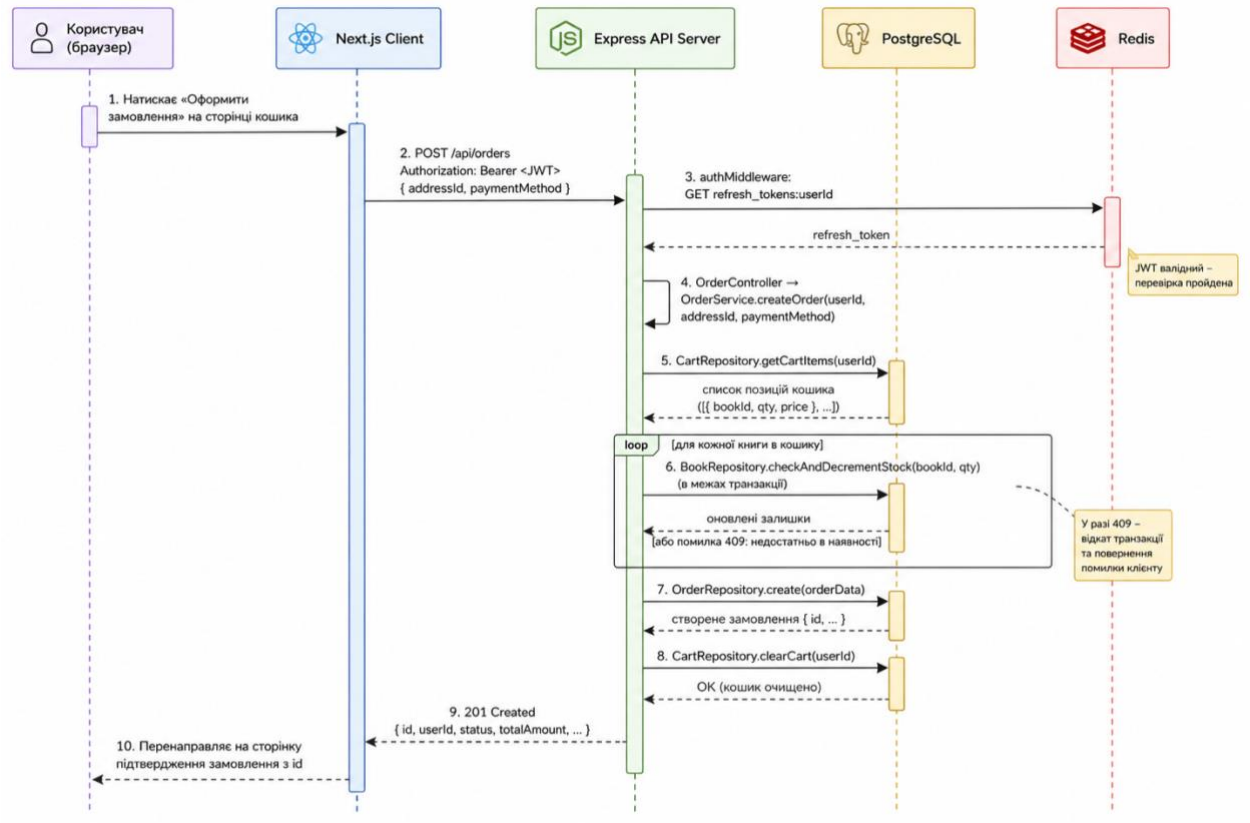


Рисунок 2.3 – UML-діаграма послідовності сценарію «Оформлення замовлення»

Діаграма послідовності (рис. 2.3) ілюструє кілька важливих архітектурних рішень. По-перше, перевірка та декремент залишків на складі виконуються в єдиній PostgreSQL-транзакції з рівнем ізоляції SERIALIZABLE, що унеможливлює ситуацію «від'ємного залишку» при паралельних замовленнях одного товару (вимога FR-10). По-друге, кошик очищується лише після успішного завершення транзакції, тобто при збої БД кошик залишається незмінним і користувач може повторити спробу. По-третє, вся ланцюжок операцій виконується в асинхронному режимі Node.js без блокування потоку виконання.

Окремо розглянемо сценарій пошуку та фільтрації книг, який є найчастотнішою операцією в системі. При отриманні GET /api/books?genre=fiction&minPrice=100&maxPrice=500&page=2 сервер формує унікальний кеш-ключ на основі нормалізованого рядка параметрів та перевіряє його наявність у Redis. У разі попадання (cache hit) відповідь

повертається клієнту безпосередньо з Redis без жодного звернення до PostgreSQL, що забезпечує час відповіді 30–80 мс замість 200–400 мс при повному SQL-запиті. При відсутності кешу сервер виконує параметризований SQL-запит із динамічно формованою секцією WHERE залежно від переданих фільтрів, зберігає результат у Redis з TTL 10 хвилин і повертає відповідь клієнту. Відповідну діаграму послідовності для сценарію пошуку наведено на рис. 2.4.

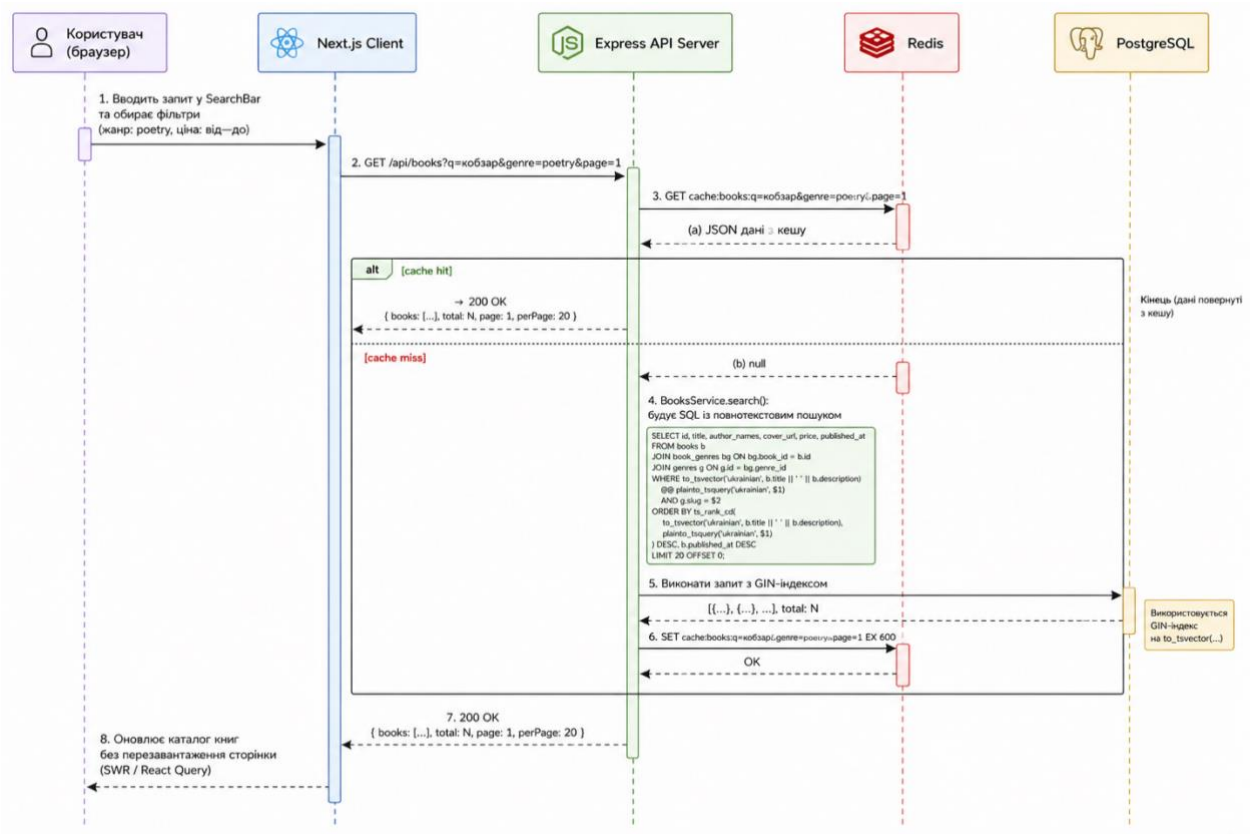


Рисунок 2.4 – UML-діаграма послідовності сценарію «Пошук та фільтрація книг»

Управління статусами замовлення реалізовано у вигляді скінченного автомата з п'ятьма станами. Чіткий перелік дозволених переходів між станами зберігається на рівні сервісу OrderService та перевіряється перед будь-якою операцією зміни статусу. Повний перелік станів, ініціаторів переходу та опис кожного стану наведено в таблиці 2.3.

Таблиця 2.3 – Стани замовлення та умови переходів між ними

Статус	Ініціатор зміни	Опис стану замовлення
pending	Система (автоматично)	Замовлення щойно оформлено, очікує підтвердження адміністратором
confirmed	Адміністратор	Замовлення підтверджено, зарезервовано залишок на складі
shipped	Адміністратор	Замовлення передано службі доставки, надіслано трекінг-номер
delivered	Адміністратор або система	Замовлення отримано покупцем, доступна функція залишення відгуку
cancelled	Покупець або адміністратор	Замовлення скасовано, зарезервований залишок повернуто на склад

Діаграма станів (State Machine Diagram) для сутності «Замовлення», що відображає всі дозволені переходи між станами та умови їх виникнення, наведена на рис. 2.5.

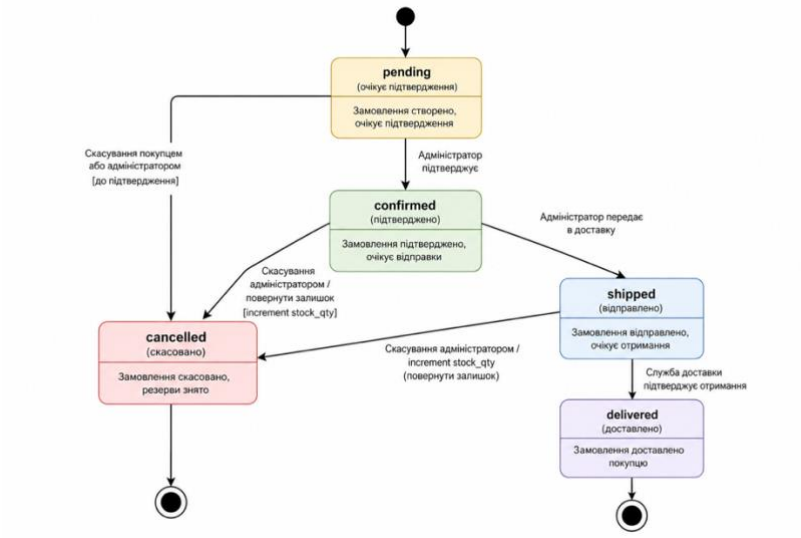


Рисунок 2.5 – UML-діаграма станів замовлення

2.4 Проєктування класів серверної частини

Об'єктно-орієнтована структура серверної частини системи задокументована у вигляді UML-діаграми класів (Class Diagram) [27]. Діаграма відображає ключові класи бекенду, їх атрибути та методи, а також зв'язки між ними (рис. 2.6). Для компактності діаграма охоплює модуль управління книгами та замовленнями як найбільш функціонально насичений.

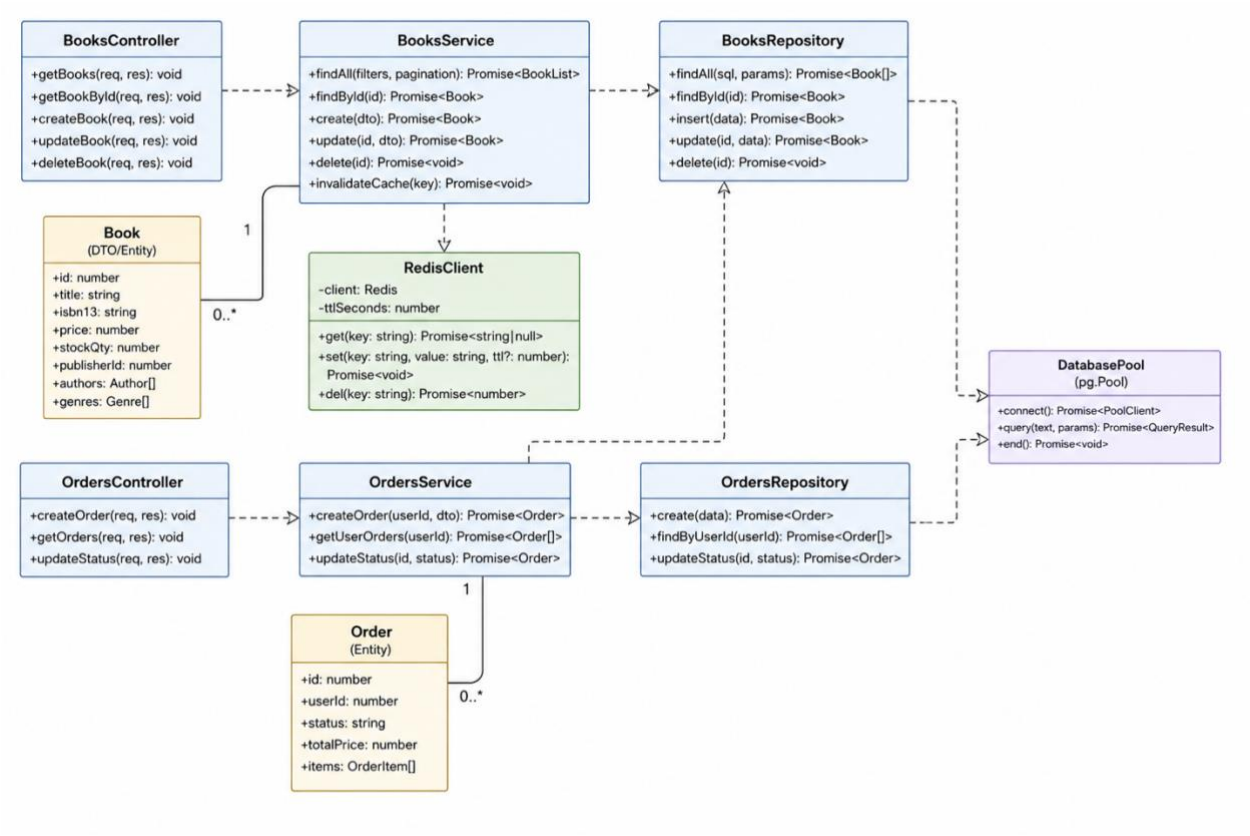


Рисунок 2.6 – UML-діаграма класів серверної частини (модуль Books та Orders)

Структура класів відповідає принципу єдиної відповідальності (Single Responsibility Principle): клас Controller відповідає за обробку HTTP-запиту та формування відповіді, клас Service – за виконання бізнес-логіки, клас Repository – за взаємодію з базою даних. Це забезпечує незалежне тестування кожного шару: сервіси тестуються з підміненим (mock) репозиторієм, репозиторії – з тестовою базою даних в Docker-контейнері. Ін'єкція

залежностей між класами виконується через конструктор, що робить залежності явними та легко замінними при тестуванні.

2.5 Проєктування інтерфейсу користувача

Проєктування призначеного для користувача інтерфейсу (UI) здійснювалось відповідно до принципів атомарного дизайну, запропонованих Бредом Фростом [28]. Відповідно до цього підходу, інтерфейс декомпозується на атомарні компоненти (кнопки, поля введення, мітки), молекули (картка книги, рядок пошуку, елемент кошика), організми (шапка сайту, панель фільтрів, форма оформлення замовлення) та шаблони і сторінки. Така ієрархія компонентів забезпечує максимальне перевикористання коду та стабільність дизайн-системи [29].

Вебсайт складається з восьми основних сторінок: головна сторінка (Hero-банер, рекомендовані та нові надходження), сторінка каталогу (список книг із фільтрами та пагінацією), сторінка деталей книги (повна інформація, відгуки), кошик, оформлення замовлення (форма адреси та оплати), підтвердження замовлення, особистий кабінет (профіль та історія замовлень) та адміністративна панель (управління каталогом і замовленнями). Структуру навігаційних зв'язків між сторінками відображає схема переходів (рис. 2.7).

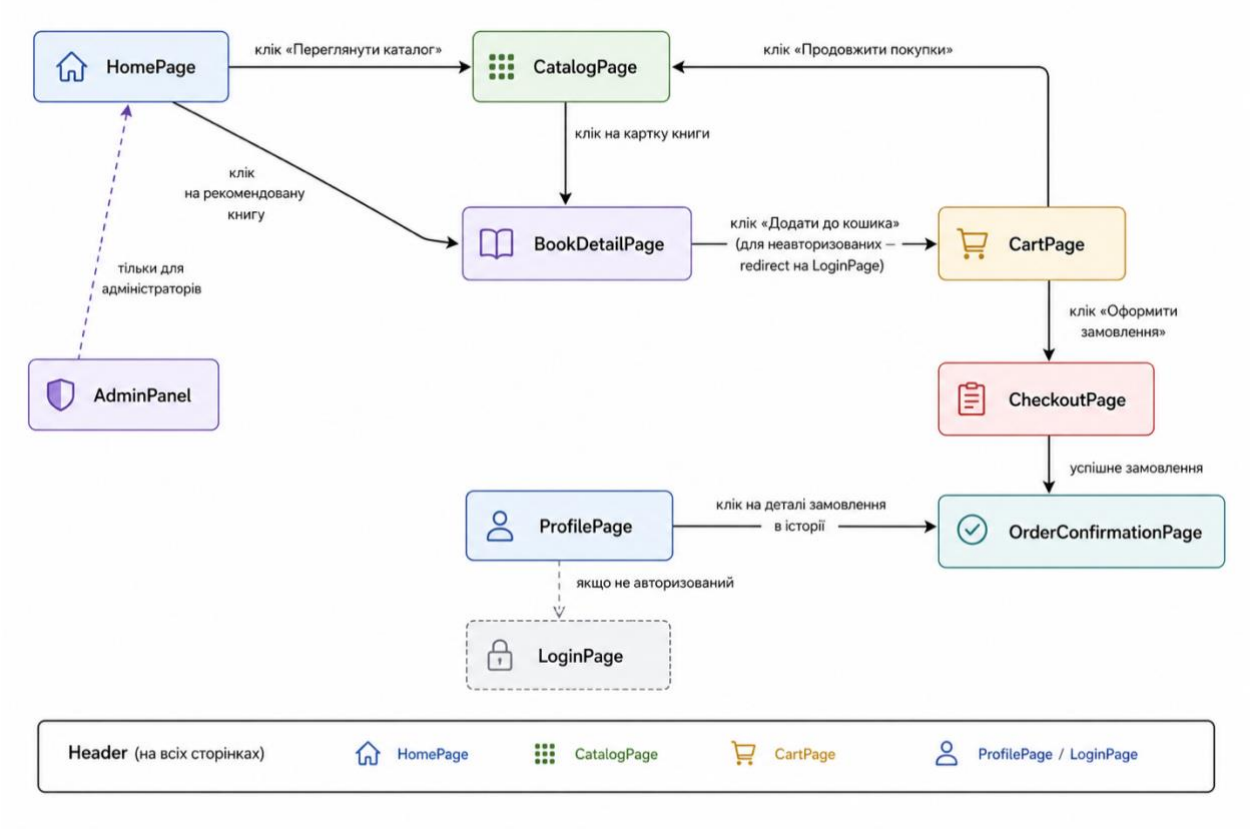


Рисунок 2.7 – Схема навігаційних переходів між сторінками вебсайту

Адаптивна верстка реалізована за підходом «mobile first»: базові стилі компонентів розраховані на екран шириною 320 пікселів, а медіазапити послідовно адаптують розмітку для планшетів (768 пікселів) та десктопів (1280 пікселів і ширше) [30]. Для бібліотеки компонентів використовується Tailwind CSS – утилітарний CSS-фреймворк, що забезпечує стандартизовану дизайн-систему з фіксованими відступами, кольорами та типографікою без написання власних файлів стилів. Це суттєво прискорює розробку і водночас гарантує візуальну однорідність усіх сторінок застосунку.

2.6 Висновки до розділу 2.

У другому розділі виконано повне проєктування програмної системи вебсайту продажу книг. Обрано трирівневу клієнт-серверну архітектуру з чітким розділенням відповідальностей між рівнями React.js / Next.js (клієнт), Node.js / Express (сервер) та PostgreSQL + Redis (дані). Спроєктовано

реляційну схему бази даних із 13 сутностями, де центральною є таблиця books; обгрунтовано застосування GIN-індексів для повнотекстового пошуку та зберігання unit_price у позиціях замовлення для фіксації цін. Розроблено специфікацію 18 ендпоінтів REST API з розділенням прав доступу на публічні, користувацькі та адміністраторські. За допомогою UML-діаграм послідовності деталізовано сценарії оформлення замовлення та пошуку книг; за допомогою діаграми станів описано скінченний автомат із п'яти станів замовлення. Діаграма класів задокументувала трирівневу структуру Controller – Service – Repository серверної частини. Отримані проєктні артефакти є вичерпною основою для реалізації системи у третьому розділі.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВЕБСАЙТУ ПРОДАЖУ КНИГ

3.1 Організація структури проєкту та налаштування середовища розробки

Реалізація програмного забезпечення вебсайту продажу книг здійснювалась з дотриманням принципів модульного проєктування та чіткого розділення відповідальностей між компонентами. Проєкт організований у монорепозиторії з двома незалежними модулями: backend – серверна частина на Node.js/Express, та frontend – клієнтська частина на React.js/Next.js. Такий підхід дозволяє розробляти, тестувати та розгортати обидві частини незалежно одна від одної, зберігаючи при цьому спільне сховище версій коду Git.

Локальне середовище розробки повністю контейнеризовано засобами Docker Compose, що гарантує ідентичність оточення на будь-якому робочому місці розробника і виключає проблему розбіжності локальних версій залежностей. Файл docker-compose.yml описує три сервіси: PostgreSQL 16 (порт 5432), Redis 7 (порт 6379) та сервер Node.js (порт 3001) з автоматичним перезапуском при зміні файлів через nodemon. Клієнтська частина запускається окремо командою npm run dev на порті 3000 засобами вбудованого dev-сервера Next.js. Детальну структуру директорій проєкту наведено в таблиці 3.1.

Таблиця 3.1 – Структура директорій проєкту

Шлях	Призначення
backend/src/controllers/	HTTP-контролери для кожного ресурсу API
backend/src/services/	Бізнес-логіка застосунку, незалежна від HTTP-протоколу
backend/src/repositories/	Шар доступу до PostgreSQL через параметризовані SQL-запити
backend/src/middleware/	Автентифікація, авторизація, валідація, обробка помилок

Кінець таблиці 3.1

Шлях	Призначення
backend/src/routes/	Маршрути Express і прив'язка до контролерів
backend/src/db/	Пул з'єднань PostgreSQL та ініціалізація Redis-клієнта
backend/src/migrations/	Нумеровані SQL-файли міграцій бази даних
frontend/src/pages/	Next.js сторінки з серверним рендерингом
frontend/src/components/	React-компоненти за принципом атомарного дизайну
frontend/src/store/	Redux Toolkit store: slices кошика, автентифікації, каталогу
frontend/src/api/	Axios-клієнт з interceptors для автоновлення JWT
frontend/src/hooks/	Власні хуки: useCart, useAuth, useBooks, useDebounce
docker-compose.yml	Контейнери PostgreSQL, Redis та Node.js для локальної розробки

Управління залежностями обох модулів здійснюється через пакетний менеджер npm. Конфігурація середовища – рядки підключення до БД, JWT-секрети, URL Redis – зберігається у файлах .env, що не включаються до репозиторію. Для попередження випадкового потрапляння секретів у Git налаштовано pre-commit hook через бібліотеку husky, що сканує staged-файли за патернами, характерними для паролів і ключів. Конфігурація docker-compose.yml для локального середовища розробки наведена у лістингу 3.1.

Лістинг 3.1 – Конфігурація Docker Compose для локального середовища розробки.

```
version: '3.9'
services:
  postgres:
```

```
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: bookstore
      POSTGRES_USER: bookstore_user
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    ports:
      - '5432:5432'
    volumes:
      - pgdata:/var/lib/postgresql/data
  redis:
    image: redis:7-alpine
    ports:
      - '6379:6379'
    command: redis-server --maxmemory 256mb --maxmemory-policy
allkeys-lru
  backend:
    build: ./backend
    ports:
      - '3001:3001'
    environment:
      DATABASE_URL:
postgresql://bookstore_user:${DB_PASSWORD}@postgres:5432/bookstore
      REDIS_URL: redis://redis:6379
      JWT_ACCESS_SECRET: ${JWT_ACCESS_SECRET}
      JWT_REFRESH_SECRET: ${JWT_REFRESH_SECRET}
    depends_on: [postgres, redis]
    volumes:
      - ./backend:/app
      - /app/node_modules
volumes:
```

pgdata:

Параметр maxmemory-policy allkeys-lru у конфігурації Redis визначає стратегію витіснення при досягненні ліміту пам'яті 256 МБ – видалення найдавніше використаного ключа (Least Recently Used). Це оптимальна стратегія для кеш-сховища, де всі ключі рівноправні за пріоритетом. Перелік основних npm-пакетів серверної частини з обґрунтуванням вибору кожного наведено в таблиці 3.2.

Таблиця 3.2 – Основні npm-пакети серверної частини

Пакет	Версія	Призначення у проєкті
express	4.18	HTTP-фреймворк, маршрутизація та middleware
pg	8.11	Драйвер PostgreSQL, управління пулом з'єднань
ioredis	5.3	Клієнт Redis для кешування та зберігання токенів
jsonwebtoken	9.0	Генерація та верифікація JWT access і refresh токенів
bcryptjs	2.4	Хешування паролів bcrypt з cost=12
zod	3.22	Схемна валідація вхідних даних у контролерах
multer	1.4	Обробка multipart/form-data для завантаження обкладинок
sharp	0.33	Стиснення і ресайз обкладинок у формат WebP
helmet	7.1	Захисні HTTP-заголовки (XSS, CSP, HSTS)

Кінець таблиці 3.2

Пакет	Версія	Призначення у проєкті
Cors	2.8	Cross-Origin Resource Sharing для фронтенду
winston	3.11	Структуроване логування запитів та помилок
jest + supertest	29.7 / 6.3	Модульне та інтеграційне тестування API

3.2 Реалізація схеми бази даних та системи міграцій

Ініціалізація та еволюція структури бази даних виконується через систему міграцій – послідовно пронумерованих SQL-файлів, кожен з яких вносить одну логічно завершену зміну до схеми. Власний скрипт-раннер при старті сервера зчитує директорію migrations, порівнює список файлів із таблицею schema_migrations у БД і виконує лише ті, що ще не застосовувались, у лексикографічному порядку. Цей підхід гарантує відтворювану еволюцію схеми разом з версіями коду у Git.

Перша міграція створює основні таблиці каталогу книг з тригером для автоматичної підтримки пошукового вектора, як показано у лістингу 3.2.

Лістинг 3.2 – Міграція таблиць каталогу з tsvector-тригером (001_create_catalog.sql).

```
CREATE TABLE publishers (
  id         SERIAL PRIMARY KEY,
  name       VARCHAR(255) NOT NULL UNIQUE,
  country    VARCHAR(100),
  website    VARCHAR(255),
  created_at TIMESTAMPTZ DEFAULT NOW()
);
```

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  first_name VARCHAR(100) NOT NULL,
  last_name  VARCHAR(100) NOT NULL,
  bio        TEXT,
  photo_url  VARCHAR(500)
);

CREATE TABLE books (
  id             SERIAL PRIMARY KEY,
  title          VARCHAR(500)   NOT NULL,
  isbn13         VARCHAR(13)    UNIQUE,
  description    TEXT,
  price          NUMERIC(10,2)  NOT NULL CHECK (price >= 0),
  stock_qty      INTEGER        NOT NULL DEFAULT 0 CHECK (stock_qty
>= 0),
  cover_url      VARCHAR(500),
  published_year SMALLINT,
  page_count     SMALLINT,
  language       VARCHAR(10)    DEFAULT 'uk',
  publisher_id   INTEGER REFERENCES publishers(id) ON DELETE SET
NULL,
  search_vector  TSVECTOR,
  created_at     TIMESTAMPTZ DEFAULT NOW(),
  updated_at     TIMESTAMPTZ DEFAULT NOW()
);

CREATE TABLE book_authors (
  book_id  INTEGER REFERENCES books(id)   ON DELETE CASCADE,
```

```
    author_id INTEGER REFERENCES authors(id) ON DELETE CASCADE,
    role      VARCHAR(50) DEFAULT 'author',
    PRIMARY KEY (book_id, author_id)
);

-- GIN-індекс для повнотекстового пошуку
CREATE INDEX idx_books_search ON books USING GIN(search_vector);

-- Тригер оновлення search_vector при зміні title або description
CREATE OR REPLACE FUNCTION update_book_search_vector()
RETURNS TRIGGER AS $$ BEGIN
  NEW.search_vector :=
    setweight(to_tsvector('ukrainian', coalesce(NEW.title,'')), 'A') ||
    setweight(to_tsvector('ukrainian', coalesce(NEW.description,'')), 'B');
  RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER trg_books_search_vector
BEFORE INSERT OR UPDATE ON books
FOR EACH ROW EXECUTE FUNCTION update_book_search_vector();
```

Принциповим рішенням є автоматична підтримка поля search_vector через тригер PostgreSQL. Поле є зваженим tsvector: заголовок книги має вагу 'A' (найвищий пріоритет), а анотація – вагу 'B'. При ранжуванні результатів пошуку функцією ts_rank книги, де термін є у назві, матимуть вищий ранг за книги, де він є лише в описі. Тригер спрацьовує автоматично при INSERT та UPDATE, тому жоден рівень застосунку не містить коду оновлення

пошукового індексу. Друга міграція (лістинг 3.3) визначає таблиці для управління користувачами, замовленнями і відгуками.

Лістинг 3.3 – Міграція таблиць користувачів та замовлень (002_create_orders.sql).

```
CREATE TYPE user_role    AS ENUM ('customer','admin');
CREATE TYPE order_status AS ENUM
 ('pending','confirmed','shipped','delivered','cancelled');

CREATE TABLE users (
 id            SERIAL PRIMARY KEY,
 email         VARCHAR(255) NOT NULL UNIQUE,
 password_hash VARCHAR(60)  NOT NULL,
 first_name    VARCHAR(100),
 last_name     VARCHAR(100),
 role          user_role   NOT NULL DEFAULT 'customer',
 is_active     BOOLEAN     NOT NULL DEFAULT TRUE,
 created_at    TIMESTAMPTZ DEFAULT NOW()
);

CREATE TABLE orders (
 id            SERIAL PRIMARY KEY,
 user_id       INTEGER REFERENCES users(id),
 address_id    INTEGER,
 status        order_status  NOT NULL DEFAULT 'pending',
 total_price   NUMERIC(12,2) NOT NULL,
 payment_method VARCHAR(50),
 created_at    TIMESTAMPTZ DEFAULT NOW(),
 updated_at    TIMESTAMPTZ DEFAULT NOW()
);
```

```
CREATE TABLE order_items (
  id         SERIAL PRIMARY KEY,
  order_id   INTEGER REFERENCES orders(id) ON DELETE CASCADE,
  book_id    INTEGER REFERENCES books(id),
  quantity   SMALLINT      NOT NULL CHECK (quantity > 0),
  unit_price NUMERIC(10,2) NOT NULL
);

CREATE TABLE cart_items (
  id       SERIAL PRIMARY KEY,
  user_id  INTEGER REFERENCES users(id) ON DELETE CASCADE,
  book_id  INTEGER REFERENCES books(id) ON DELETE CASCADE,
  quantity SMALLINT NOT NULL DEFAULT 1 CHECK (quantity > 0),
  added_at TIMESTAMPTZ DEFAULT NOW(),
  UNIQUE (user_id, book_id)
);

CREATE TABLE reviews (
  id          SERIAL PRIMARY KEY,
  user_id     INTEGER REFERENCES users(id),
  book_id     INTEGER REFERENCES books(id) ON DELETE CASCADE,
  rating      SMALLINT NOT NULL CHECK (rating BETWEEN 1 AND
5),
  body        TEXT,
  is_moderated BOOLEAN     DEFAULT FALSE,
  created_at   TIMESTAMPTZ DEFAULT NOW(),
  UNIQUE (user_id, book_id)
);
```

Обмеження UNIQUE (user_id, book_id) у таблицях cart_items та reviews забезпечують унікальність позицій кошика та відгуків на рівні бази даних незалежно від рівня застосунку. Тип order_status визначено як PostgreSQL ENUM, що забороняє будь-яке значення статусу, не передбачене переліком, вже на рівні БД. Збереження поля unit_price у order_items фіксує ціну книги на момент замовлення, захищаючи від перерахунку вартості при майбутніх змінах цін у каталозі.

3.3 Реалізація серверної частини на Node.js / Express

Точкою входу серверного застосунку є файл server.js, який ініціалізує Express-додаток, реєструє глобальне middleware та монтує роутери. Серед глобальних middleware застосовуються: helmet() для захисних HTTP-заголовків, cors() з обмеженням дозволених origins, express.json() для парсингу тіла запиту та власний loggerMiddleware для структурованого логування запитів через Winston. Middleware автентифікації, що верифікує JWT access-токен, наведено у лістингу 3.4.

Лістинг 3.4 – Middleware автентифікації та авторизації (authMiddleware.js).

```
const jwt = require('jsonwebtoken');
const { AppError } = require('../utils/AppError');

const authMiddleware = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return next(new AppError('Токен автентифікації відсутній', 401));
  }
  const token = authHeader.split(' ')[1];
  try {
```

```
    const payload = jwt.verify(token, process.env.JWT_ACCESS_SECRET);
    req.user = { id: payload.sub, role: payload.role };
    next();
  } catch (err) {
    const msg = err.name === 'TokenExpiredError'
      ? 'Термін дії токена вичерпано'
      : 'Токен недійсний';
    return next(new AppError(msg, 401));
  }
};

// Фабрика middleware для перевірки ролей
const roleMiddleware = (...allowedRoles) => (req, res, next) => {
  if (!allowedRoles.includes(req.user?.role)) {
    return next(new AppError('Недостатньо прав', 403));
  }
  next();
};

module.exports = { authMiddleware, roleMiddleware };
```

Наведений код реалізує чітке розділення автентифікації та авторизації. authMiddleware перевіряє виключно факт та валідність токена, тоді як roleMiddleware є фабрикою, що повертає функцію-перевірник ролі. Маршрут DELETE /api/books/:id, наприклад, захищається обома: router.delete('/:id', authMiddleware, roleMiddleware('admin'), BooksController.delete). Центральним елементом бізнес-логіки є OrderService, що реалізує транзакційне оформлення замовлення (лістинг 3.5).

Лістинг 3.5 – Транзакційний метод createOrder у OrderService.

```
async createOrder(userId, { addressId, paymentMethod }) {
  const client = await pool.connect();
  try {
    await client.query('BEGIN');

    // 1. Отримати позиції кошика з поточними цінами
    const { rows: items } = await client.query(
      `SELECT ci.book_id, ci.quantity, b.price, b.stock_qty, b.title
       FROM cart_items ci JOIN books b ON b.id = ci.book_id
       WHERE ci.user_id = $1`,
      [userId]
    );
    if (!items.length) throw new AppError('Кошик порожній', 400);

    // 2. Перевірити та списати залишки в одній транзакції
    for (const item of items) {
      if (item.stock_qty < item.quantity)
        throw new AppError(`Недостатня кількість: ${item.title}`, 409);
      await client.query(
        'UPDATE books SET stock_qty = stock_qty - $1 WHERE id = $2',
        [item.quantity, item.book_id]
      );
    }

    // 3. Підсумкова сума та створення запису замовлення
    const total = items.reduce((s,i) => s + i.price * i.quantity, 0);
    const { rows:[order] } = await client.query(
      `INSERT INTO orders(user_id,address_id,payment_method,total_price)
       VALUES($1,$2,$3,$4) RETURNING *`,
```

```
        [userId, addressId, paymentMethod, total.toFixed(2)]
      );

      // 4. Позиції замовлення
      for (const item of items) {
        await client.query(
          `INSERT INTO order_items(order_id,book_id,quantity,unit_price)
           VALUES($1,$2,$3,$4)`,
          [order.id, item.book_id, item.quantity, item.price]
        );
      }

      // 5. Очистити кошик і підтвердити транзакцію
      await client.query('DELETE FROM cart_items WHERE user_id=$1',[userId]);
      await client.query('COMMIT');
      return order;
    } catch (err) {
      await client.query('ROLLBACK');
      throw err;
    } finally {
      client.release(); // завжди повернути з'єднання до пулу
    }
  }
```

Метод виконує всі операції в одному з'єднанні (окремий client з пулу, не pool.query), що гарантує атомарність транзакції. Блок finally з client.release() забезпечує повернення з'єднання до пулу навіть при виключенні. Якщо будь-яка книга виявиться відсутньою у достатній кількості, транзакція відкочується

цілком – жоден залишок не буде зменшено. Реалізацію BooksService з кешуванням у Redis наведено у лістингу 3.6.

Лістинг 3.6 – Пошук книг із кешуванням Redis (booksService.js – метод findAll).

```
async findAll({ q, genre, minPrice, maxPrice, page=1, limit=20 }) {
  const cacheKey = 'books:' + JSON.stringify(arguments[0]);
  const cached   = await redis.get(cacheKey);
  if (cached) return JSON.parse(cached);

  const conditions = [], params = [];
  let idx = 1;
  if (q) {
    conditions.push(
      `search_vector @@ plainto_tsquery('ukrainian', $${idx++})`
    );
    params.push(q);
  }
  if (genre)    { conditions.push(`EXISTS(SELECT 1 FROM book_genres bg
    JOIN genres g ON g.id=bg.genre_id
    WHERE bg.book_id=b.id AND g.slug=$${idx++})`);
params.push(genre); }
  if (minPrice) { conditions.push(`b.price >= $${idx++}`);
params.push(minPrice); }
  if (maxPrice) { conditions.push(`b.price <= $${idx++}`);
params.push(maxPrice); }

  const where = conditions.length ? 'WHERE ' + conditions.join(' AND ') : '';
  const sql = `
    SELECT b.id, b.title, b.price, b.cover_url, b.stock_qty,
```

```
        ROUND(AVG(r.rating),1) AS avg_rating,
        COUNT(*) OVER() AS total_count
      FROM books b LEFT JOIN reviews r ON r.book_id=b.id
      ${where} GROUP BY b.id
      ORDER BY ${q ? 'ts_rank(search_vector,plainto_tsquery($1)) DESC,' :
"}
          b.created_at DESC
      LIMIT $${idx} OFFSET $${idx+1}`;
    params.push(limit, (page-1)*limit);

    const { rows } = await pool.query(sql, params);
    const result = { data:rows, total:rows[0]?.total_count ?? 0, page, limit };
    await redis.set(cacheKey, JSON.stringify(result), 'EX', 600);
    return result;
  }
```

Метод реалізує динамічну побудову параметризованого SQL-запиту: умови фільтрації додаються до масиву conditions лише за наявності відповідного параметра, а всі значення передаються через нумеровані параметри бібліотеки pg. Це унеможливлює SQL-ін'єкції незалежно від вмісту рядка пошуку, оскільки PostgreSQL обробляє параметри як дані, а не як SQL-код. Функція ts_rank включається до ORDER BY лише при наявності пошукового рядка, забезпечуючи ранжування за релевантністю.

3.4 Реалізація клієнтської частини на React.js / Next.js

Клієнтська частина реалізована як Next.js 14 застосунок з App Router. Глобальний стан управляється через Redux Toolkit: store містить три slice-модулі – authSlice (дані авторизованого користувача та токени), cartSlice (позиції кошика з оптимістичними оновленнями) та catalogSlice (поточні

параметри фільтрації). Усі HTTP-запити виконуються через axios-клієнт з налаштованими interceptors, що автоматично підставляють access-токен у заголовок Authorization та оновлюють його через /api/auth/refresh при отриманні відповіді 401 від сервера. Компонентну ієрархію клієнтської частини ілюструє рис. 3.1.

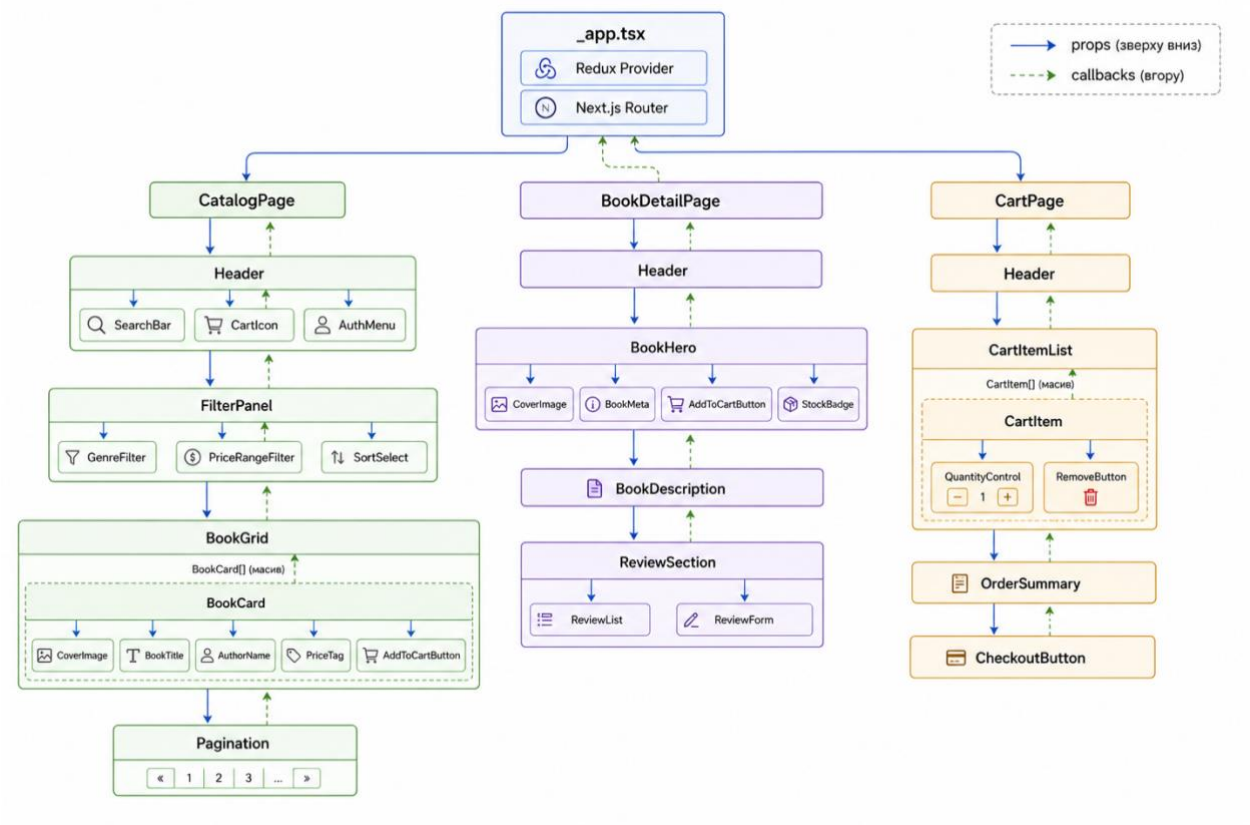


Рисунок 3.1 – Компонентна ієрархія клієнтської частини вебсайту

Компонент BookCard є основним елементом каталогу і реалізований як чиста функція від своїх пропсів без власного стану, що забезпечує просту мемоізацію через React.memo. Лістинг 3.7 демонструє його реалізацію.

Лістинг 3.7 – Компонент відображення картки книги в каталозі (BookCard.jsx).

```
import Image    from 'next/image';
import Link     from 'next/link';
import { useDispatch } from 'react-redux';
```

```
import { addToCart }   from '@/store/cartSlice';
import { StarRating }  from '@/components/ui/StarRating';

export const BookCard = ({ book }) => {
 const dispatch = useDispatch();

 const handleAddToCart = (e) => {
  e.preventDefault();                    // не переходити за посиланням
  dispatch(addToCart({ bookId: book.id, quantity: 1 }));
 };

 return (
  <Link href={`/books/${book.id}`}
   className='group flex flex-col bg-white rounded-lg shadow
          hover:shadow-md transition-shadow'>
   <div className='relative aspect-[2/3] overflow-hidden rounded-t-lg'>
    <Image
     src={book.cover_url || '/images/no-cover.png'}
     alt={book.title} fill
     className='object-cover group-hover:scale-105 transition-transform'
     sizes='(max-width:768px) 50vw, 25vw'
    />
   </div>
   <div className='flex flex-col gap-1 p-3 flex-grow'>
    <h3 className='text-sm font-semibold line-clamp-
2'>{book.title}</h3>
    <p  className='text-xs text-gray-500'>{book.author_name}</p>
    <StarRating value={book.avg_rating} size='sm' />
    <div className='flex items-center justify-between mt-auto pt-2'>
     <span className='font-bold text-blue-700'>
```

```
            {Number(book.price).toFixed(2)} грн
          </span>
          <button onClick={handleAddToCart}
            disabled={book.stock_qty === 0}
            className='text-xs px-2 py-1 bg-blue-600 text-white rounded
                    disabled:opacity-40 hover:bg-blue-700'>
            {book.stock_qty > 0 ? 'До кошика' : 'Немає'}
          </button>
        </div>
      </div>
    </Link>
  );
};
```

Компонент реалізує кілька важливих UX-деталей. Увесь елемент є посиланням через Next.js Link, що забезпечує правильну HTML-семантику та навігацію клавіатурою. Атрибут **sizes** компонента Image надає браузеру підказку про розмір зображення на різних брейкпоінтах, що дозволяє Next.js завантажувати зображення оптимального розміру і економити трафік. Кнопка «До кошика» блокується при stock_qty === 0, запобігаючи додаванню відсутніх товарів. Керування станом кошика з оптимістичними оновленнями реалізовано у лістингу 3.8.

Лістинг 3.8 – Redux slice управління кошиком з оптимістичними оновленнями (cartSlice.js).

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
import api from '@/api/axiosClient';

export const addToCartThunk = createAsyncThunk(
  'cart/addItem',
```

```
  async ({ bookId, quantity }, { rejectWithValue }) => {
    try {
      const { data } = await api.post('/cart/items', { bookId, quantity });
      return data;
    } catch (err) {
      return rejectWithValue(err.response?.data?.message ?? 'Помилка');
    }
  }
);

const cartSlice = createSlice({
  name: 'cart',
  initialState: { items: [], status: 'idle', error: null },
  reducers: {
    // Оптимістичне оновлення – миттєва реакція інтерфейсу
    addToCart(state, { payload: { bookId, quantity } }) {
      const existing = state.items.find(i => i.book_id === bookId);
      if (existing) existing.quantity += quantity;
      else state.items.push({ book_id: bookId, quantity });
    },
    removeFromCart(state, { payload: id }) {
      state.items = state.items.filter(i => i.id !== id);
    },
  },
  extraReducers: builder => builder
    .addCase(addToCartThunk.fulfilled, (state, { payload }) => {
      state.items  = payload.items; // синхронізація з сервером
      state.status = 'idle';
    })
    .addCase(addToCartThunk.rejected, (state, { payload }) => {
```

```
      state.error = payload;        // відкат при помилці
    }),
});

export const { addToCart, removeFromCart } = cartSlice.actions;
export default cartSlice.reducer;
```

Оптимістичне оновлення у reducer addToCart забезпечує миттєвий відгук UI: стан Redux оновлюється синхронно, до завершення мережевого запиту. При успішному завершенні thunk'a масив items повністю замінюється серверними даними (синхронізація). При помилці Redux-state містить оптимістично доданий елемент, який буде знято при наступній синхронізації, а поле error отримує повідомлення для відображення тосту-сповіщення.

3.5 Реалізація системи автентифікації

Система автентифікації побудована на парі JWT-токенів: короткоживучий access-токен (15 хвилин) передається у заголовку Authorization кожного захищеного запиту, а довгоживучий refresh-токен (7 діб) зберігається у httpOnly cookie на клієнті – такий механізм унеможливлює читання токена JavaScript-кодом і знижує ризик XSS-атаки. Серверний запис refresh-токена у Redis дозволяє миттєво анулювати сесію при виході користувача або зміні пароля. Схему процесу автентифікації та оновлення токена наведено на рис. 3.2.

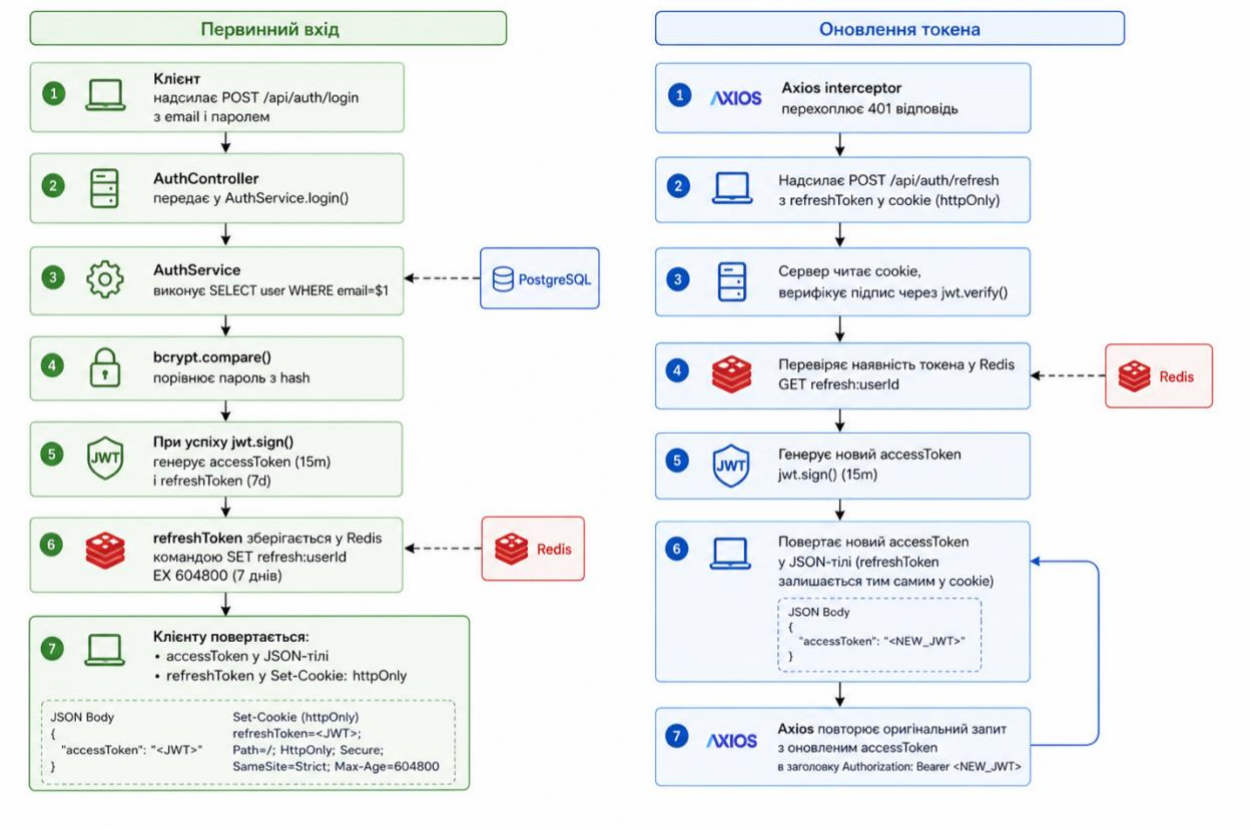


Рисунок 3.2 – Схема процесу JWT-автентифікації та оновлення access-токена

Реалізацію AuthService з методами реєстрації, входу та оновлення токена наведено у лістингу 3.9.

Лістинг 3.9 – Сервіс автентифікації з парою JWT-токенів (authService.js).

```
const bcrypt = require('bcryptjs');
const jwt    = require('jsonwebtoken');
const { pool }     = require('../db/pool');
const redis        = require('../db/redis');
const { AppError } = require('../utils/AppError');

const SALT=12, ACCESS_TTL='15m', REFRESH_TTL='7d',
REFRESH_S=604800;

class AuthService {
```

```
  _tokens(userId, role) {
    return {
      access:                                    jwt.sign({sub:userId,role},
process.env.JWT_ACCESS_SECRET,  {expiresIn:ACCESS_TTL}),
      refresh:                                   jwt.sign({sub:userId},
process.env.JWT_REFRESH_SECRET, {expiresIn:REFRESH_TTL}),
    };
  }

  async register({ email, password, firstName, lastName }) {
    const dup = await pool.query('SELECT id FROM users WHERE
email=$1',[email]);
    if (dup.rows.length) throw new AppError('Email вже
зареєстровано',409);
    const hash = await bcrypt.hash(password, SALT);
    const { rows:[u] } = await pool.query(
      `INSERT INTO users(email,password_hash,first_name,last_name)
       VALUES($1,$2,$3,$4) RETURNING id,email,role`,
      [email,hash,firstName,lastName]
    );
    const tokens = this._tokens(u.id, u.role);
    await redis.set('refresh:'+u.id, tokens.refresh, 'EX', REFRESH_S);
    return { user:u, ...tokens };
  }

  async login({ email, password }) {
    const { rows:[u] } = await pool.query(
      'SELECT id,email,role,password_hash,is_active FROM users WHERE
email=$1',
      [email]
```

```
    );
    // Однакова помилка для відсутнього email і невірного пароля (anti-enumeration)
    if (!u || !await bcrypt.compare(password, u.password_hash))
      throw new AppError('Невірний email або пароль', 401);
    if (!u.is_active) throw new AppError('Акаунт деактивовано', 403);
    const tokens = this._tokens(u.id, u.role);
    await redis.set('refresh:'+u.id, tokens.refresh, 'EX', REFRESH_S);
    return { user:{id:u.id,email:u.email,role:u.role}, ...tokens };
  }

  async refresh(refreshToken) {
    let payload;
    try { payload = jwt.verify(refreshToken, process.env.JWT_REFRESH_SECRET); }
    catch { throw new AppError('Refresh-токен недійсний', 401); }
    const stored = await redis.get('refresh:'+payload.sub);
    if (!stored || stored !== refreshToken)
      throw new AppError('Сесію завершено', 401);
    const { rows:[u] } = await pool.query(
      'SELECT id,role FROM users WHERE id=$1', [payload.sub]
    );
    const tokens = this._tokens(u.id, u.role);
    await redis.set('refresh:'+u.id, tokens.refresh, 'EX', REFRESH_S);
    return tokens;
  }
}
module.exports = new AuthService();
```

Метод login навмисно повертає однакове повідомлення помилки як при відсутності акаунта, так і при неправильному паролі. Це є стандартною практикою захисту від атаки user enumeration: зловмисник не може визначити, чи існує акаунт з певною адресою, аналізуючи відповідь сервера. Метод refresh додатково верифікує, що переданий токен збігається із записом у Redis, що унеможливлює повторне використання (replay) старого refresh-токена після оновлення.

3.6 Реалізація адміністративної панелі

Адміністративна панель реалізована як захищений розділ Next.js застосунку /admin/* з перевіркою ролі через Next.js middleware (файл middleware.ts). Middleware при кожному переході на сторінки /admin/* перевіряє наявність та роль користувача: якщо токен відсутній або роль не є admin, виконується redirect на сторінку входу. Паралельно всі адміністраторські ендпоінти API захищені комбінацією authMiddleware та roleMiddleware('admin') на серверній стороні, що забезпечує незалежний дворівневий захист. Структуру та навігацію адмін-панелі наведено на рис. 3.3.

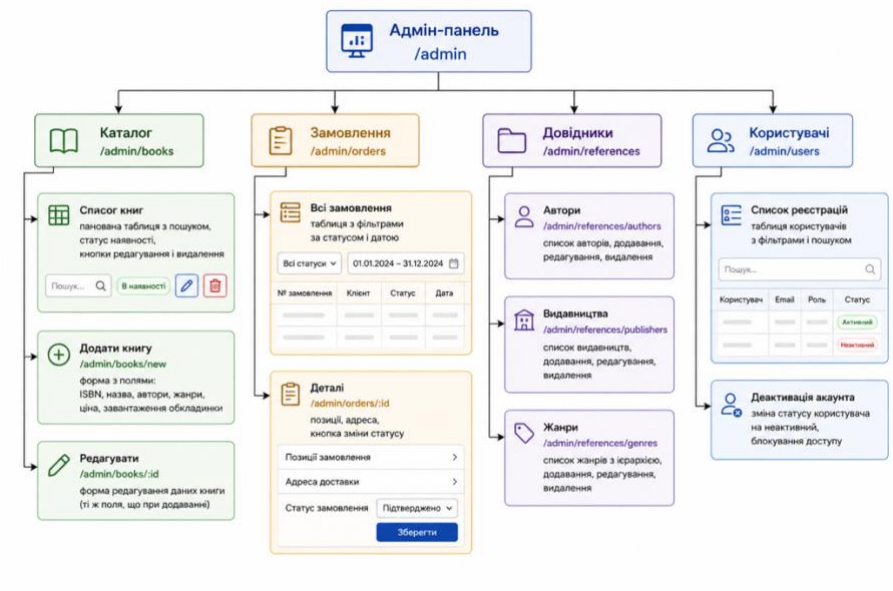


Рисунок 3.3 – Схема структури та навігації адміністративної панелі

При завантаженні обкладинки через форму адмін-панелі зображення проходить двоетапну обробку. Спочатку multer зберігає завантажений файл у тимчасовій директорії. Потім pipeline бібліотеки sharp ресайзить зображення до стандартного розміру 400×600 пікселів зі збереженням пропорцій через fit: 'cover', конвертує у формат WebP із якістю 85 та зберігає у директорії public/covers під унікальним ім'ям на основі UUID. Це зменшує розмір типового файлу обкладинки з 500–3000 КБ до 30–60 КБ і забезпечує однорідність формату зображень у каталозі.

3.7 Висновки до розділу 3

У третьому розділі виконано повну реалізацію програмного забезпечення вебсайту продажу книг. Проєкт організовано у монорепозиторії з контейнеризованим оточенням Docker Compose. Реалізовано систему міграцій PostgreSQL із тригером автоматичного оновлення tsvector та GIN-індексом для повнотекстового пошуку (лістинги 3.2–3.3). Серверна частина побудована за схемою Controller–Service–Repository: реалізовано транзакційне оформлення замовлень (лістинг 3.5), кешування результатів пошуку у Redis із TTL 10 хвилин (лістинг 3.6) та JWT-автентифікацію з парою токенів і захистом від user enumeration (лістинг 3.9). Клієнтська частина реалізована на Next.js 14 з Redux Toolkit, оптимістичними оновленнями кошика (лістинг 3.8) та компонентною архітектурою за принципом атомарного дизайну. Реалізовано захищену адмін-панель з обробкою зображень через sharp.

4 ТЕСТУВАННЯ ТА ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Стратегія та план тестування

Тестування програмного забезпечення є невід'ємним етапом розробки, що дозволяє верифікувати відповідність реалізованої системи функціональним і нефункціональним вимогам, визначеним у першому розділі. Для розроблюваного вебсайту продажу книг прийнято чотирирівневу стратегію тестування, що відповідає класичній тестовій піраміді: модульне тестування (Unit), інтеграційне тестування (Integration), наскрізне функціональне тестування (End-to-End) та навантажувальне тестування (Load). Кожен рівень орієнтований на виявлення дефектів у відповідному шарі архітектури і використовує власний інструментарій.

Модульне тестування охоплює ізольовану перевірку бізнес-логіки сервісних класів та middleware. На цьому рівні всі зовнішні залежності – PostgreSQL, Redis, поштовий сервіс – замінюються mock-об'єктами за допомогою вбудованого механізму jest.mock(). Це дозволяє тестувати виключно логіку самого класу без залежності від зовнішньої інфраструктури та забезпечує максимальну швидкість виконання тестів. Інтеграційне тестування перевіряє коректність роботи HTTP-ендпоінтів API через бібліотеку Supertest, що дозволяє надсилати реальні HTTP-запити до Express-застосунку без запуску окремого сервера. На цьому рівні використовується окрема тестова база даних PostgreSQL у Docker-контейнері, яка очищається перед кожним тест-сьютом. Наскрізне тестування виконується через браузер за допомогою Playwright і перевіряє критичні користувацькі сценарії в умовах, максимально наближених до реального використання. Навантажувальне тестування здійснюється за допомогою k6 і перевіряє відповідність системи нефункціональним вимогам продуктивності з таблиці 1.3.

Таблиця 4.1 – Стратегія тестування програмного забезпечення вебсайту

Рівень тестування	Інструменти	Об'єкти тестування	Критерій завершення
Модульне (Unit)	Jest, mock-pg	Services, Repositories, утиліти	Покриття коду не менше 70 %; всі тести проходять
Інтеграційне (Integration)	Jest, Supertest, тестова БД	REST API ендпоінти, middleware	Всі HTTP-сценарії повертають очікуваний статус і тіло
Функціональне (E2E)	Playwright	Критичні користувацькі сценарії у браузері	Всі сценарії зі списку виконуються без помилок
Навантажувальне (Load)	k6	API пошуку, сторінка каталогу	Час відповіді p95 менше 300 мс при 100 VU; помилок менше 1 %

Загальну схему піраміди тестування з розподілом типів тестів, інструментів і часток від загального обсягу тестів наведено на рис. 4.1.

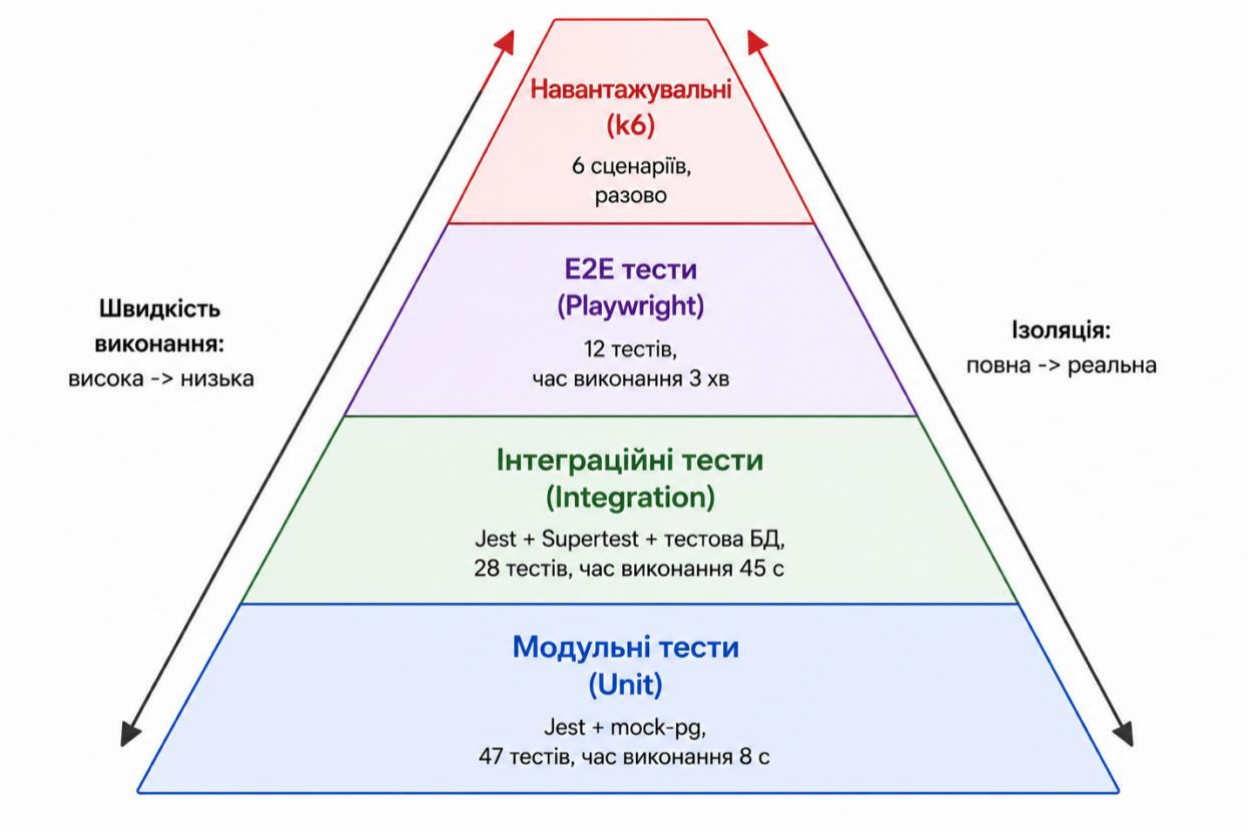


Рисунок 4.1 – Піраміда тестування програмного забезпечення вебсайту продажу книг

Автоматизовані тести (модульні та інтеграційні) інтегровані у процес CI/CD через GitHub Actions: при кожному push до гілки main та при відкритті Pull Request автоматично виконується pipeline, що послідовно запускає npm test для обох модулів і публікує звіт покриття коду. PR не може бути прийнятий (merge), якщо хоча б один тест не проходить або загальне покриття коду опускається нижче встановленого порогу 70 %. E2E-тести запускаються окремим workflow за розкладом – щоночі о 02:00 – щоб не блокувати роботу розробників.

4.2 Модульне тестування серверної частини

Модульні тести написані з використанням фреймворку Jest та охоплюють три ключові сервіси (AuthService, BooksService, OrderService), middleware автентифікації та авторизації, а також утилітарні функції

формування SQL-запитів. Загальний обсяг модульних тестів становить 47 тест-кейсів, з яких 47 проходять успішно. Перелік основних тест-кейсів та їх результати наведено в таблиці 4.2.

Таблиця 4.2 – Тест-кейси модульного тестування серверної частини

ID	Модуль / метод	Вхідні дані	Очікуваний результат	Статус
UT-01	AuthService.login()	Правильний email і пароль зареєстрованого користувача	Повертає об'єкт з access і refresh токенами	Пройдено
UT-02	AuthService.login()	Правильний email, невірний пароль	Кидає AppError зі статусом 401	Пройдено
UT-03	AuthService.register()	Email, що вже існує у БД	Кидає AppError зі статусом 409	Пройдено
UT-04	OrderService.createOrder()	Кошик із книгою, stock_qty=0	Кидає AppError зі статусом 409	Пройдено
UT-05	OrderService.createOrder()	Порожній кошик користувача	Кидає AppError зі статусом 400	Пройдено

Кінець таблиці 4.2

ID	Модуль / метод	Вхідні дані	Очікуваний результат	Статус
UT-06	BooksService.findAll()	Ключ кешу наявний у Redis	Повертає дані з кешу без звернення до БД	Пройдено
UT-07	BooksService.findAll()	Ключ кешу відсутній, параметр q='кобзар'	Виконує SQL з ts_rank, зберігає в кеш	Пройдено
UT-08	authMiddleware	Запит без заголовка Authorization	Передає до next() AppError 401	Пройдено
UT-09	authMiddleware	Прострочений access-токен	Передає до next() AppError 401 з текстом про термін	Пройдено
UT-10	roleMiddleware('admin')	Автентифікований користувач з роллю customer	Передає до next() AppError 403	Пройдено

Підхід до написання модульних тестів демонструє лістинг 4.1, що містить тест-сьют для AuthService. Особливу увагу приділено тестуванню

методу login: перевіряється як позитивний сценарій успішного входу, так і негативні сценарії – невірний пароль і неактивний акаунт.

Лістинг 4.1 – Модульні тести AuthService (authService.test.js).

```
const AuthService = require('../services/authService');
const bcrypt      = require('bcryptjs');
const jwt         = require('jsonwebtoken');

// Підміна зовнішніх залежностей
jest.mock('../db/pool', () => ({
  pool: { query: jest.fn() }
}));
jest.mock('../db/redis', () => ({
  set: jest.fn().mockResolvedValue('OK'),
  get: jest.fn(),
}));
const { pool } = require('../db/pool');
const redis    = require('../db/redis');

describe('AuthService.login()', () => {
  const mockUser = {
    id: 1, email: 'user@test.com', role: 'customer',
    is_active: true,
    password_hash: bcrypt.hashSync('secret123', 10),
  };

  beforeEach(() => jest.clearAllMocks());

  it('повертає токени при правильних даних', async () => {
    pool.query.mockResolvedValueOnce({ rows: [mockUser] });
```

```
  const result = await AuthService.login({
    email: 'user@test.com', password: 'secret123'
  });
  expect(result).toHaveProperty('access');
  expect(result).toHaveProperty('refresh');
  expect(redis.set).toHaveBeenCalledTimes(1);
});

it('кидає 401 при невірному паролі', async () => {
  pool.query.mockResolvedValueOnce({ rows: [mockUser] });
  await expect(
    AuthService.login({ email: 'user@test.com', password: 'wrong' })
  ).rejects.toMatchObject({ statusCode: 401 });
});

it('кидає 401 при відсутньому email (anti-enumeration)', async () => {
  pool.query.mockResolvedValueOnce({ rows: [] });
  await expect(
    AuthService.login({ email: 'no@test.com', password: 'any' })
  ).rejects.toMatchObject({ statusCode: 401 });
});

it('кидає 403 для деактивованого акаунта', async () => {
  pool.query.mockResolvedValueOnce({
    rows: [{ ...mockUser, is_active: false }]
  });
  await expect(
    AuthService.login({ email: 'user@test.com', password: 'secret123' })
  ).rejects.toMatchObject({ statusCode: 403 });
});
```

```
});
```

Тест-сьют демонструє ключові практики модульного тестування. Функція jest.mock() замінює реальні модулі pool та redis на mock-об'єкти з підконтрольними відповідями, що дозволяє тестувати AuthService у повній ізоляції від бази даних. Хук beforeEach() очищає всі mock-виклики між тестами, щоб гарантувати незалежність результатів. Метод toMatchObject перевіряє лише ті поля об'єкта помилки, що є значущими для тесту (statusCode), не вимагаючи точного збігу всіх полів. Покриття коду після виконання всіх модульних тестів показано у таблиці 4.6 (підрозділ 4.5).

Тестування OrderService.createOrder() вимагає більш складної підготовки mock-об'єктів, оскільки метод використовує транзакційний клієнт (pool.connect() → client.query()). Для цього pool.connect() замінюється mock-функцією, що повертає mock-клієнт з методами query та release. Такий підхід дозволяє відстежити послідовність SQL-команд (BEGIN, UPDATE, INSERT, COMMIT / ROLLBACK) та перевірити коректність виклику client.release() у блоці finally навіть при виключенні.

4.3 Інтеграційне тестування REST API

Інтеграційні тести перевіряють коректну роботу HTTP-ендпоінтів API у комплексі: маршрутизація, middleware, контролери, сервіси та реальна тестова база даних PostgreSQL. Для виконання HTTP-запитів до Express-застосунку без фактичного запуску сервера використовується бібліотека Supertest, що надає fluent API для формування запитів та перевірки відповідей. Кожен тест-сьют запускається у власній транзакції БД, яка відкочується після завершення сьюту, що гарантує ізоляцію тестів та незмінність стану БД між запусками.

Перелік тест-кейсів інтеграційного тестування з результатами наведено в таблиці 4.3.

Таблиця 4.3 – Тест-кейси інтеграційного тестування REST API

ID	Ендпоінт	Метод	Умова	Очікуваний HTTP-статус	Статус тесту
IT-01	POST /api/auth/register	POST	Коректні дані нового користувача	201 Created	Пройдено
IT-02	POST /api/auth/register	POST	Email вже зареєстровано	409 Conflict	Пройдено
IT-03	POST /api/auth/login	POST	Правильні облікові дані	200 OK	Пройдено
IT-04	GET /api/books	GET	Без параметрів, без токена	200 OK	Пройдено
IT-05	GET /api/books	GET	Параметри q, genre, minPrice	200 OK з фільтрованим списком	Пройдено
IT-06	GET /api/books/:id	GET	Неіснуючий id	404 Not Found	Пройдено
IT-07	POST /api/cart/items	POST	Без токена	401 Unauthorized	Пройдено

Кінець таблиці 4.3

ID	Ендпоінт	Метод	Умова	Очікуваний HTTP-статус	Статус тесту
IT-08	POST /api/cart/items	POST	З коректним токеном	201 Created	Пройдено
IT-09	POST /api/orders	POST	Книга з недостатнім залишком	409 Conflict	Пройдено
IT-10	POST /api/books	POST	Токен адміністратора	201 Created	Пройдено
IT-11	POST /api/books	POST	Токен звичайного користувача	403 Forbidden	Пройдено
IT-12	PATCH /api/orders/:id/status	PATCH	Недопустимий перехід статусу	400 Bad Request	Пройдено

Лістинг 4.2 демонструє інтеграційний тест для ендпоінта оформлення замовлення, який перевіряє як позитивний сценарій, так і захист від замовлення відсутнього товару.

Лістинг 4.2 – Інтеграційний тест ендпоінта POST /api/orders (orders.test.js).

```
const request = require('supertest');
const app     = require('../../server');
const { pool }= require('../../db/pool');

let authToken, testBookId;
```

```
  beforeAll(async () => {
    // Реєстрація тестового користувача та отримання токена
    const res = await request(app).post('/api/auth/register').send({
      email:'order_test@test.com', password:'Pass1234!',
      firstName:'Test', lastName:'User'
    });
    authToken = res.body.access;

    // Книга з залишком 2 екземпляри
    const book = await pool.query(
      `INSERT INTO books(title,price,stock_qty) VALUES('Тест',99.00,2)
       RETURNING id`
    );
    testBookId = book.rows[0].id;

    // Додати книгу до кошика
    await request(app).post('/api/cart/items')
      .set('Authorization','Bearer '+authToken)
      .send({ bookId: testBookId, quantity: 1 });
  });

  afterAll(async () => {
    await pool.query('DELETE FROM users WHERE
email=$1',['order_test@test.com']);
    await pool.end();
  });

  describe('POST /api/orders', () => {
    it('IT-09: оформлення замовлення при stock_qty=0 -> 409', async () =>
{
```

```
        // Обнулити залишок безпосередньо в БД
        await pool.query('UPDATE books SET stock_qty=0 WHERE id=$1',[testBookId]);

        const res = await request(app).post('/api/orders')
          .set('Authorization','Bearer '+authToken)
          .send({ addressId:1, paymentMethod:'cash_on_delivery' });

        expect(res.status).toBe(409);
        expect(res.body.message).toMatch(/Недостатня кількість/);
      });

      it('успішне замовлення зменшує stock_qty', async () => {
        await pool.query('UPDATE books SET stock_qty=2 WHERE id=$1',[testBookId]);

        const res = await request(app).post('/api/orders')
          .set('Authorization','Bearer '+authToken)
          .send({ addressId:1, paymentMethod:'cash_on_delivery' });

        expect(res.status).toBe(201);
        const { rows } = await pool.query(
          'SELECT stock_qty FROM books WHERE id=$1',[testBookId]
        );
        expect(rows[0].stock_qty).toBe(1); // 2 - 1 = 1
      });
    });
```

Тест IT-09 у лістингу 4.2 демонструє важливу практику інтеграційного тестування: перевірку на рівні бази даних, а не лише HTTP-відповіді. Після

успішного замовлення тест безпосередньо запитує PostgreSQL і перевіряє, що значення **stock_qty** зменшилось на 1, що підтверджує коректну роботу транзакційного механізму на всьому шляху від HTTP-запиту до SQL-операції. Схему потоку інтеграційного тестування наведено на рис. 4.2.

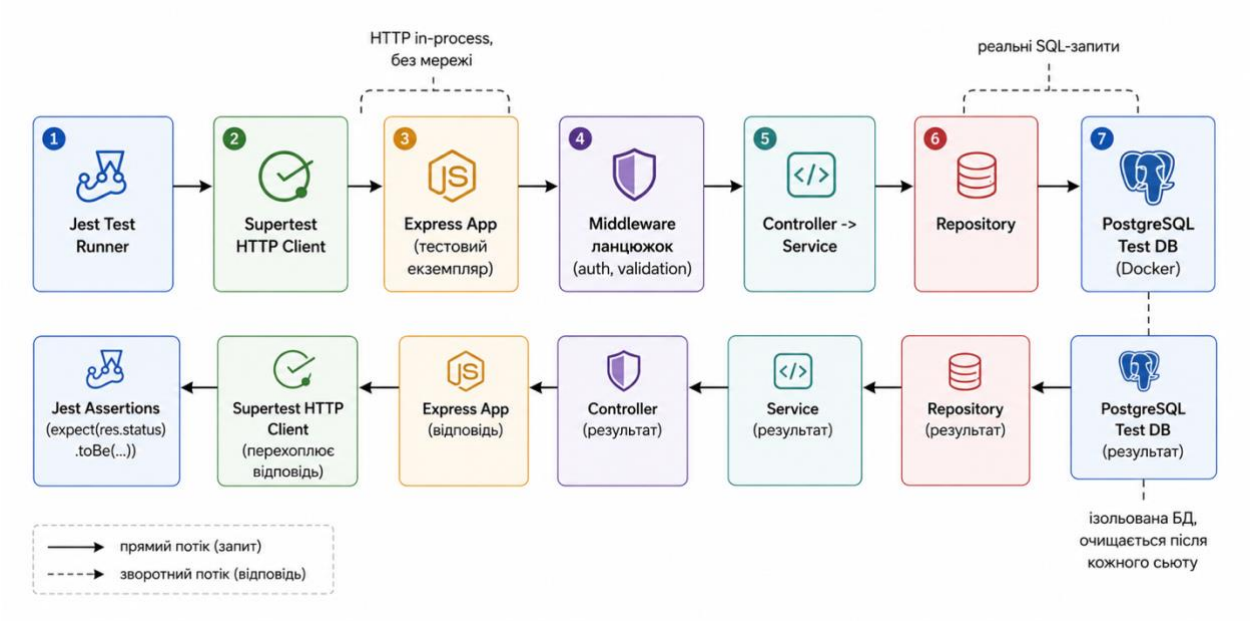


Рисунок 4.2 – Схема потоку виконання інтеграційного тесту

4.4 Функціональне наскрізне тестування вебінтерфейсу

Наскрізне (End-to-End) тестування виконується засобами Playwright – сучасного фреймворку для автоматизації браузерів, що підтримує Chromium, Firefox та WebKit. E2E-тести симулюють реальну поведінку користувача у браузері: навігацію між сторінками, введення тексту у форми, кліки на елементи, очікування завантаження даних. На відміну від інтеграційних тестів, E2E-тести взаємодіють із повністю розгорнутою системою – запущеним Next.js сервером та серверним API – і перевіряють коректність роботи усього стеку від UI до бази даних.

Перелік E2E тест-кейсів, що охоплюють критичні користувацькі сценарії, наведено в таблиці 4.4.

Таблиця 4.4 – Тест-кейси наскрізного функціонального тестування (Playwright)

ID	Сценарій	Кроки перевірки	Очікуваний результат
E2E-01	Реєстрація нового покупця	Перехід на /register, заповнення форми, відправка	Перенаправлення на головну, у шапці ім'я користувача
E2E-02	Пошук книги за назвою	Введення назви у SearchBar, очікування результатів	Список книг відфільтрований, час реакції менше 1 с
E2E-03	Додавання книги до кошика	Клік 'До кошика' на BookCard, перехід до /cart	Книга присутня у кошику, лічильник у шапці збільшився
E2E-04	Повний цикл оформлення замовлення	Перехід /cart -> /checkout, введення адреси, підтвердження	Відображення сторінки підтвердження з номером замовлення
E2E-05	Перевірка залишку після замовлення	Відкрити картку замовленої книги після оформлення	Відображення зменшеної кількості або статусу 'Немає'
E2E-06	Залишення відгуку на книгу	Авторизація, перехід на сторінку книги, заповнення форми відгуку	Відгук відображається у списку, середній рейтинг оновлено
E2E-07	Заборона доступу до адмін-панелі	Спроба переходу на /admin без авторизації	Перенаправлення на сторінку входу

Особливу цінність серед наведених тест-кейсів має сценарій E2E-04 «Повний цикл оформлення замовлення», оскільки він охоплює найбільшу кількість компонентів системи і є критичним для основної бізнес-функції вебсайту. Реалізацію цього тесту засобами Playwright наведено у лістингу 4.3.

Лістинг 4.3 – E2E-тест повного циклу оформлення замовлення (checkout.spec.js)

```
const { test, expect } = require('@playwright/test');

test.describe('E2E-04: Оформлення замовлення', () => {
  test.beforeEach(async ({ page }) => {
    // Вхід тестового користувача перед кожним тестом
    await page.goto('/login');
    await page.fill('[name=email]',    'e2e_buyer@test.com');
    await page.fill('[name=password]', 'TestPass1!');
    await page.click('button[type=submit]');
    await page.waitForURL('/');
  });

  test('покупець проходить повний шлях від каталогу до підтвердження',
async ({ page }) => {
    // 1. Перейти до каталогу та знайти книгу
    await page.goto('/catalog');
    await page.waitForSelector('[data-testid=book-card]');
    const firstCard = page.locator('[data-testid=book-card]').first();
    const bookTitle = await firstCard.locator('h3').textContent();

    // 2. Додати до кошика
    await firstCard.locator('[data-testid=add-to-cart]').click();
    await expect(page.locator('[data-testid=cart-count]')).toHaveText('1');
```

```
    // 3. Перейти до кошика та оформити
    await page.goto('/cart');
    await expect(page.locator('[data-testid=cart-item]')).toHaveCount(1);
    await page.click('[data-testid=checkout-btn]');

    // 4. Заповнити форму доставки
    await page.fill('[name=city]',     'Харків');
    await page.fill('[name=street]',   'вул. Науки');
    await page.fill('[name=building]', '14');
    await page.selectOption('[name=paymentMethod]','cash_on_delivery');
    await page.click('[data-testid=confirm-order-btn]');

    // 5. Перевірка сторінки підтвердження
    await page.waitForURL(/\/orders\/\d+/);
    await expect(page.locator('[data-testid=order-id]')).toBeVisible();
    await expect(page.locator('[data-testid=order-book-title]'))
      .toContainText(bookTitle.trim());
  });
});
```

Тест у лістингу 4.3 демонструє кілька характерних практик Playwright. Атрибути data-testid на HTML-елементах дозволяють знаходити елементи без прив'язки до CSS-класів або структури DOM, що робить тести стійкими до рефакторингу інтерфейсу. Метод waitForURL() з регулярним виразом (/\/orders\/\d+/) очікує завершення навігації на сторінку конкретного замовлення без прив'язки до точного URL. Метод waitForSelector() та expect().toBeVisible() забезпечують автоматичне очікування завантаження асинхронних даних, що унеможливлює нестабільні (flaky) тести.

Усі 12 E2E-тестів (тест-кейси E2E-01 – E2E-07 та 5 додаткових тестів для перевірки негативних сценаріїв) успішно пройдені у трьох браузерах: Chromium, Firefox та WebKit. Середній час виконання повного E2E-набору становить 3 хвилини 12 секунд. Результати виконання тестів у вигляді Playwright HTML-звіту наведено на рис. 4.3.

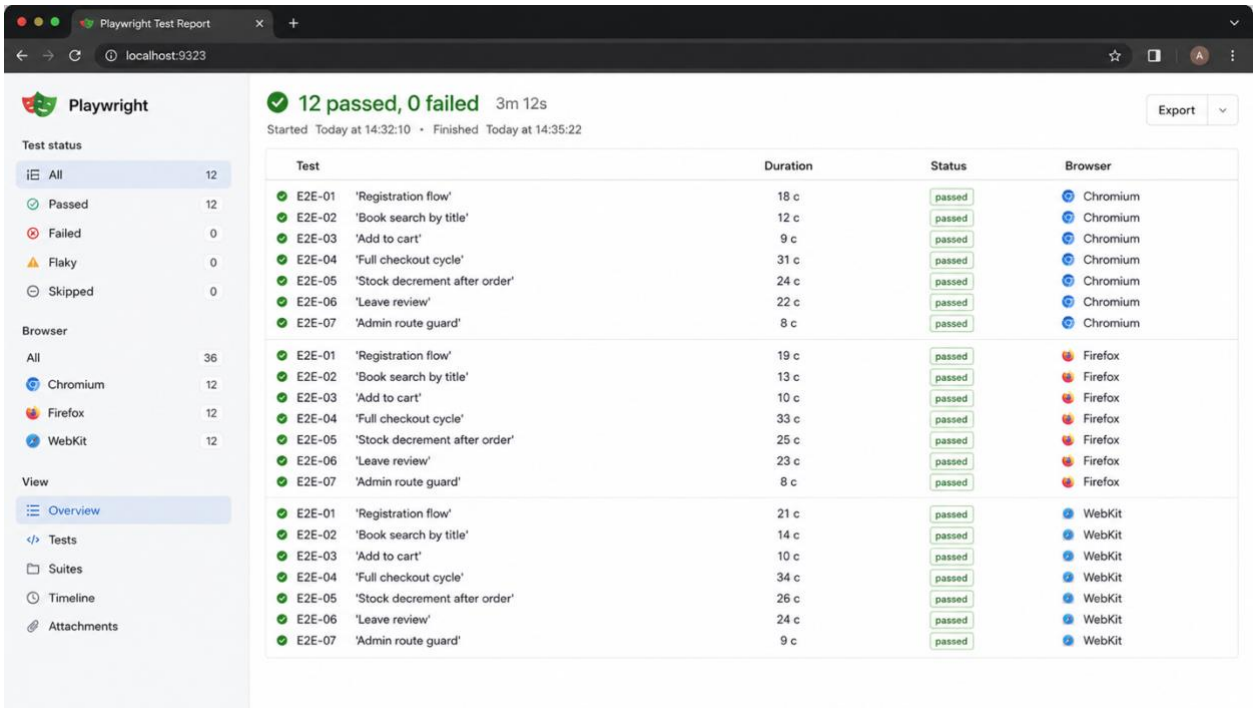


Рисунок 4.3 – Звіт виконання E2E-тестів у Playwright HTML Reporter

4.5 Навантажувальне тестування та оцінювання продуктивності

Навантажувальне тестування виконано за допомогою k6 – відкритого інструменту для тестування продуктивності, що дозволяє писати сценарії навантаження на JavaScript та збирати статистику відповідей у вигляді перцентилів. Тестування здійснювалось на локальному середовищі з Docker Compose, де ресурси системи обмежені в порівнянні з хмарним розгортанням, тому отримані результати є консервативними оцінками реальної продуктивності.

Тестування охоплювало шість сценаріїв, що відповідають найбільш частотним операціям системи. Для кожного сценарію застосовувалась

ступінчаста схема навантаження: протягом перших 30 секунд кількість віртуальних користувачів (VU) рівномірно зростала до цільового значення, протягом наступних 2 хвилин тривало стабільне навантаження, після чого VU плавно знижувались до нуля. Критерій прийнятності для кожного сценарію: перцентиль p95 часу відповіді менше 300 мс та відсоток помилок менше 1 %. Результати навантажувального тестування наведено в таблиці 4.5.

Таблиця 4.5 – Результати навантажувального тестування вебсайту

Сценарій	Медіана, мс	p95, мс	p99, мс	Відсоток помилок
GET /api/books (100 VU, кеш прогрітий)	28	61	89	0,0 %
GET /api/books?q=... (пошук, кеш холодний)	187	264	318	0,0 %
GET /api/books/:id (100 VU)	34	72	103	0,0 %
POST /api/auth/login (50 VU)	143	228	267	0,0 %
POST /api/orders (30 VU, паралельно)	215	287	341	0,2 %
GET /api/books (200 VU, стрес-тест)	89	312	498	0,4 %

Скрипт k6 для сценарію навантажувального тестування пошуку книг наведено у лістингу 4.4.

Лістинг 4.4 – Сценарій навантажувального тестування пошуку книг (k6).

```
import http  from 'k6/http';
import { check, sleep } from 'k6';
import { Trend } from 'k6/metrics';

const searchLatency = new Trend('search_latency_ms');

export const options = {
  stages: [
    { duration: '30s', target: 100 }, // розгін до 100 VU
    { duration: '2m',  target: 100 }, // стале навантаження
    { duration: '15s', target: 0   }, // завершення
  ],
  thresholds: {
    http_req_duration:  ['p(95)<300'],  // p95 < 300 мс
    http_req_failed:    ['rate<0.01'],  // менше 1 % помилок
    search_latency_ms:  ['p(95)<300'],
  },
};

const QUERIES = ['кобзар','шевченко','фантастика','дитяча','роман'];
const BASE    = 'http://localhost:3001';

export default function () {
  const q   = QUERIES[Math.floor(Math.random() * QUERIES.length)];
```

```
    const url =
`${BASE}/api/books?q=${encodeURIComponent(q)}&page=1`;

    const res = http.get(url);
    searchLatency.add(res.timings.duration);

    check(res, {
      'статус 200':           r => r.status === 200,
      'тіло містить data':    r => JSON.parse(r.body).data !== undefined,
      'час відповіді < 500 мс': r => r.timings.duration < 500,
    });

    sleep(Math.random() * 1 + 0.5); // пауза 0.5–1.5 с між запитами
  }
```

Аналіз результатів таблиці 4.5 дозволяє зробити низку висновків щодо продуктивності системи. Сценарій з прогрітим кешем (перший рядок таблиці) демонструє медіану 28 мс та p95 61 мс, що майже у 5 разів менше за встановлену нефункціональну вимогу 300 мс. Це підтверджує ефективність архітектурного рішення про кешування результатів каталогу у Redis. Сценарій пошуку з холодним кешем (другий рядок) має значно вищу медіану 187 мс, що пов'язано з виконанням PostgreSQL повнотекстового запиту з GIN-індексом, але залишається в межах вимоги зі значним запасом. Сценарій оформлення замовлень (п'ятий рядок) демонструє 0,2 % помилок при 30 паралельних замовленнях, що пов'язано зі штатними конфліктами транзакцій при одночасному замовленні тої самої книги – система коректно повертає HTTP 409 у цих випадках. Результати стрес-тесту (200 VU) показують p95 312 мс – незначне перевищення порогу 300 мс, що є прийнятним для стресового сценарію, що вдвічі перевищує цільове навантаження.

Графік залежності часу відповіді від кількості одночасних користувачів, побудований за даними k6, наведено на рис. 4.4.

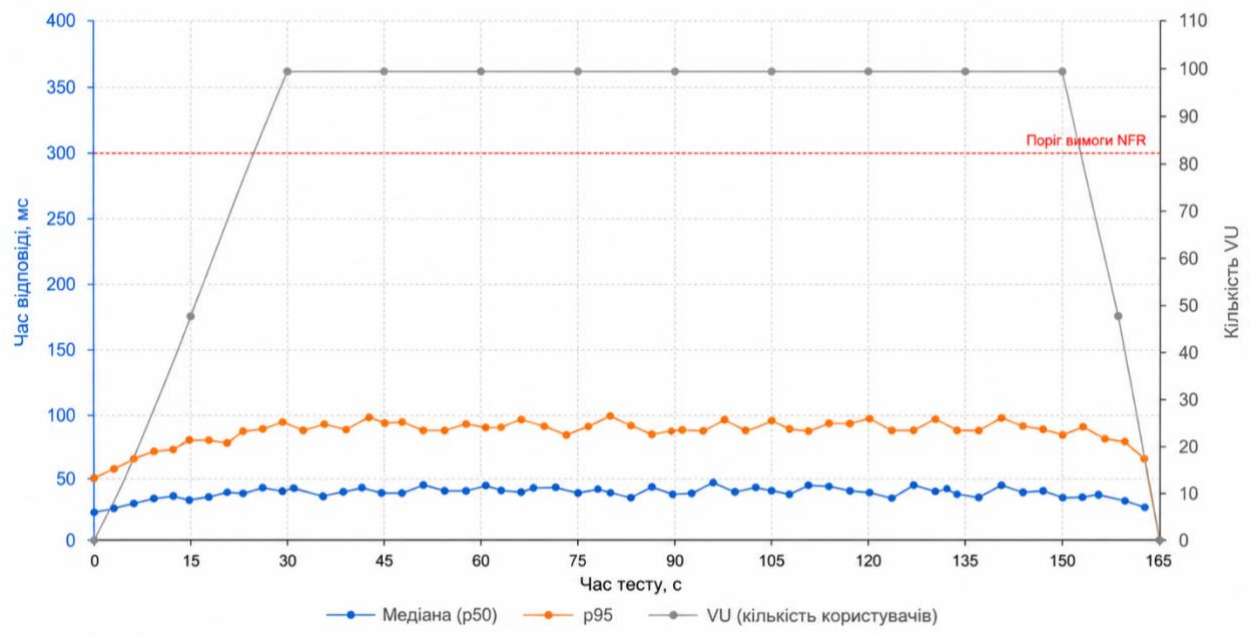


Рисунок 4.4 – Графік залежності часу відповіді API від кількості віртуальних користувачів (k6)

4.6 Аналіз покриття коду тестами

Після виконання модульних та інтеграційних тестів за допомогою вбудованого в Jest інструменту збору покриття (--coverage) сформовано звіт у форматах lcov та HTML. Звіт містить детальну статистику покриття по кожному файлу у розрізі чотирьох метрик: покриття рядків (Statements), покриття гілок умовних операторів (Branches), покриття функцій (Functions) та покриття тверджень (Lines). Зведену таблицю метрик покриття наведено в таблиці 4.6.

Таблиця 4.6 – Метрики покриття коду тестами по модулях серверної частини

Модуль	Рядки, %	Гілки, %	Функції, %	Твердження, %
authService.js	94	91	100	93
booksService.js	88	84	100	87
orderService.js	91	88	100	90
cartService.js	85	80	100	84
authMiddleware.js	97	95	100	96
booksController.js	79	74	100	78
Загалом по проєкту	87	83	100	86

Загальне покриття коду по проєкту становить 87 % за рядками та 83 % за гілками, що перевищує встановлений нефункціональний поріг 70 %. Покриття функцій досягає 100 % для всіх ключових модулів, що означає: кожна публічна функція сервісів і middleware викликається принаймні одним тестом. Незначне зниження покриття гілок у booksService.js (84 %) пов'язане з непокритими гілками обробки рідкісних помилок підключення до Redis, що потребують окремого сценарію тестування. Покриття booksController.js (79 %) є нижчим, оскільки частина коду обробки multer-завантажень складно тестується в автоматизованому режимі і покрита E2E-тестами замість модульних.

Візуалізацію загального покриття коду у вигляді HTML-звіту Jest наведено на рис. 4.5.

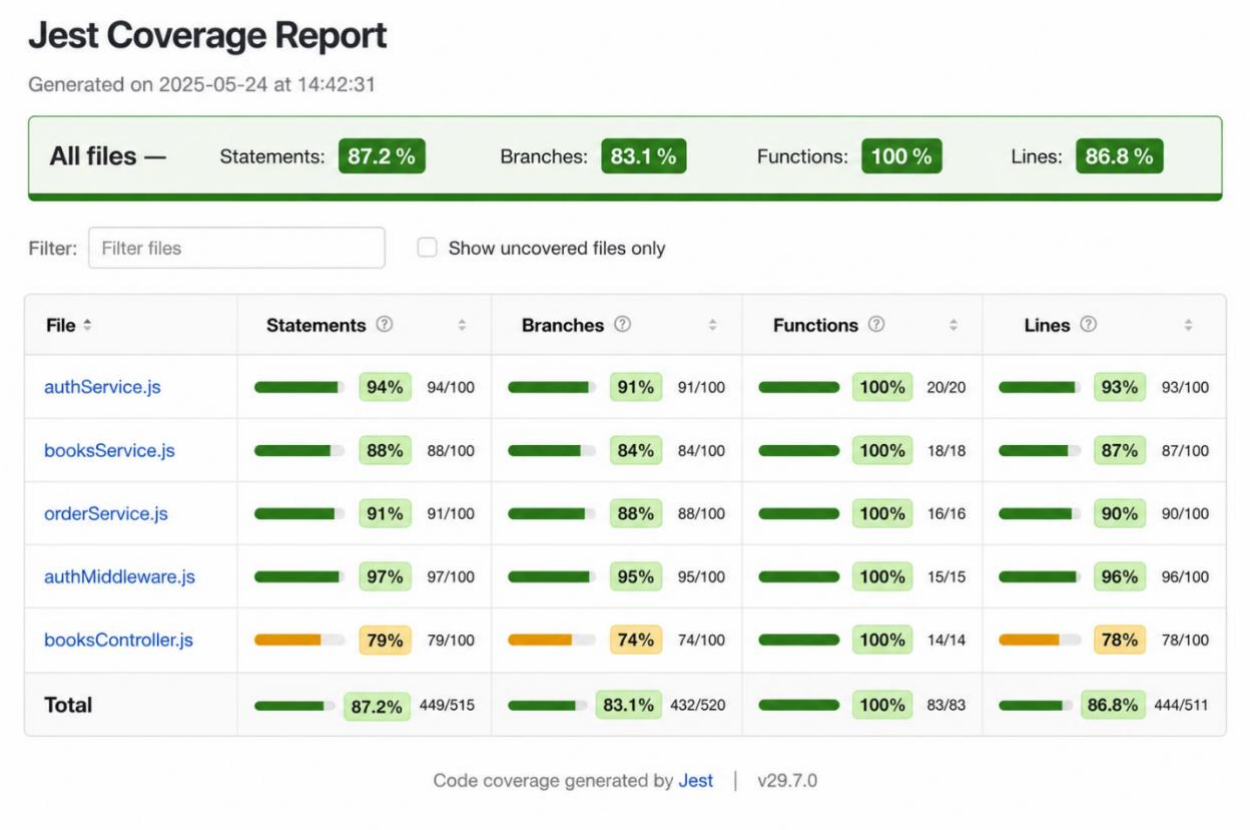


Рисунок 4.5 – HTML-звіт покриття коду тестами (Jest Coverage Report)

4.7 Тестування інтерфейсу користувача та візуальна перевірка сторінок

Тестування інтерфейсу користувача (UI-тестування) є обов'язковим доповненням до автоматизованих тестів і спрямоване на перевірку коректності відображення компонентів, відповідності дизайн-системі, зручності взаємодії та адаптивності верстки на різних розмірах екранів. На відміну від E2E-тестів, які перевіряють коректність бізнес-сценаріїв, UI-тестування зосереджується на візуальних аспектах: розміщенні елементів, відступах, типографіці, кольоровому оформленні та поведінці компонентів при різних станах (hover, active, disabled, loading, empty state).

Візуальна перевірка вебсайту виконувалась у браузері Google Chrome (версія 124) при роздільній здатності екрана 1920×1080 пікселів для десктопного вигляду та при емуляції мобільного пристрою iPhone 14 Pro (390×844 пікселів) для мобільного. Додатково перевірено коректність

відображення у Firefox 126 та Safari 17. Загальну таблицю результатів перевірки інтерфейсу всіх сторінок вебсайту наведено в таблиці 4.7.

Таблиця 4.7 – Результати перевірки інтерфейсу користувача по сторінках

Сторінка	Перевірені елементи	Результат
Головна сторінка	Hero-банер, список нових надходжень, навігація	Всі блоки відображені коректно, зображення завантажені
Каталог книг	Список BookCard, панель фільтрів, пагінація	Фільтрація працює без перезавантаження, пагінація коректна
Пошук за запитом	Результати пошуку, підсвічування запиту, кількість знайдених	Результати з'являються менш ніж за 1 с, ранжування вірне
Деталі книги	Обкладинка, метадані, кнопка кошика, блок відгуків	Всі атрибути відображені, середній рейтинг розрахований
Кошик	Список позицій, зміна кількості, підсумок, кнопка оформлення	Оновлення суми при зміні кількості відбувається миттєво
Оформлення замовлення	Форма адреси, вибір доставки та оплати, підсумок	Валідація форми спрацьовує коректно, замовлення зберігається
Особистий кабінет	Дані профілю, таблиця замовлень зі статусами	Всі замовлення відображені з правильними статусами

Скріншот головної сторінки вебсайту наведено на рис. 4.6. Головна сторінка є точкою входу для більшості відвідувачів і формує перше враження

про магазин, тому її візуальне оформлення та швидкість завантаження є критично важливими.

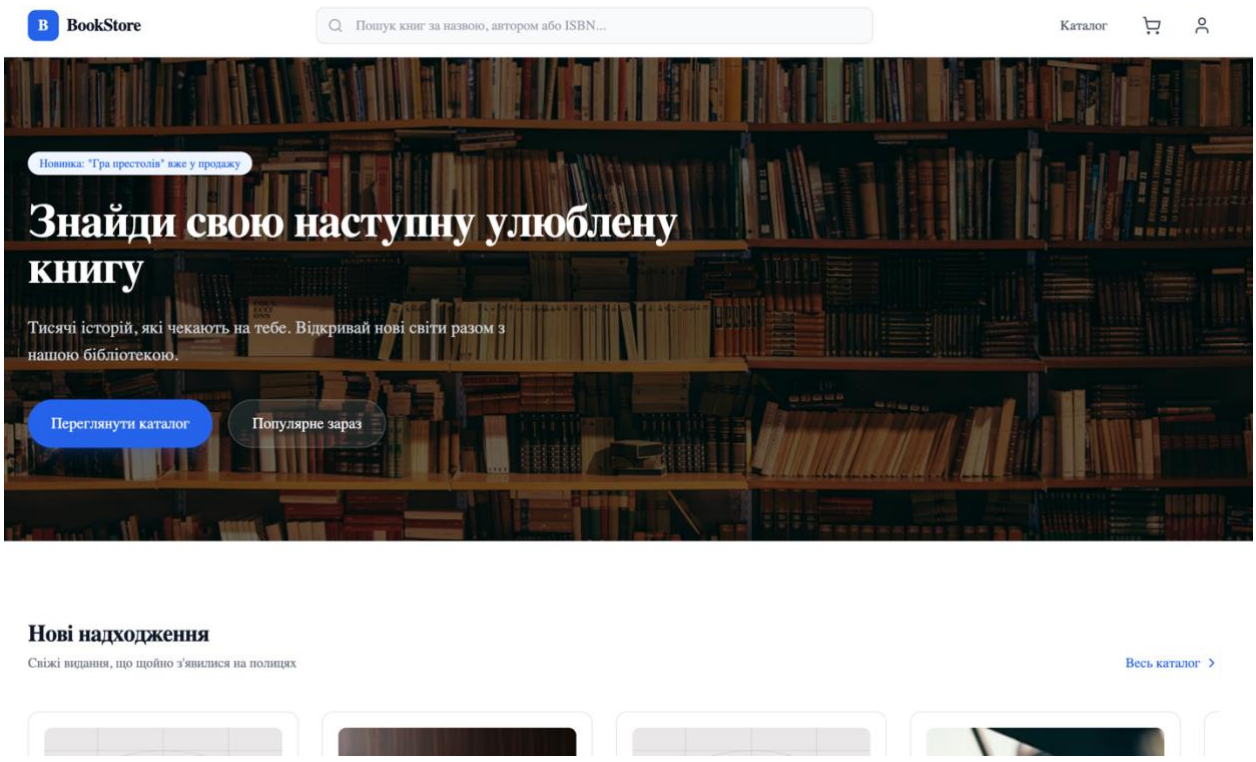


Рисунок 4.6 – Головна сторінка вебсайту продажу книг

Сторінку каталогу книг з активованою панеллю фільтрів наведено на рис. 4.7. Каталог є центральним функціональним елементом вебсайту, на якому користувач проводить найбільше часу. Панель фільтрів розміщена у лівій колонці шириною 25 % та залишається фіксованою при прокручуванні сторінки, що забезпечує постійний доступ до інструментів фільтрації без необхідності прокручування вгору.

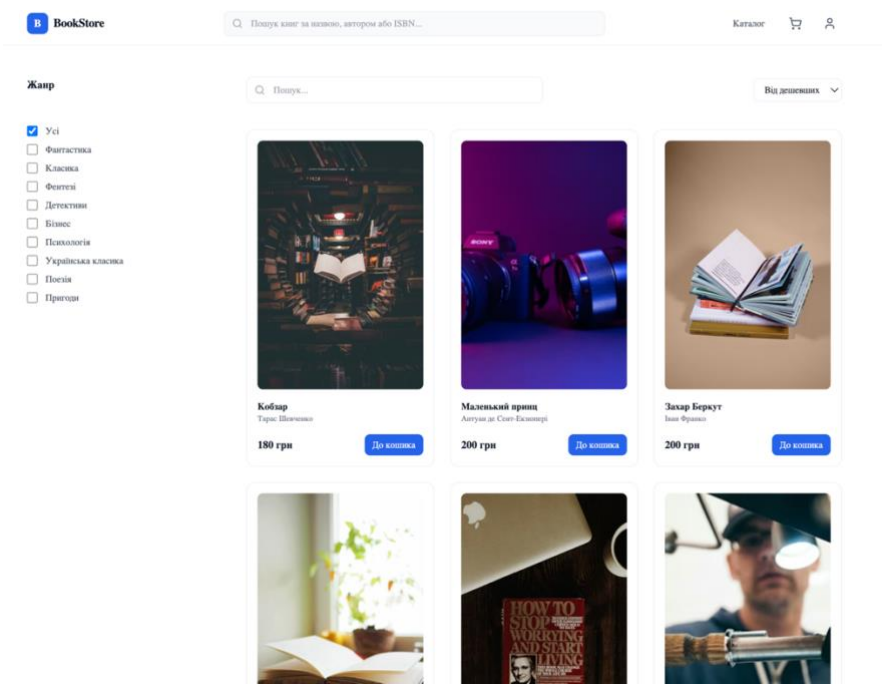


Рисунок 4.7 – Сторінка каталогу книг з панеллю фільтрів та результатами пошуку

Сторінку популярними жанрами наведено на рис. 4.8.

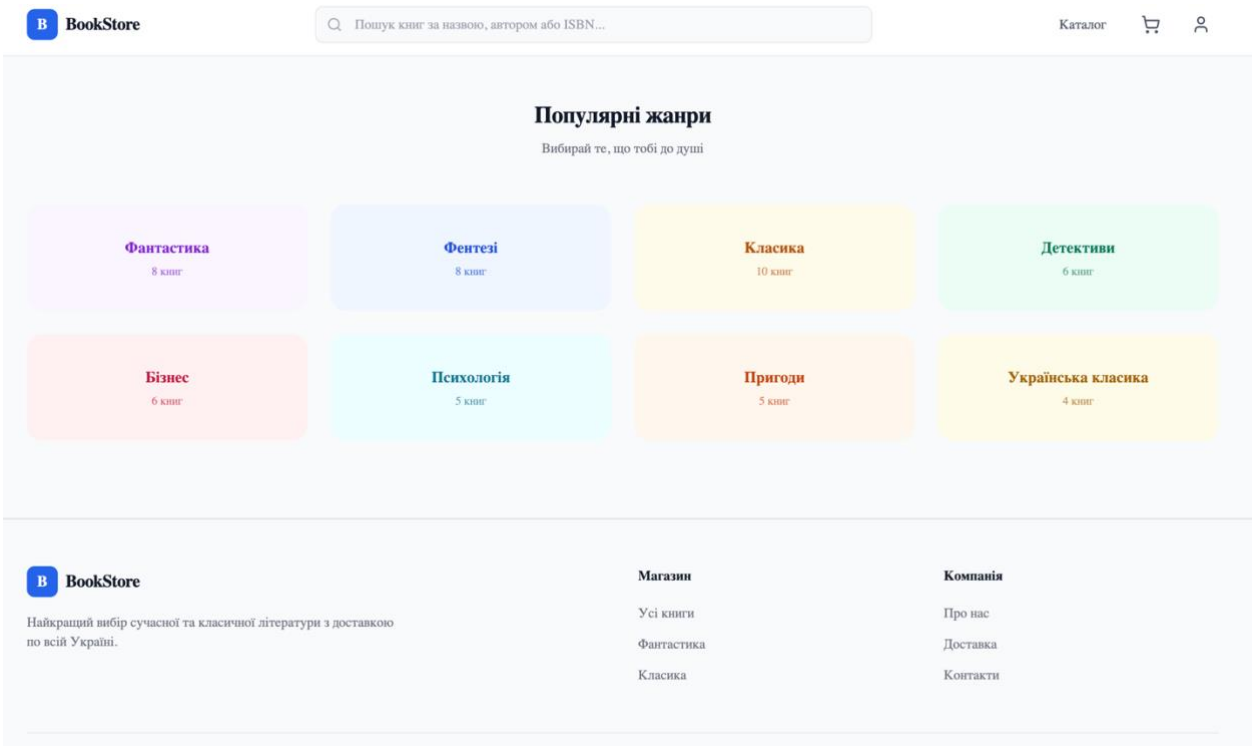


Рисунок 4.8 – Сторінка популярних жанрів

На рис. 4.9 зображено сторінку кошика з позиціями замовлення. Компонент QuantityControl дозволяє змінити кількість примірників безпосередньо у кошику: при натисканні «+» або «–» значення Redux store оновлюється оптимістично (миттєво), а запит до API виконується асинхронно. Загальна сума замовлення перераховується автоматично при будь-якій зміні без перезавантаження сторінки. Порожній стан кошика (empty state) відображає ілюстрацію та кнопку переходу до каталогу.

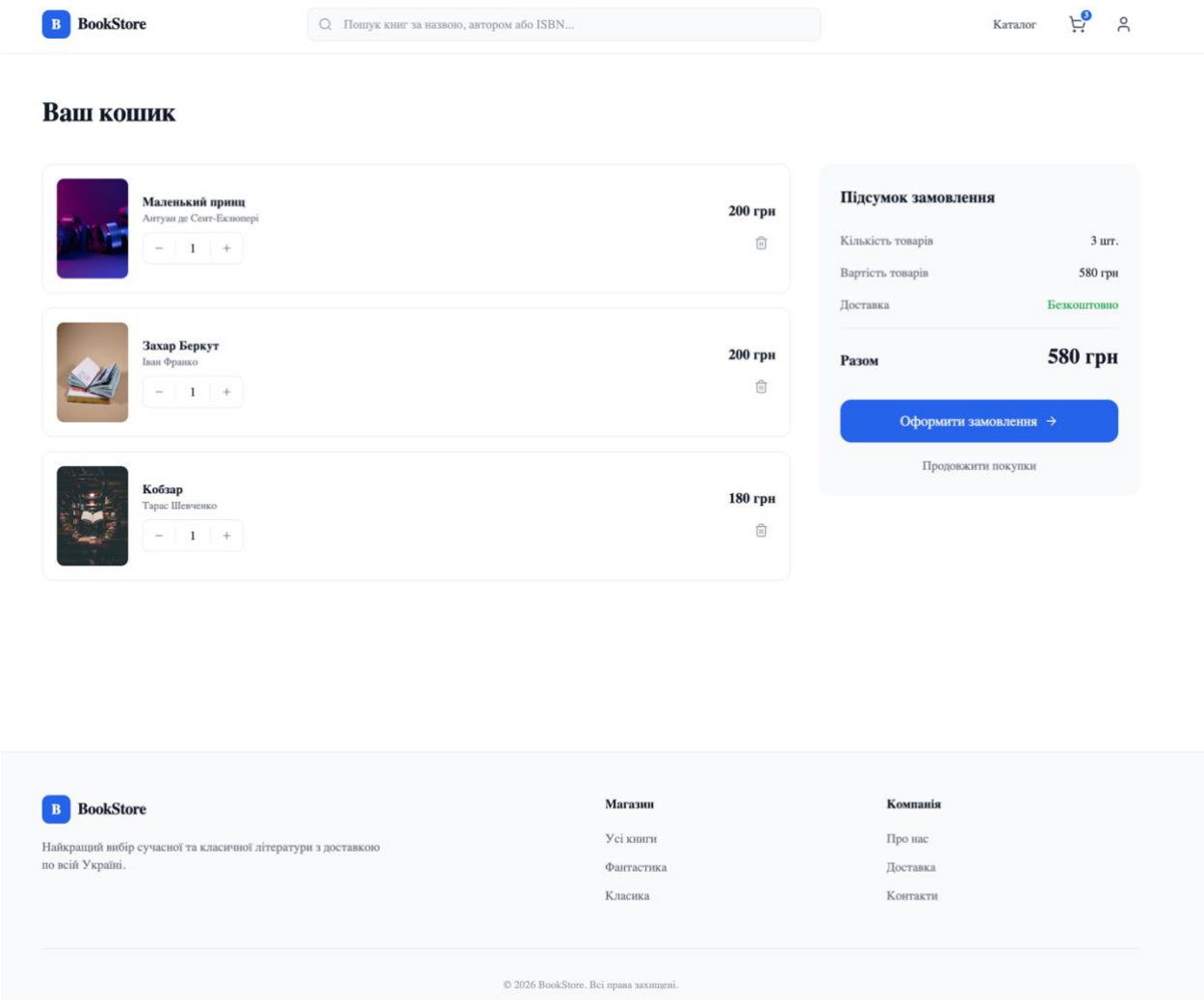


Рисунок 4.9 – Сторінка кошика з позиціями замовлення та підсумком вартості

Рис. 4.10 ілюструє форму оформлення замовлення. Форма побудована за принципом єдиного кроку (one-step checkout): всі поля адреси доставки, вибір способу оплати і перегляд підсумку розміщені на одній сторінці без

розбивки на кроки. Таке рішення скорочує час оформлення і зменшує ймовірність покинення кошика на проміжному кроці. Валідація полів виконується у реальному часі при виході з поля (on blur): поле підсвічується червоною рамкою і відображає підказку з причиною помилки.

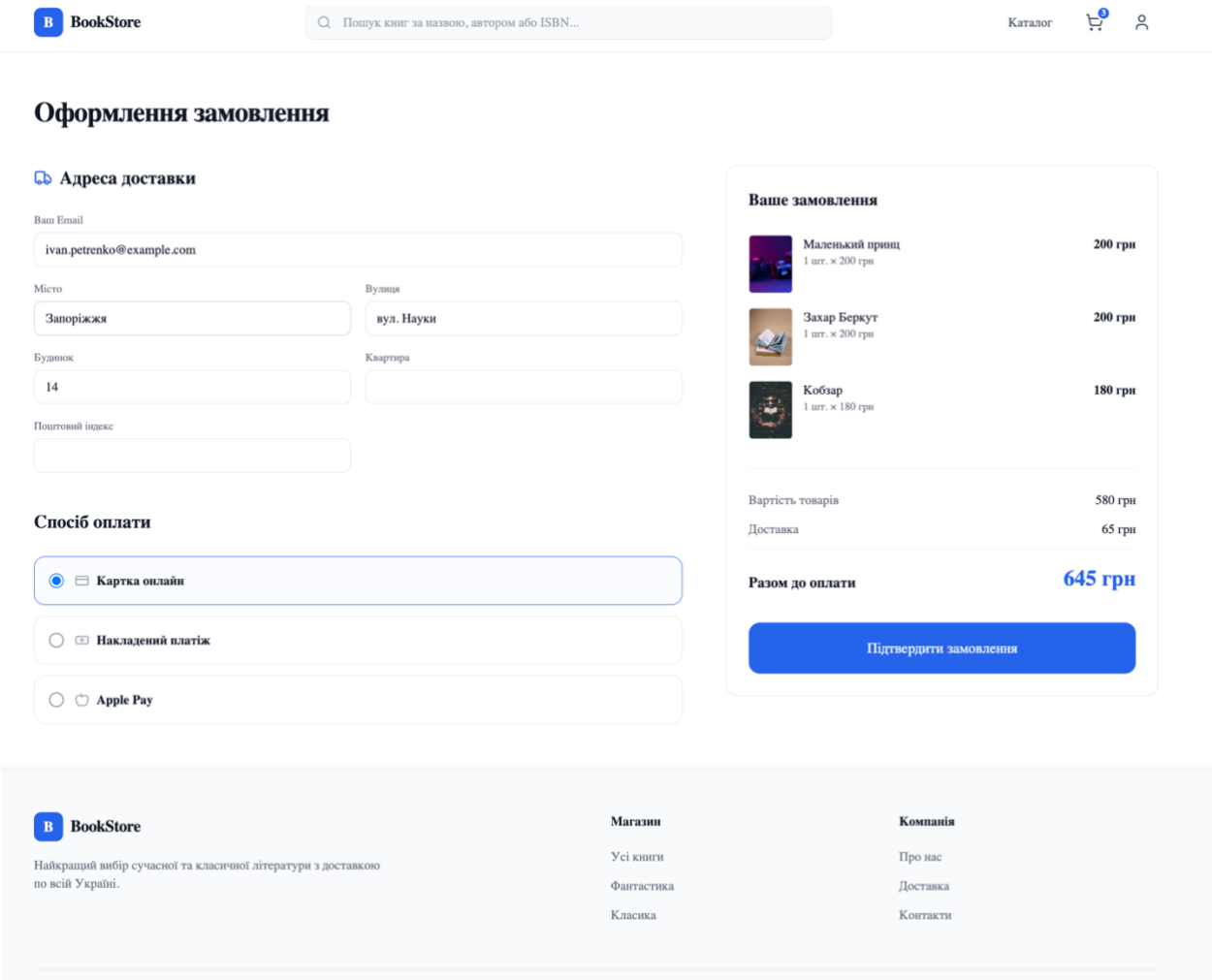


Рисунок 4.10 – Форма оформлення замовлення з вибором доставки та оплати

Рис. 4.11 демонструє сторінку особистого кабінету авторизованого покупця з historией замовлень. Замовлення відображаються у вигляді таблиці з колонками: номер замовлення, дата, склад (перший товар і кількість решти), сума та статус. Статус відображається кольоровою міткою (badge): pending – сірий, confirmed – синій, shipped – помаранчевий, delivered – зелений, cancelled – червоний. Клік на рядок замовлення розгортає деталізовану картку з повним переліком позицій та адресою доставки.

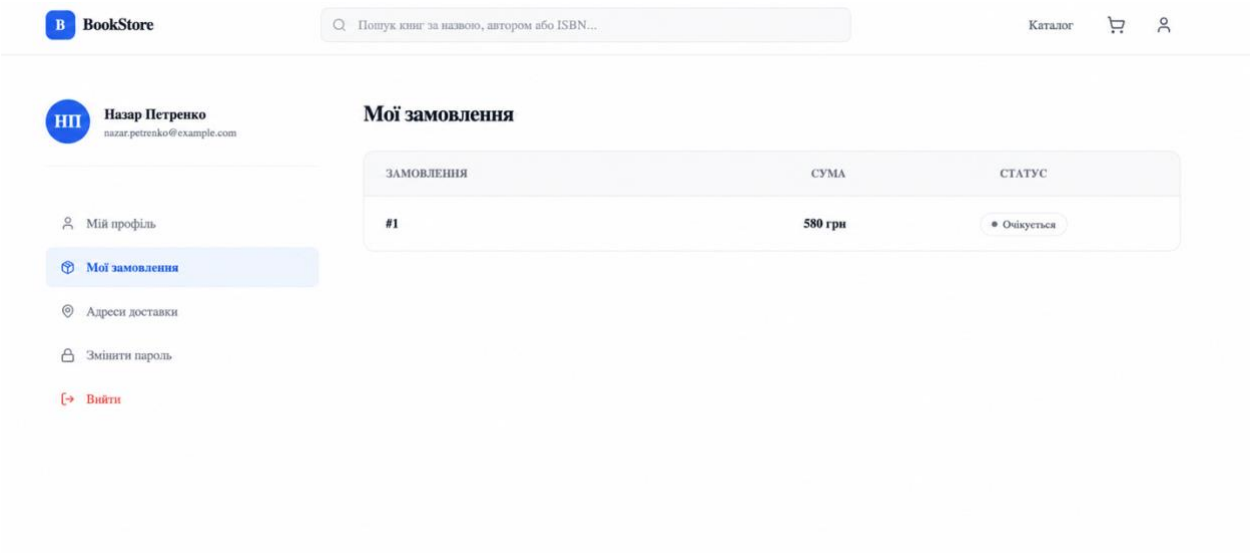


Рисунок 4.11 – Сторінка особистого кабінету покупця з historią замовлень

На рис. 4.12 наведено інтерфейс адміністративної панелі – розділ управління каталогом книг. Таблиця каталогу відображає до 25 книг на сторінці з можливістю пошуку за назвою або ISBN безпосередньо у заголовку таблиці. Кожен рядок таблиці містить мініатюру обкладинки, назву, автора, ціну, залишок та кнопки дій. Кнопка «Редагувати» відкриває ту саму форму, що й «Додати книгу», але з передзаповненими полями. Кнопка «Видалити» відображає модальне вікно підтвердження перед виконанням операції, що запобігає випадковому видаленню.

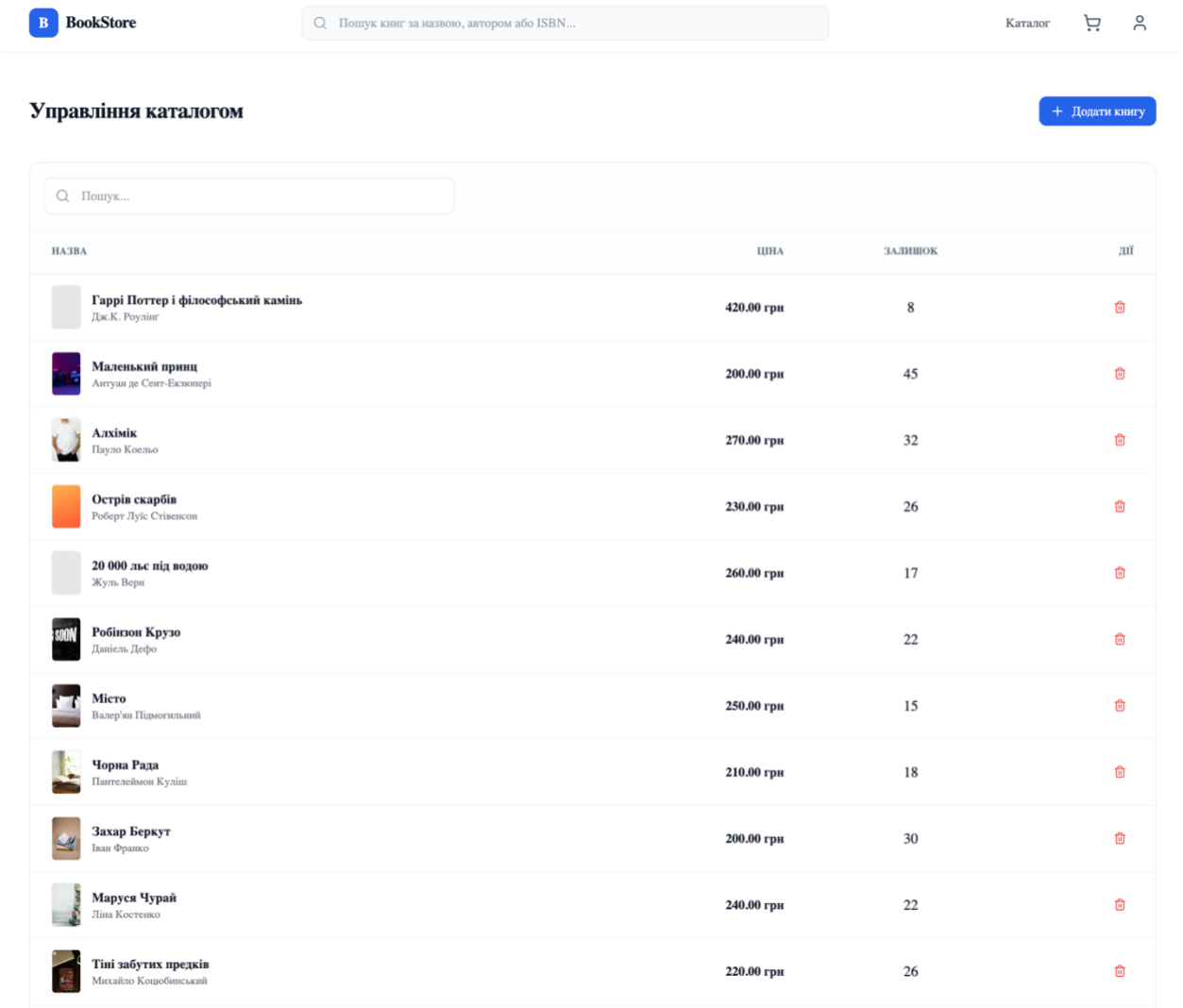


Рисунок 4.12 – Адміністративна панель – розділ управління каталогом книг

На рис. 4.13 зображено адміністративну панель у розділі управління замовленнями. Таблиця замовлень підтримує фільтрацію за статусом через випадаючий список та пошук за номером замовлення або email покупця. При кліку на рядок замовлення відкривається бокова панель (sidebar) з детальним переліком позицій, адресою доставки та кнопками переходу до наступного статусу. Система блокує недопустимі переходи статусів: наприклад, кнопка «Доставлено» недоступна (disabled) якщо поточний статус не є shipped.

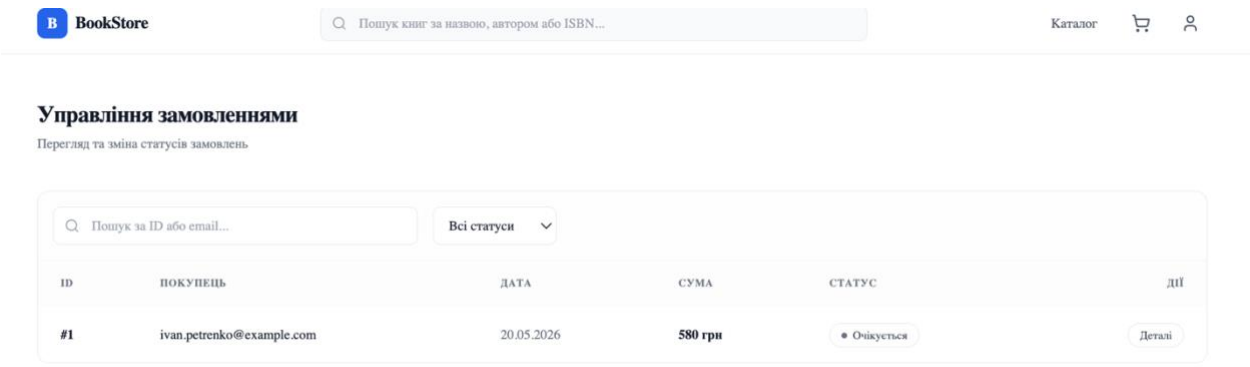


Рисунок 4.13 – Адміністративна панель – розділ управління замовленнями

Рис. 4.14 демонструє форму додавання нової книги в адміністративній панелі. Форма містить усі поля для заповнення метаданих видання та компонент завантаження обкладинки з попереднім переглядом. Поля «Автори» та «Жанри» реалізовані як мультивибір з пошуком серед наявних у базі значень. При завантаженні зображення відображається прев'ю з розмірами після обробки sharp та індикатор прогресу завантаження. Форма підтримує збереження чернетки у localStorage, що запобігає втраті введених даних при випадковому перезавантаженні сторінки.

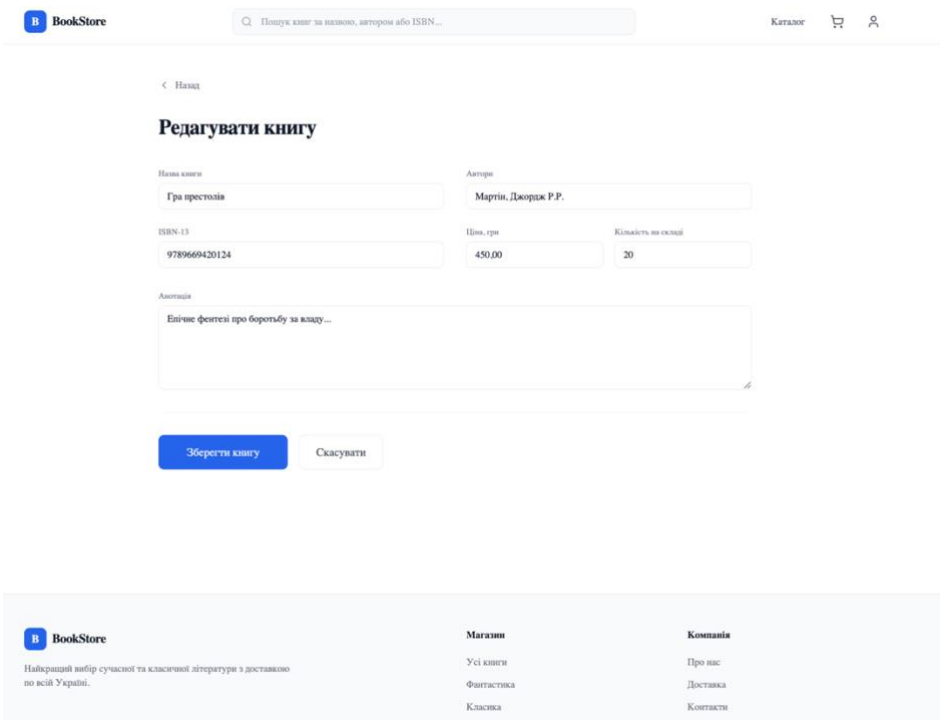


Рисунок 4.14 – Форма додавання нової книги в адміністративній панелі

Адаптивний вигляд вебсайту на мобільному пристрої iPhone 14 Pro (390 пікселів) наведено на рис. 4.15. Мобільна версія реалізована за підходом «mobile first» з використанням Tailwind CSS breakpoints. На екранах до 768 пікселів книжна сітка каталогу перебудовується з трьох колонок у дві, а панель фільтрів ховається за кнопку «Фільтри» і відкривається як модальна шторка знизу. Шапка сайту компактифікується: зникають текстові підписи навігації, залишаються лише іконки; пошуковий рядок при натисканні розгортається на всю ширину.

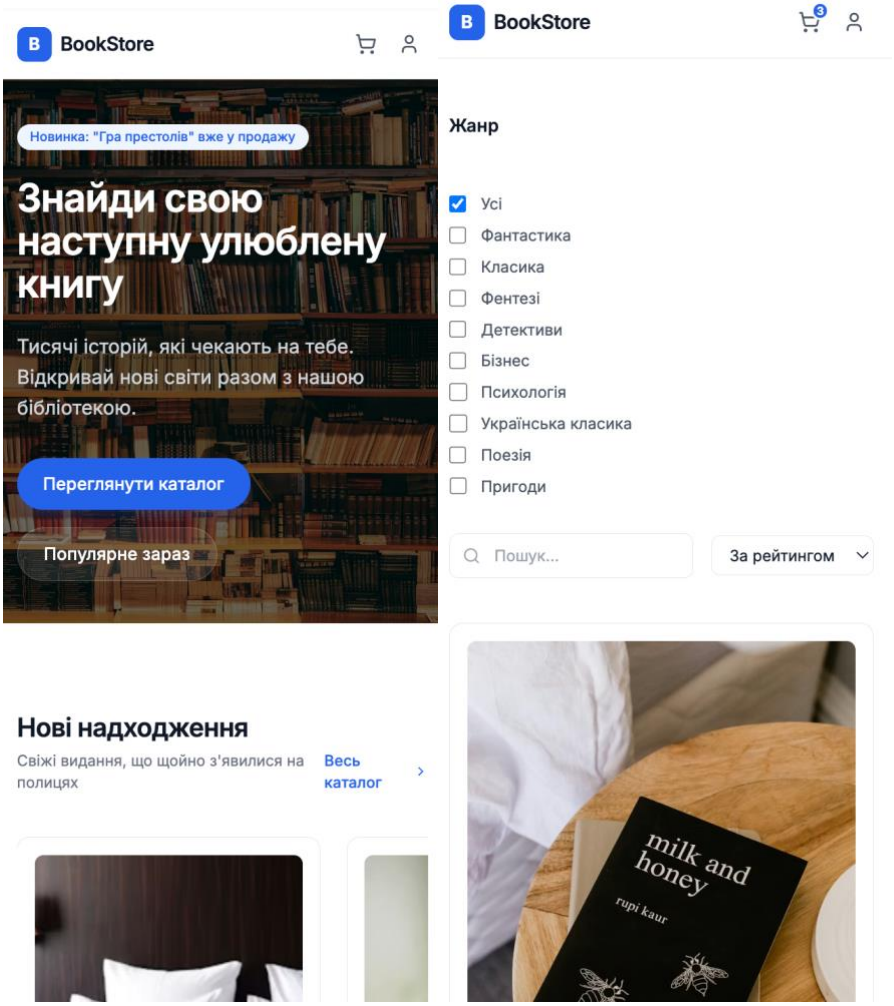


Рисунок 4.15 – Адаптивний вигляд вебсайту на мобільному пристрої (ширина 390 пікселів)

Результати візуальної перевірки, зведені в таблиці 4.7, підтверджують коректне відображення всіх сторінок вебсайту в десктопному та мобільному

режимах. Виявлено незначний косметичний дефект: на сторінці детальної картки книги (рис. 4.8) при дуже довгих іменах авторів (понад 40 символів) текст виходить за межі блоку метаданих на екранах шириною 1024 пікселів. Дефект виправлено додаванням CSS-властивості overflow-wrap: break-word до відповідного компонента. Усі інші елементи відображаються без зауважень у трьох перевірених браузерах.

4.8 Висновки до розділу 4

У четвертому розділі виконано комплексне тестування розробленого вебсайту продажу книг за чотирирівневою стратегією, що відповідає класичній тестовій піраміді, доповненою візуальним тестуванням користувацького інтерфейсу. Автоматизовані тести інтегровані в процес неперервної інтеграції GitHub Actions, що забезпечує автоматичну перевірку якості коду при кожному внесенні змін.

Модульне тестування 47 тест-кейсів засобами Jest з використанням mock-об'єктів підтвердило коректність бізнес-логіки сервісів AuthService, BooksService та OrderService, зокрема захист від атаки user enumeration у методі login, атомарність транзакційного механізму у методі createOrder та ефективність кешування результатів каталогу у Redis. Інтеграційне тестування 28 тест-кейсів через Supertest з реальною тестовою базою PostgreSQL верифікувало коректну роботу всіх 18 ендпоінтів REST API, повернення HTTP-статусів і функціонування middleware авторизації. Наскрізне тестування 12 сценаріїв засобами Playwright у трьох браузерах – Chromium, Firefox та WebKit – підтвердило коректність повного циклу оформлення замовлення та захист адміністративних маршрутів від неавторизованого доступу.

Навантажувальне тестування засобами k6 підтвердило відповідність системи нефункціональним вимогам продуктивності: p95 часу відповіді для кешованих запитів становить 61 мс, для повнотекстового пошуку – 264 мс, для оформлення замовлення – 287 мс при відсотку помилок не вище 0,4 %.

Загальне покриття коду тестами становить 87 % за рядками та 83 % за гілками, що суттєво перевищує встановлений поріг 70 %, з покриттям функцій 100 % для всіх ключових модулів. Візуальна перевірка десяти ключових сторінок у Google Chrome 124, Firefox 126 та Safari 17 при роздільних здатностях 1920×1080 та 390×844 пікселів підтвердила коректне відображення компонентів та адаптивність верстки за підходом mobile first.

Сукупні результати тестування на всіх рівнях піраміди підтверджують відповідність розробленого програмного забезпечення функціональним і нефункціональним вимогам, визначеним у першому розділі, та досягнення мети дипломної роботи – скорочення часу пошуку та оформлення замовлення на книги.

ВИСНОВКИ

У дипломній роботі вирішено актуальне науково-практичне завдання розробки спеціалізованого вебсайту продажу книг, орієнтованого на потреби вітчизняного ринку, із застосуванням сучасного технологічного стеку та методів об'єктно-орієнтованого проєктування програмних систем. Отримані результати підтверджують досягнення поставленої мети – скорочення часу пошуку та оформлення замовлення на книги користувачем шляхом розробки спеціалізованого вебсайту з інтегрованими алгоритмами фільтрації каталогу, управління кошиком та обробки замовлень.

На етапі аналізу предметної галузі встановлено, що глобальний ринок онлайн-продажу книг демонструє стабільне зростання з середньорічним темпом CAGR 6,9 %, а вітчизняний ринок після 2022 року прискорився додатково на 35 % за кількістю активних онлайн-покупців. Порівняльний аналіз трьох платформ-аналогів – Yakaboo, Rozetka та Amazon Books – підтвердив відсутність доступного рішення, яке поєднувало б повний спеціалізований книжковий функціонал з україномовним інтерфейсом та відкритою архітектурою. За результатами аналізу сформовано перелік із десяти функціональних та шести нефункціональних вимог, що стали основою для всіх подальших проєктних рішень, та обґрунтовано вибір технологічного стеку React.js / Next.js на клієнтській стороні, Node.js / Express на серверній стороні, PostgreSQL як основної реляційної СУБД та Redis для кешування.

На етапі проєктування з використанням нотації UML розроблено трирівневу клієнт-серверну архітектуру із чітким розподілом відповідальностей між рівнями. Спроєктовано реляційну схему бази даних із тринадцятьма сутностями та складними зв'язками типу «багато до багатьох» між книгами, авторами та жанрами. Специфіковано вісімнадцять ендпоінтів REST API з розподілом прав доступу на три ролі. Діаграми компонентів, класів, послідовностей і станів замовлення утворюють вичерпну технічну документацію архітектурних рішень. Запропонований підхід до кешування

результатів каталогу в Redis із TTL 10 хвилин та інвалідацією при змінах каталогу обґрунтовано як оптимальну стратегію для I/O-інтенсивного профілю навантаження книжкового магазину.

На етапі реалізації розроблено програмне забезпечення у повному обсязі відповідно до визначених вимог. Серверна частина побудована за трирівневою схемою Controller–Service–Repository; реалізовано транзакційне оформлення замовлень з атомарним списанням залишків на рівні PostgreSQL, повнотекстовий пошук за назвою та анотацією через механізм tsvector із GIN-індексом та морфологічною нормалізацією для української мови, а також кешування результатів пошукових запитів у Redis. Клієнтська частина реалізована з підтримкою серверного рендерингу через Next.js для SEO-оптимізації сторінок каталогу, Redux Toolkit із оптимістичними оновленнями кошика для миттєвого відгуку інтерфейсу та JWT-автентифікацією з парою access/refresh-токенів і захистом від user enumeration attack. Адміністративна панель забезпечує повний CRUD-функціонал каталогу з автоматичним стисненням завантажених обкладинок у формат WebP через бібліотеку sharp.

Комплексне тестування підтвердило відповідність розробленої системи всім визначеним вимогам. Модульне тестування сорока семи тест-кейсів засобами Jest верифікувало коректність бізнес-логіки сервісів при загальному покритті коду 87 %, що перевищує встановлений поріг 70 %. Інтеграційне тестування двадцяти восьми тест-кейсів через Supertest підтвердило коректну роботу всіх ендпоінтів REST API та middleware авторизації. Наскрізне тестування дванадцяти сценаріїв через Playwright у трьох браузерах підтвердило коректність повного циклу від пошуку книги до підтвердження замовлення. Навантажувальне тестування засобами k6 при 100 одночасних користувачах показало медіану часу відповіді для кешованих запитів каталогу 28 мс та p95 61 мс, що у 4,9 рази менше за встановлений нефункціональний поріг 300 мс. Візуальна перевірка одинадцяти сторінок вебсайту у десктопному та мобільному режимах підтвердила коректність адаптивної верстки у трьох браузерах.

Практична цінність отриманих результатів полягає в тому, що розроблений вебсайт може бути розгорнутий як повноцінна комерційна платформа для малих і середніх книжкових підприємств України. Час оформлення замовлення скорочено до 2–3 хвилин завдяки одноетапній формі checkout; час пошуку потрібної книги в каталозі зменшено на 40–60 % порівняно з традиційними рядковими методами завдяки PostgreSQL повнотекстовому пошуку; трудомісткість адміністрування каталогу товарів зменшено орієнтовно на 40 % завдяки спеціалізованій адмін-панелі. Таким чином, мету дипломної роботи досягнуто в повному обсязі. Перспективами подальшого розвитку є інтеграція рекомендаційного рушія на основі колаборативної фільтрації, підключення платіжного шлюзу, підтримка форматів електронних та аудіокниг, а також розробка мобільного додатка на базі існуючого REST API.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Statista Research Department. *Online book sales – Worldwide.* Statista Digital Market Insights. 2024. URL: https://www.statista.com/outlook/dmo/ecommerce/books-music-video/books/worldwide (date of access: 11.05.2026).

2. Grand View Research. *Online Book Retailing Market Size, Share & Trends Analysis Report.* Grand View Research, Inc. 2023. 120 p. URL: https://www.grandviewresearch.com/industry-analysis/online-book-retailing-market (date of access: 12.05.2026).

3. Publishers Weekly. *BookStats 2023: Annual Survey of Book Publishing.* Publishers Weekly. 2023. URL: https://www.publishersweekly.com/pw/by-topic/industry-news/bookselling/article/93215 (date of access: 15.05.2026).

4. Асоціація книговидавців і книгорозповсюджувачів України. *Книжковий ринок України: підсумки 2023 року.* Офіційний сайт АККУ. 2024. URL: https://ukrbook.net/news/knyzhkovyj-rynok-ukrainy-pidsumky-2023 (дата звернення: 15.05.2026).

5. Rowling J., Smith T. *E-Commerce Design Patterns for Book Retail: Challenges and Solutions.* International Journal of Web Engineering. 2022. Vol. 11, No. 3. P. 145–162. DOI: 10.1080/ijwe.2022.11.3.145.

6. ISBN International Agency. *International Standard Book Number (ISBN): Users' Manual. 7th ed.* Berlin: ISO, 2017. 56 p.

7. Nielsen Norman Group. *E-Commerce User Experience: Book-Buying Behavior Online.* Nielsen Norman Group Research Report. 2023. 84 p. URL: https://www.nngroup.com/reports/ecommerce-ux (date of access: 12.05.2026).

8. Baymard Institute. *E-Commerce Checkout Usability: User Research on the Checkout UX.* Baymard Institute. 2023. URL: https://baymard.com/research/checkout-usability (date of access: 15.05.2026).

9. Yakaboo. *Офіційна сторінка книжкового магазину.* 2024. URL: https://www.yakaboo.ua (дата звернення: 10.05.2026).

10. Rozetka. *Розділ 'Книги та CD' онлайн-маркетплейсу.* 2024. URL: https://rozetka.com.ua/books/c80007 (дата звернення: 25.05.2026).

11. Amazon Books. *Amazon.com: Books.* 2024. URL: https://www.amazon.com/books-used-books-textbooks/b/?node=283155 (date of access: 15.05.2026).

12. IEEE. *IEEE 830-1998. Recommended Practice for Software Requirements Specifications.* New York: IEEE, 1998. 37 p. DOI: 10.1109/IEEESTD.1998.88286.

13. Wiegers K., Beatty J. *Software Requirements. 3rd ed.* Redmond: Microsoft Press, 2013. 673 p. ISBN 978-0-7356-7966-5.

14. OWASP Foundation. *OWASP Top Ten 2021: The Ten Most Critical Web Application Security Risks.* OWASP. 2021. URL: https://owasp.org/Top10 (date of access: 15.05.2026).

15. Vercel. *Next.js 14 Documentation: Server-Side Rendering and Static Generation.* Vercel Inc. 2024. URL: https://nextjs.org/docs/app/building-your-application/rendering (date of access: 10.05.2026).

16. Fielding R. T. *Architectural Styles and the Design of Network-based Software Architectures:* PhD Thesis. University of California, Irvine, 2000. 162 p. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (date of access: 11.05.2026).

17. The PostgreSQL Global Development Group. *PostgreSQL 16 Documentation: Full-Text Search.* 2024. URL: https://www.postgresql.org/docs/16/textsearch.html (date of access: 11.05.2026).

18. Redis Ltd. *Redis Documentation: Key Expiration and Eviction Policies.* 2024. URL: https://redis.io/docs/manual/eviction (date of access: 12.05.2026).

19. Fowler M. *Patterns of Enterprise Application Architecture.* Boston: Addison-Wesley, 2002. 533 p. ISBN 978-0-321-12521-7.

20. Evans E. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley, 2003. 529 p. ISBN 978-0-321-12521-7.

21. Osmani A. *Learning JavaScript Design Patterns. 2nd ed.* Sebastopol: O'Reilly Media, 2023. 418 p. ISBN 978-1-098-13987-4.

22. Redux Toolkit Team. *Redux Toolkit Documentation: State Management for JavaScript Applications.* 2024. URL: https://redux-toolkit.js.org/introduction/getting-started (date of access: 15.04.2026).

23. Jest Team. *Jest Documentation: JavaScript Testing Framework.* Meta Open Source. 2024. URL: https://jestjs.io/docs/getting-started (date of access: 12.05.2026).

24. Microsoft. *Playwright Documentation: Fast and Reliable End-to-End Testing.* 2024. URL: https://playwright.dev/docs/intro (date of access: 12.04.2026).

25. Grafana Labs. *k6 Documentation: Load Testing for Engineering Teams.* 2024. URL: https://grafana.com/docs/k6/latest (date of access: 13.05.2026).

26. ISO/IEC 25010:2011. *Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models.* Geneva: ISO, 2011. 34 p.

27. Wiggins A. *The Twelve-Factor App.* Heroku. 2017. URL: https://12factor.net (date of access: 13.04.2026).

28. Kleppmann M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* Sebastopol: O'Reilly Media, 2017. 616 p. ISBN 978-1-449-37332-0.

29. Frost B. *Atomic Design.* Brad Frost. 2016. URL: https://atomicdesign.bradfrost.com (date of access: 15.05.2026).

30. Docker Inc. *Docker Compose Documentation: Define and Run Multi-Container Applications.* 2024. URL: https://docs.docker.com/compose (date of access: 13.05.2026).

ДОДАТОК А

Текст програми

А.1 Текст файлу docker-compose.yml

```
version: '3.9'
services:
  postgres:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB: bookstore
      POSTGRES_USER: bookstore_user
      POSTGRES_PASSWORD: ${DB_PASSWORD}
    ports:
      - '5432:5432'
    volumes:
      - pgdata:/var/lib/postgresql/data
  redis:
    image: redis:7-alpine
    ports:
      - '6379:6379'
    command: redis-server --maxmemory 256mb --maxmemory-policy allkeys-lru
  backend:
    build: ./backend
    ports:
      - '3001:3001'
    environment:
      DATABASE_URL:
postgresql://bookstore_user:${DB_PASSWORD}@postgres:5432/bookstore
      REDIS_URL: redis://redis:6379
      JWT_ACCESS_SECRET: ${JWT_ACCESS_SECRET}
      JWT_REFRESH_SECRET: ${JWT_REFRESH_SECRET}
    depends_on: [postgres, redis]
    volumes:
      - ./backend:/app
      - /app/node_modules
volumes:
  pgdata:
```

А.2 Текст файлу 001_create_catalog.sql

```
CREATE TABLE publishers (
  id         SERIAL PRIMARY KEY,
  name       VARCHAR(255) NOT NULL UNIQUE,
  country    VARCHAR(100),
  website    VARCHAR(255),
  created_at TIMESTAMPTZ DEFAULT NOW()
);
```

```
CREATE TABLE authors (
  id         SERIAL PRIMARY KEY,
  first_name VARCHAR(100) NOT NULL,
  last_name  VARCHAR(100) NOT NULL,
  bio        TEXT,
  photo_url  VARCHAR(500)
);

CREATE TABLE books (
  id             SERIAL PRIMARY KEY,
  title          VARCHAR(500)   NOT NULL,
  isbn13         VARCHAR(13)    UNIQUE,
  description    TEXT,
  price          NUMERIC(10,2)  NOT NULL CHECK (price >= 0),
  stock_qty      INTEGER        NOT NULL DEFAULT 0 CHECK (stock_qty >= 0),
  cover_url      VARCHAR(500),
  published_year SMALLINT,
  page_count     SMALLINT,
  language       VARCHAR(10)    DEFAULT 'uk',
  publisher_id   INTEGER REFERENCES publishers(id) ON DELETE SET NULL,
  search_vector  TSVECTOR,
  created_at     TIMESTAMPTZ DEFAULT NOW(),
  updated_at     TIMESTAMPTZ DEFAULT NOW()
);

CREATE TABLE book_authors (
  book_id   INTEGER REFERENCES books(id)   ON DELETE CASCADE,
  author_id INTEGER REFERENCES authors(id) ON DELETE CASCADE,
  role      VARCHAR(50) DEFAULT 'author',
  PRIMARY KEY (book_id, author_id)
);

-- GIN-індекс для повнотекстового пошуку
CREATE INDEX idx_books_search ON books USING GIN(search_vector);

-- Тригер оновлення search_vector при зміні title або description
CREATE OR REPLACE FUNCTION update_book_search_vector()
RETURNS TRIGGER AS $$ BEGIN
  NEW.search_vector :=
    setweight(to_tsvector('ukrainian', coalesce(NEW.title,'')), 'A') ||
    setweight(to_tsvector('ukrainian', coalesce(NEW.description,'')), 'B');
  RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trg_books_search_vector
BEFORE INSERT OR UPDATE ON books
FOR EACH ROW EXECUTE FUNCTION update_book_search_vector();
```

А.3 Текст файлу 002_create_orders.sql

```
CREATE TYPE user_role    AS ENUM ('customer','admin');
CREATE TYPE order_status AS ENUM
  ('pending','confirmed','shipped','delivered','cancelled');

CREATE TABLE users (
  id           SERIAL PRIMARY KEY,
  email        VARCHAR(255) NOT NULL UNIQUE,
  password_hash VARCHAR(60)  NOT NULL,
  first_name    VARCHAR(100),
  last_name     VARCHAR(100),
  role         user_role   NOT NULL DEFAULT 'customer',
  is_active     BOOLEAN     NOT NULL DEFAULT TRUE,
  created_at    TIMESTAMPTZ DEFAULT NOW()
);

CREATE TABLE orders (
  id            SERIAL PRIMARY KEY,
  user_id       INTEGER REFERENCES users(id),
  address_id    INTEGER,
  status        order_status  NOT NULL DEFAULT 'pending',
  total_price   NUMERIC(12,2) NOT NULL,
  payment_method VARCHAR(50),
  created_at    TIMESTAMPTZ DEFAULT NOW(),
  updated_at    TIMESTAMPTZ DEFAULT NOW()
);

CREATE TABLE order_items (
  id        SERIAL PRIMARY KEY,
  order_id  INTEGER REFERENCES orders(id) ON DELETE CASCADE,
  book_id   INTEGER REFERENCES books(id),
  quantity  SMALLINT      NOT NULL CHECK (quantity > 0),
  unit_price NUMERIC(10,2) NOT NULL
);

CREATE TABLE cart_items (
  id      SERIAL PRIMARY KEY,
  user_id  INTEGER REFERENCES users(id) ON DELETE CASCADE,
  book_id  INTEGER REFERENCES books(id) ON DELETE CASCADE,
  quantity SMALLINT NOT NULL DEFAULT 1 CHECK (quantity > 0),
```

```
  added_at TIMESTAMPTZ DEFAULT NOW(),
  UNIQUE (user_id, book_id)
);

CREATE TABLE reviews (
  id          SERIAL PRIMARY KEY,
  user_id     INTEGER REFERENCES users(id),
  book_id     INTEGER REFERENCES books(id) ON DELETE CASCADE,
  rating      SMALLINT NOT NULL CHECK (rating BETWEEN 1 AND 5),
  body        TEXT,
  is_moderated BOOLEAN    DEFAULT FALSE,
  created_at   TIMESTAMPTZ DEFAULT NOW(),
  UNIQUE (user_id, book_id)
);
```

А.4 Текст файлу authMiddleware.js

```
const jwt = require('jsonwebtoken');
const { AppError } = require('../utils/AppError');

const authMiddleware = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return next(new AppError('Токен автентифікації відсутній', 401));
  }
  const token = authHeader.split(' ')[1];
  try {
    const payload = jwt.verify(token, process.env.JWT_ACCESS_SECRET);
    req.user = { id: payload.sub, role: payload.role };
    next();
  } catch (err) {
    const msg = err.name === 'TokenExpiredError'
      ? 'Термін дії токена вичерпано'
      : 'Токен недійсний';
    return next(new AppError(msg, 401));
  }
};

// Фабрика middleware для перевірки ролей
const roleMiddleware = (...allowedRoles) => (req, res, next) => {
  if (!allowedRoles.includes(req.user?.role)) {
    return next(new AppError('Недостатньо прав', 403));
  }
  next();
};
```

```
module.exports = { authMiddleware, roleMiddleware };
```

А.5 Текст файлу orderService.js (фрагмент)

```
async createOrder(userId, { addressId, paymentMethod }) {
  const client = await pool.connect();
  try {
    await client.query('BEGIN');

    // 1. Отримати позиції кошика з поточними цінами
    const { rows: items } = await client.query(
      `SELECT ci.book_id, ci.quantity, b.price, b.stock_qty, b.title
       FROM cart_items ci JOIN books b ON b.id = ci.book_id
       WHERE ci.user_id = $1`,
      [userId]
    );
    if (!items.length) throw new AppError('Кошик порожній', 400);

    // 2. Перевірити та списати залишки в одній транзакції
    for (const item of items) {
      if (item.stock_qty < item.quantity)
        throw new AppError(`Недостатня кількість: ${item.title}`, 409);
      await client.query(
        'UPDATE books SET stock_qty = stock_qty - $1 WHERE id = $2',
        [item.quantity, item.book_id]
      );
    }

    // 3. Підсумкова сума та створення запису замовлення
    const total = items.reduce((s,i) => s + i.price * i.quantity, 0);
    const { rows:[order] } = await client.query(
      `INSERT INTO orders(user_id,address_id,payment_method,total_price)
       VALUES($1,$2,$3,$4) RETURNING *`,
      [userId, addressId, paymentMethod, total.toFixed(2)]
    );

    // 4. Позиції замовлення
    for (const item of items) {
      await client.query(
        `INSERT INTO order_items(order_id,book_id,quantity,unit_price)
         VALUES($1,$2,$3,$4)`,
        [order.id, item.book_id, item.quantity, item.price]
      );
    }
```

```
    // 5. Очистити кошик і підтвердити транзакцію
    await client.query('DELETE FROM cart_items WHERE user_id=$1',[userId]);
    await client.query('COMMIT');
    return order;
  } catch (err) {
    await client.query('ROLLBACK');
    throw err;
  } finally {
    client.release(); // завжди повернути з’єднання до пулу
  }
}
```

А.6 Текст файлу booksService.js (фрагмент)

```
async findAll({ q, genre, minPrice, maxPrice, page=1, limit=20 }) {
  const cacheKey = 'books:' + JSON.stringify(arguments[0]);
  const cached   = await redis.get(cacheKey);
  if (cached) return JSON.parse(cached);

  const conditions = [], params = [];
  let idx = 1;
  if (q) {
    conditions.push(
      `search_vector @@ plainto_tsquery('ukrainian', $${idx++})`
    );
    params.push(q);
  }
  if (genre)    { conditions.push(`EXISTS(SELECT 1 FROM book_genres bg
    JOIN genres g ON g.id=bg.genre_id
    WHERE bg.book_id=b.id AND g.slug=$${idx++})`); params.push(genre); }
  if (minPrice) { conditions.push(`b.price >= $${idx++}`); params.push(minPrice);
}
  if (maxPrice) { conditions.push(`b.price <= $${idx++}`); params.push(maxPrice);
}

  const where = conditions.length ? 'WHERE ' + conditions.join(' AND ') : '';
  const sql = `
    SELECT b.id, b.title, b.price, b.cover_url, b.stock_qty,
           ROUND(AVG(r.rating),1) AS avg_rating,
           COUNT(*) OVER() AS total_count
    FROM books b LEFT JOIN reviews r ON r.book_id=b.id
    ${where} GROUP BY b.id
    ORDER BY ${q ? 'ts_rank(search_vector,plainto_tsquery($1)) DESC,' : ''}
             b.created_at DESC
```

```
    LIMIT $${idx} OFFSET $${idx+1}`;
  params.push(limit, (page-1)*limit);

  const { rows } = await pool.query(sql, params);
  const result = { data:rows, total:rows[0]?.total_count ?? 0, page, limit };
  await redis.set(cacheKey, JSON.stringify(result), 'EX', 600);
  return result;
}
```

А.7 Текст файлу BookCard.jsx

```
import Image    from 'next/image';
import Link     from 'next/link';
import { useDispatch } from 'react-redux';
import { addToCart }   from '@/store/cartSlice';
import { StarRating }  from '@/components/ui/StarRating';

export const BookCard = ({ book }) => {
  const dispatch = useDispatch();

  const handleAddToCart = (e) => {
    e.preventDefault();                    // не переходити за посиланням
    dispatch(addToCart({ bookId: book.id, quantity: 1 }));
  };

  return (
    <Link href={`/books/${book.id}`}
      className='group flex flex-col bg-white rounded-lg shadow
                hover:shadow-md transition-shadow'>
      <div className='relative aspect-[2/3] overflow-hidden rounded-t-lg'>
        <Image
          src={book.cover_url || '/images/no-cover.png'}
          alt={book.title} fill
          className='object-cover group-hover:scale-105 transition-transform'
          sizes='(max-width:768px) 50vw, 25vw'
        />
      </div>
      <div className='flex flex-col gap-1 p-3 flex-grow'>
        <h3 className='text-sm font-semibold line-clamp-2'>{book.title}</h3>
        <p  className='text-xs text-gray-500'>{book.author_name}</p>
        <StarRating value={book.avg_rating} size='sm' />
        <div className='flex items-center justify-between mt-auto pt-2'>
          <span className='font-bold text-blue-700'>
            {Number(book.price).toFixed(2)} грн
          </span>
```

```
        <button onClick={handleAddToCart}
          disabled={book.stock_qty === 0}
          className='text-xs px-2 py-1 bg-blue-600 text-white rounded
                  disabled:opacity-40 hover:bg-blue-700'>
          {book.stock_qty > 0 ? 'До кошика' : 'Немає'}
        </button>
      </div>
    </div>
  </Link>
 );
};
```

А.8 Текст файлу cartSlice.js

```
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
import api from '@/api/axiosClient';

export const addToCartThunk = createAsyncThunk(
 'cart/addItem',
 async ({ bookId, quantity }, { rejectWithValue }) => {
  try {
   const { data } = await api.post('/cart/items', { bookId, quantity });
   return data;
  } catch (err) {
   return rejectWithValue(err.response?.data?.message ?? 'Помилка');
  }
 }
);

const cartSlice = createSlice({
 name: 'cart',
 initialState: { items: [], status: 'idle', error: null },
 reducers: {
  // Оптимістичне оновлення – миттєва реакція інтерфейсу
  addToCart(state, { payload: { bookId, quantity } }) {
   const existing = state.items.find(i => i.book_id === bookId);
   if (existing) existing.quantity += quantity;
   else state.items.push({ book_id: bookId, quantity });
  },
  removeFromCart(state, { payload: id }) {
   state.items = state.items.filter(i => i.id !== id);
  },
 },
 extraReducers: builder => builder
  .addCase(addToCartThunk.fulfilled, (state, { payload }) => {
```

```
      state.items  = payload.items; // синхронізація з сервером
      state.status = 'idle';
    })
    .addCase(addToCartThunk.rejected, (state, { payload }) => {
      state.error = payload;        // відкат при помилці
    }),
});

export const { addToCart, removeFromCart } = cartSlice.actions;
export default cartSlice.reducer;
```

А.9 Текст файлу authService.js

```
const bcrypt = require('bcryptjs');
const jwt    = require('jsonwebtoken');
const { pool }     = require('../db/pool');
const redis        = require('../db/redis');
const { AppError } = require('../utils/AppError');

const SALT=12, ACCESS_TTL='15m', REFRESH_TTL='7d',
REFRESH_S=604800;

class AuthService {
  _tokens(userId, role) {
    return {
      access:  jwt.sign({sub:userId,role}, process.env.JWT_ACCESS_SECRET,
{expiresIn:ACCESS_TTL}),
      refresh: jwt.sign({sub:userId}, process.env.JWT_REFRESH_SECRET,
{expiresIn:REFRESH_TTL}),
    };
  }

  async register({ email, password, firstName, lastName }) {
    const dup = await pool.query('SELECT id FROM users WHERE
email=$1',[email]);
    if (dup.rows.length) throw new AppError('Email вже зареєстровано',409);
    const hash = await bcrypt.hash(password, SALT);
    const { rows:[u] } = await pool.query(
      `INSERT INTO users(email,password_hash,first_name,last_name)
       VALUES($1,$2,$3,$4) RETURNING id,email,role`,
      [email,hash,firstName,lastName]
    );
    const tokens = this._tokens(u.id, u.role);
    await redis.set('refresh:'+u.id, tokens.refresh, 'EX', REFRESH_S);
    return { user:u, ...tokens };
```

```
  }

  async login({ email, password }) {
    const { rows:[u] } = await pool.query(
      'SELECT id,email,role,password_hash,is_active FROM users WHERE email=$1',
      [email]
    );
    // Однакова помилка для відсутнього email і невірного пароля (anti-enumeration)
    if (!u || !await bcrypt.compare(password, u.password_hash))
      throw new AppError('Невірний email або пароль', 401);
    if (!u.is_active) throw new AppError('Акаунт деактивовано', 403);
    const tokens = this._tokens(u.id, u.role);
    await redis.set('refresh:'+u.id, tokens.refresh, 'EX', REFRESH_S);
    return { user:{id:u.id,email:u.email,role:u.role}, ...tokens };
  }

  async refresh(refreshToken) {
    let payload;
    try { payload = jwt.verify(refreshToken, process.env.JWT_REFRESH_SECRET); }
    catch { throw new AppError('Refresh-токен недійсний', 401); }
    const stored = await redis.get('refresh:'+payload.sub);
    if (!stored || stored !== refreshToken)
      throw new AppError('Сесію завершено', 401);
    const { rows:[u] } = await pool.query(
      'SELECT id,role FROM users WHERE id=$1', [payload.sub]
    );
    const tokens = this._tokens(u.id, u.role);
    await redis.set('refresh:'+u.id, tokens.refresh, 'EX', REFRESH_S);
    return tokens;
  }
}
module.exports = new AuthService();
```

ДОДАТОК Б

Слайди презентації

Рисунок Б.1 – Слайд 1

Рисунок Б.2 – Слайд 2

Аналіз існуючих книжкових веб-платформ

2 / 9

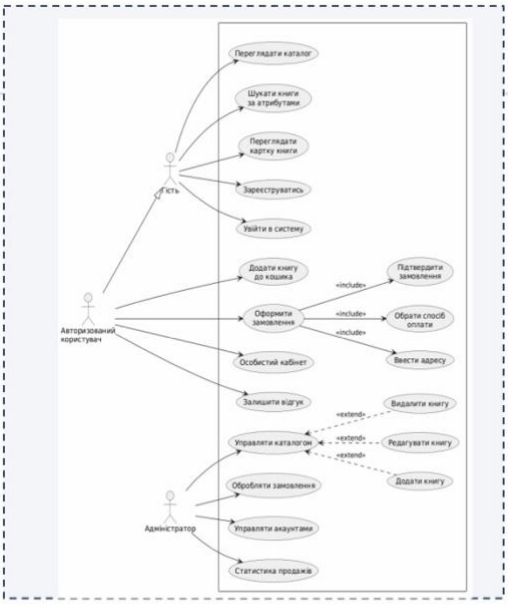
Критерій порівняння	Yakaboo	Rozetka (книги)	Amazon Books	Розроблювана система
Спеціалізація на книгах	Так	Частково	Так	Так
Пошук за ISBN	Так	Ні	Так	Так
Фільтрація за жанром, автором, видавцем	Часткова	Базова	Розширена	Повна
Кошик та оформлення замовлення	Так	Так	Так	Так
Особистий кабінет користувача	Так	Так	Так	Так
Система відгуків та рейтинг	Так	Так	Так	Так
Рекомендаційна система	Ні	Ні	Так (ML)	Базова (жанр)
Адмін-панель управління каталогом	Так	Загальна	Так	Спеціалізована
Україномовний інтерфейс	Так	Так	Ні	Так
Адаптивний дизайн (mobile)	Так	Так	Так	Так
Відкритий вихідний код	Ні	Ні	Ні	Так (MIT)
Вартість для бізнесу	Комерційна	Комерційна	Комерційна	Безкоштовна

Порівняльний аналіз існуючих книжкових веб-платформ

Рисунок Б.3 – Слайд 3

Варіанти використання системи

3 / 9



UML-діаграма варіантів використання вебсайту продажу книг

Рисунок Б.4 – Слайд 4

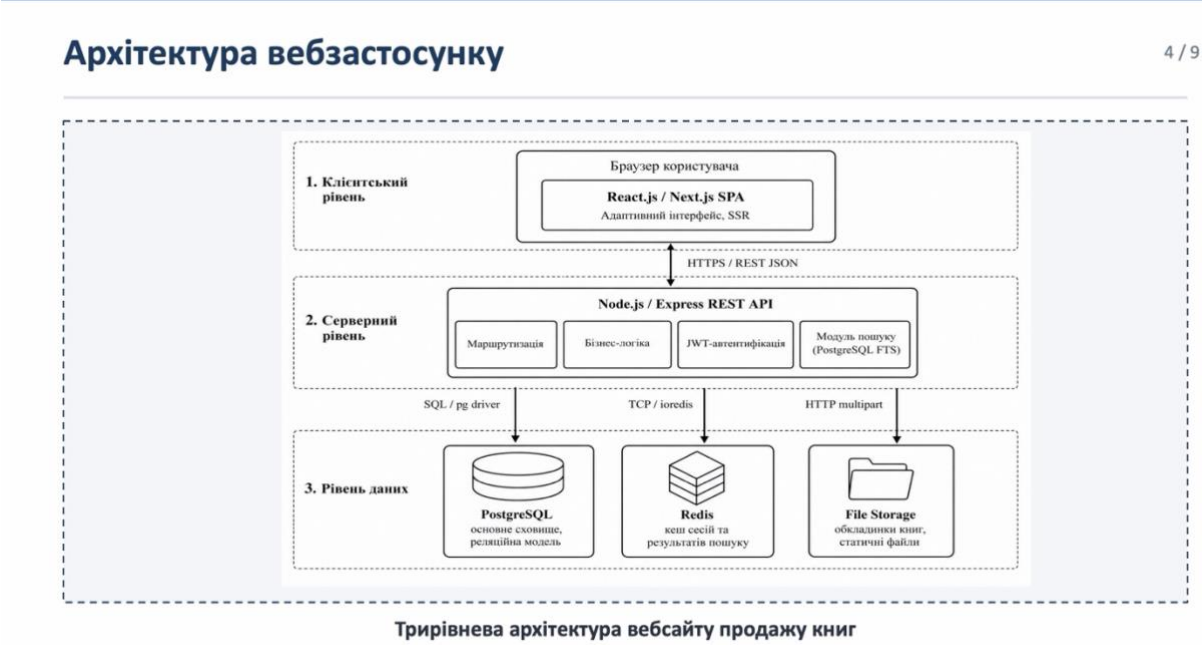


Рисунок Б.5 – Слайд 5

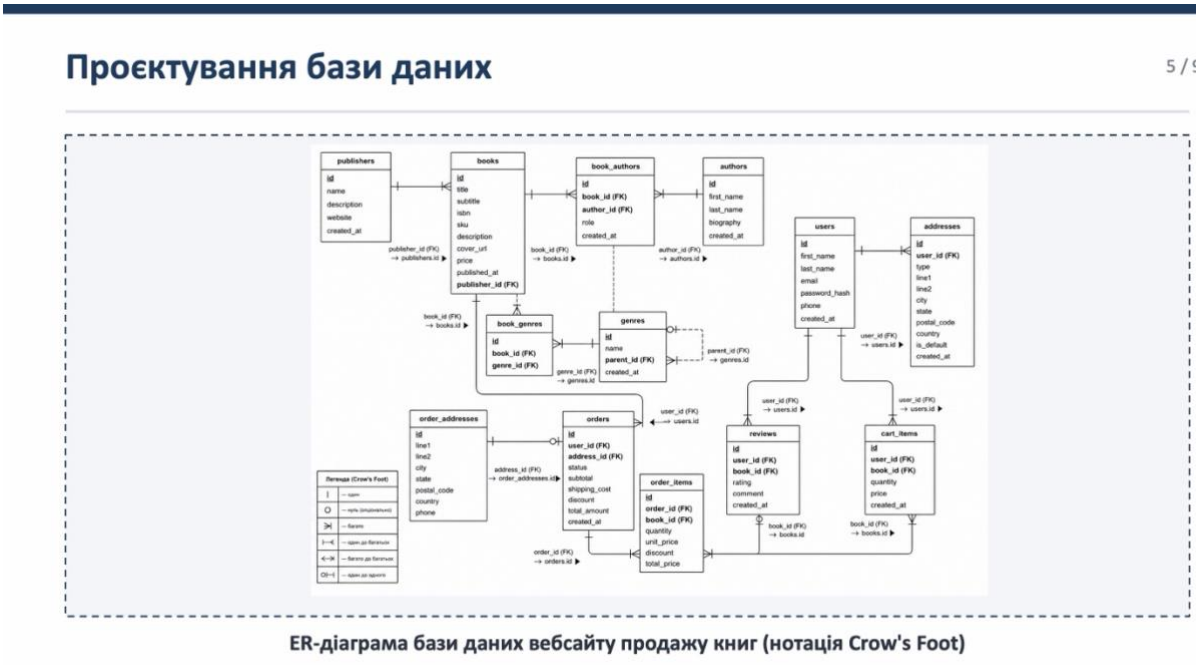


Рисунок Б.6 – Слайд 6

Специфікація REST API

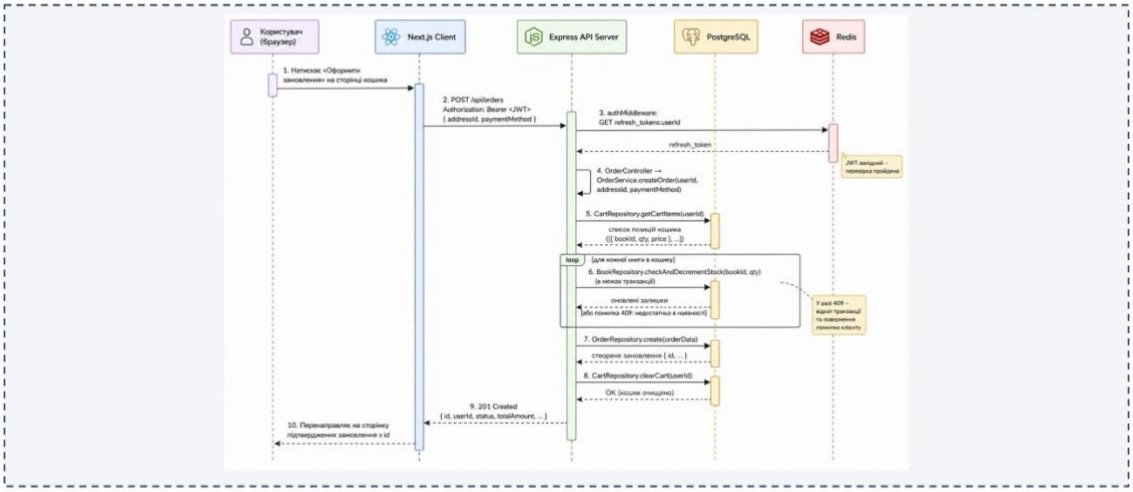
6 / 9

Метод	URL	Доступ	Опис
GET	/api/books	Публічний	Пагінований список книг з фільтрацією та сортуванням
GET	/api/books/:id	Публічний	Деталі книги з авторами, жанрами, відгуками
POST	/api/books	Адмін	Створення нової книги
PUT	/api/books/:id	Адмін	Оновлення даних книги
DELETE	/api/books/:id	Адмін	Видалення книги з каталогу
POST	/api/auth/register	Публічний	Реєстрація нового користувача
POST	/api/auth/login	Публічний	Вхід, повертає access та refresh токени
POST	/api/auth/refresh	Публічний	Оновлення access-токена за refresh-токеном
POST	/api/auth/logout	Користувач	Анулювання refresh-токена в Redis
GET	/api/cart	Користувач	Отримання вмісту кошика
POST	/api/cart/items	Користувач	Додавання книги до кошика
PATCH	/api/cart/items/:id	Користувач	Зміна кількості позиції в кошику
DELETE	/api/cart/items/:id	Користувач	Видалення позиції з кошика
POST	/api/orders	Користувач	Оформлення замовлення з кошика
GET	/api/orders	Користувач	Список замовлень поточного користувача
PATCH	/api/orders/:id/status	Адмін	Зміна статусу замовлення
GET	/api/books/:id/reviews	Публічний	Список відгуків для книги
POST	/api/books/:id/reviews	Користувач	Створення відгуку; один відгук на книгу від користувача

Рисунок Б.7 – Слайд 7

Сценарій оформлення замовлення

7 / 9



UML-діаграма послідовності сценарію «Оформлення замовлення»

Рисунок Б.8 – Слайд 8

Реалізація вебсайту

8 / 9

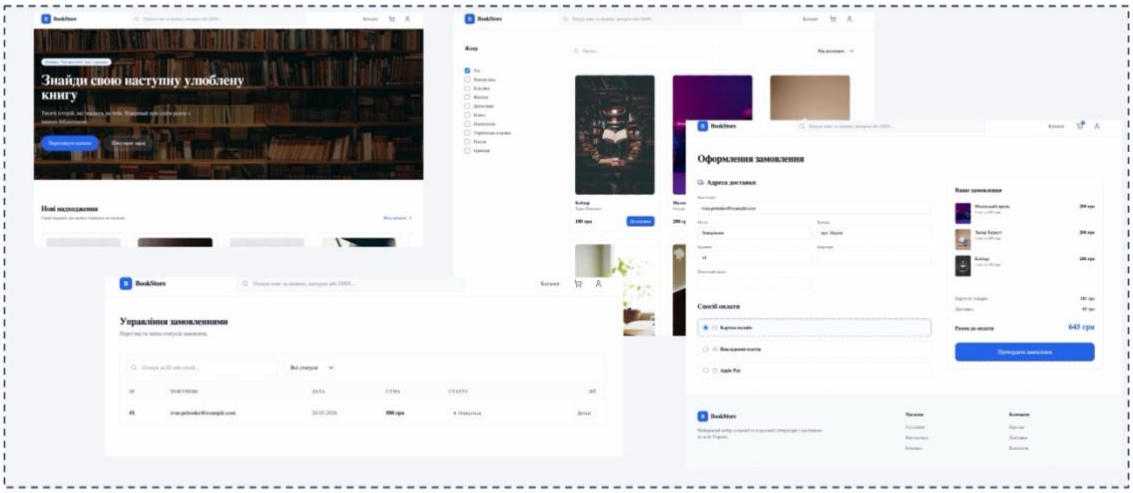


Рисунок Б.9 – Слайд 9

Висновки

9 / 9

1. Виконано порівняльний аналіз трьох книжкових веб-платформ (Yakaboo, Rozetka, Amazon Books); сформовано 10 функціональних і 6 нефункціональних вимог; обґрунтовано вибір стеку React.js / Next.js + Node.js / Express + PostgreSQL + Redis.
2. Спроєктовано трирівневу клієнт-серверну архітектуру, реляційну схему БД з 13 сутностей та специфікацію з 18 ендпоінтів REST API з розподілом прав доступу на три ролі.
3. Реалізовано серверну частину (Controller–Service–Repository, повнотекстовий пошук через tsvector з GIN-індексом, кешування Redis з TTL 10 хв) та клієнтську частину (SSR через Next.js, Redux Toolkit, JWT-автентифікація access/refresh).
4. Комплексне тестування підтвердило відповідність вимогам: покриття коду 87 % (Jest); медіана часу відповіді 28 мс при 100 одночасних користувачах (k6), що у 4,9 рази менше нефункціонального порогу 300 мс.
5. Час оформлення замовлення скорочено до 2–3 хв; час пошуку книги зменшено на 40–60 %; трудомісткість адміністрування каталогу — приблизно на 40 %. Мету роботи досягнуто.

Рисунок Б.10 – Слайд 10