МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ЗУЄВА НАТАЛІЯ ЮРІЇВНА

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____ О. В. Зелінська
«____»_____ 20__ р.

РОЗРОБКА СЕРВЕРНОЇ ЧАСТИНИ МОБІЛЬНОГО ЗАСТОСУНКУ ДЛЯ СПІЛЬНОТИ КНИГОЛЮБІВ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Т. В. Січко, доцент кафедри
інформаційних технологій,
к. т. н., доцент
_____

Оцінка: _____ / _____ / _____
(бали/за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____

Вінниця - 2025

АНОТАЦІЯ

Зуєва Н.Ю. Розробка серверної частини мобільного застосунку для спільноти книголюбів. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса. Вінниця 2025.

У кваліфікаційній (бакалаврській) роботі досліджено потреби книголюбів, проаналізовано аналоги, визначено вимоги та спроектовано серверну частину мобільного застосунку. Розроблено гнучку архітектуру з потенціалом мікросервісної еволюції, модель даних, системи автентифікації, авторизації та RESTful API. Враховано безпеку, розгортання і тестування. Створено готову, продуктивну, масштабовану та безпечну серверну інфраструктуру для застосунку україномовної спільноти книголюбів.

Ключові слова: мобільний застосунок, серверна частина, Node.js, PostgreSQL, RESTful API, безпека, масштабованість, спільнота книголюбів.

ABSTRACT

Zuieva N.Y. Development of the backend for a mobile application for a book lovers community. Specialty 122 "Computer Sciences", educational program "Computer Sciences". Vasyl Stus Donetsk National University. Vinnytsia 2025.

In this qualification (bachelor's) thesis, book lovers' needs were investigated, analogs analyzed, requirements defined, and the backend of a mobile application designed. A flexible architecture with microservice evolution potential, a data model, authentication and authorization systems, and RESTful APIs were developed. Particular attention was given to security, deployment, and testing. The result is an implementation-ready server infrastructure ensuring high performance, scalability, and security for the mobile application for the Ukrainian-speaking book lovers community.

Keywords: mobile application, backend, Node.js, PostgreSQL, RESTful API, security, scalability, book lovers community.

ЗМІСТ

ВСТУП

У сучасному цифровому світі мобільні застосунки стали невід'ємною частиною повсякденного життя, трансформуючи способи взаємодії, отримання інформації та формування спільнот за інтересами. Особливе місце серед таких спільнот посідають книголюби – активні читачі, що прагнуть не лише до індивідуального занурення у світ літератури, але й до жвавого обговорення прочитаного, обміну рекомендаціями та участі у тематичних заходах. Створення спеціалізованих мобільних платформ для таких спільнот відкриває широкі можливості для задоволення їхніх унікальних потреб, проте вимагає розробки надійної, функціональної та масштабованої серверної інфраструктури.

Актуальність дослідження зумовлена зростаючим попитом на якісні україномовні цифрові продукти та необхідністю створення платформи, що враховує культурний та мовний контекст української спільноти книголюбів. Більшість існуючих глобальних аналогів мають обмежену підтримку україномовного контенту та не завжди відповідають специфічним очікуванням місцевої аудиторії. Розробка ефективної серверної частини, здатної забезпечити високу продуктивність, безпеку персональних даних, гнучкість для подальшого розвитку та підтримку всіх необхідних функцій, є ключовим завданням для успіху такого проекту.

Метою дослідження є комплексне проектування, обґрунтування вибору технологій та опис ключових аспектів реалізації та тестування серверної частини мобільного застосунку для спільноти книголюбів, з акцентом на забезпечення високої продуктивності, масштабованості, безпеки та підтримки україномовного контенту.

Для досягнення поставленої мети було визначено наступні завдання:

1. Провести детальний аналіз потреб цільової аудиторії – спільноти книголюбів – та дослідити існуючі аналогічні програмні рішення на ринку.
2. Сформулювати вичерпний перелік функціональних та

нефункціональних вимог до серверної частини застосунку.

3. Обґрунтувати вибір оптимального технологічного стеку (платформи, мови програмування, фреймворків, систем управління базами даних та кешування) для реалізації серверної логіки.

4. Розробити архітектуру серверної частини, що передбачає можливість масштабування та потенційної еволюції до мікросервісного підходу.

5. Спроектувати детальну реляційну модель бази даних на основі PostgreSQL для зберігання інформації про книги, користувачів, авторів, відгуки та їхні взаємозв'язки.

6. Розробити систему автентифікації та авторизації користувачів на основі JWT та рольової моделі доступу.

7. Описати процес розробки ключових API-ендпоінтів та їх інтеграцію з базою даних, системою кешування та механізмами асинхронної обробки завдань.

8. Визначити стратегії та інструменти для налаштування середовища розробки, розгортання серверної частини на хмарній інфраструктурі, а також методики її тестування та забезпечення безпеки.

Об'єктом дослідження є процеси проектування, розробки та тестування серверної інфраструктури для соціально-орієнтованих мобільних застосунків.

Предметом дослідження є архітектурні підходи, технологічні рішення, моделі даних, механізми безпеки та методи тестування, що застосовуються при створенні серверної частини мобільного застосунку для спільноти книголюбів.

Робота складається зі вступу, трьох основних розділів, висновків, списку використаних джерел та додатків. У першому розділі проведено аналіз предметної області та сформульовано вимоги до системи. Другий розділ присвячено детальному проектуванню серверної частини, включаючи вибір технологій, розробку архітектури, моделювання бази даних та систем безпеки. Третій розділ описує ключові аспекти реалізації, налаштування середовища, розгортання, а також стратегії безпеки та тестування розробленого рішення.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГ ДО СЕРВЕРНОЇ ЧАСТИНИ

1.1 Огляд потреб спільноти книголюбів у мобільному застосунку

Спільнота книголюбів, об'єднана захопленням читанням та обговоренням літератури, все активніше переходить у цифровий простір. Мобільні застосунки стають для них ключовим інструментом взаємодії, задовольняючи специфічні потреби. Всебічний аналіз цих потреб є необхідною передумовою для проектування серверної частини такого застосунку. Даний підрозділ досліджує основні потреби сучасної спільноти книголюбів, враховуючи функціональні, соціальні, інформаційні, культурні аспекти та специфіку україномовної аудиторії.

Ключові потреби спільноти книголюбів (рис.1.1):

- Доступ до вичерпної інформації про книги: Фундаментальна потреба – зручний доступ до інформації про книги. Це включає бібліографічні дані (назва, автор, рік, видавництво), детальні анотації, рецензії (професійні та користувацькі), рейтинги, інформацію про різні видання (паперове, електронне, аудіо) та переклади, особливо українською мовою. Мобільний застосунок повинен мати інтуїтивний пошук за різними критеріями (жанр, автор, назва, ISBN). Сучасні рекомендаційні системи, що аналізують вподобання користувачів, значно підвищують залученість. Серверна частина має ефективно обробляти та зберігати великі масиви даних про книги та активність користувачів, забезпечуючи швидкий доступ через API. Важлива можливість для користувачів додавати нові книги (з модерацією) або пропонувати корекції.
- Можливості для соціальної взаємодії: Книголюби прагнуть обміну враженнями, обговорення творів у тематичних групах/форумах, участі у спільних читацьких викликах та книжкових клубах. Функціонал для групових чатів або дискусійних гілок вимагає від сервера підтримки асинхронних повідомлень, сповіщень та обробки значного потоку запитів. Користувацькі рейтинги та відгуки є важливим елементом соціального підтвердження. Сервер має надійно зберігати

цей контент, індексувати для пошуку та забезпечувати механізми модерації. Можливість стежити за активністю інших користувачів посилює соціальний аспект.

- Високий рівень персоналізації: Користувачі очікують індивідуально підібрані рекомендації книг, що відповідають їхнім літературним смакам та історії читання. Потрібні гнучкі інструменти для організації власної цифрової бібліотеки (списки "Прочитане", "Бажаю прочитати", "Улюблене" тощо) та відстеження прогресу. Це вимагає від серверної частини інтеграції алгоритмів машинного навчання (колаборативна фільтрація, контент-орієнтований аналіз) для аналізу поведінки користувачів. Ефективні системи рекомендацій використовують комплексний аналіз історії переглядів та оцінок. Такі системи потребують потужної БД для профілів користувачів та високопродуктивної інфраструктури.
- Врахування культурного та мовного контексту: Для україномовних книголюбів важливий доступ до україномовного контенту (книги вітчизняних та зарубіжних авторів у перекладі, відгуки українською). Застосунок повинен мати україномовний інтерфейс, пріоритезувати українські видання, авторів та сприяти популяризації української літератури. Серверна частина має гарантувати коректну обробку, зберігання та пошук україномовних даних (підтримка української абетки, правил сортування). Якісна локалізація значно підвищує привабливість застосунків та є критичною для українського ринку. Це включає можливість фільтрації контенту за мовою видання.
- Організація та участь у читацьких подіях: Потреба в організації та участі в онлайн/офлайн заходах (презентації, зустрічі з авторами, фестивалі, марафони). Застосунок може слугувати платформою для анонсів, календаря подій, сповіщень та реєстрації. Серверна частина має підтримувати функції управління подіями та інтеграцію з push-повідомленнями для інформування користувачів.
- Забезпечення безпеки та приватності даних: Фундаментальна вимога при обробці персональної інформації (списки книг, відгуки, коментарі). Серверна частина повинна гарантувати шифрування даних під час передачі (in transit) та зберігання (at rest), захист від несанкціонованого доступу та кіберзагроз.

Дотримання GDPR та українського законодавства (Закон "Про захист персональних даних") є обов'язковим. Впровадження HTTPS, надійних механізмів автентифікації та авторизації, регулярні аудити безпеки є критичними.

– Висока продуктивність, надійність та доступність: Застосунок має працювати швидко, стабільно, без збоїв, навіть під час пікових навантажень. Серверна частина має бути спроектована з урахуванням масштабованості для адаптації до зростання кількості користувачів. Використання хмарних інфраструктур (AWS, GCP, Azure) може спростити реалізацію цих вимог.

Рисунок 1.1 – Ключові потреби спільноти книголюбів

Джерело: побудовано автором

Таким чином, аналіз потреб спільноти книголюбів виявляє ключові вимоги: легкий доступ до інформації про книги, розширені соціальні можливості, глибока персоналізація, підтримка україномовного контенту, функціонал для подій, високий рівень безпеки та приватності, стабільна продуктивність. Ці потреби формують основу для визначення детальних функціональних та нефункціональних вимог до серверної частини. Ретельне врахування цих потреб є запорукою створення цінного та затребуваного продукту.

1.2 Аналіз існуючих аналогів серверних рішень для подібних застосунків

Розробка серверної частини вимагає аналізу існуючих аналогічних рішень для ідентифікації перевірених підходів, успішних патернів та потенційних слабких сторін конкурентів. Аналогами виступають популярні платформи для книголюбів з функціоналом пошуку книг, обміну відгуками, рекомендаціями та соціальною взаємодією.

Огляд провідних аналогів:

- Goodreads (Amazon): Найбільша міжнародна платформа з мільйонами користувачів та величезною базою книг і відгуків. Функціонал включає пошук, віртуальні полиці, відгуки, оцінки, тематичні групи. Ймовірна серверна архітектура – мікросервісна, що забезпечує гнучкість та масштабованість для обробки великих обсягів даних. Використовує RESTful API, ймовірно, комбінацію реляційних (PostgreSQL/MySQL) та NoSQL баз даних. Потужна рекомендаційна система базується на машинному навчанні. Недоліки для України: обмежена локалізація інтерфейсу та підтримка україномовного контенту. Деякі користувачі відзначають перевантаженість інтерфейсу, інтерфейс Goodreads зображено на рис/ 1.2 [1].

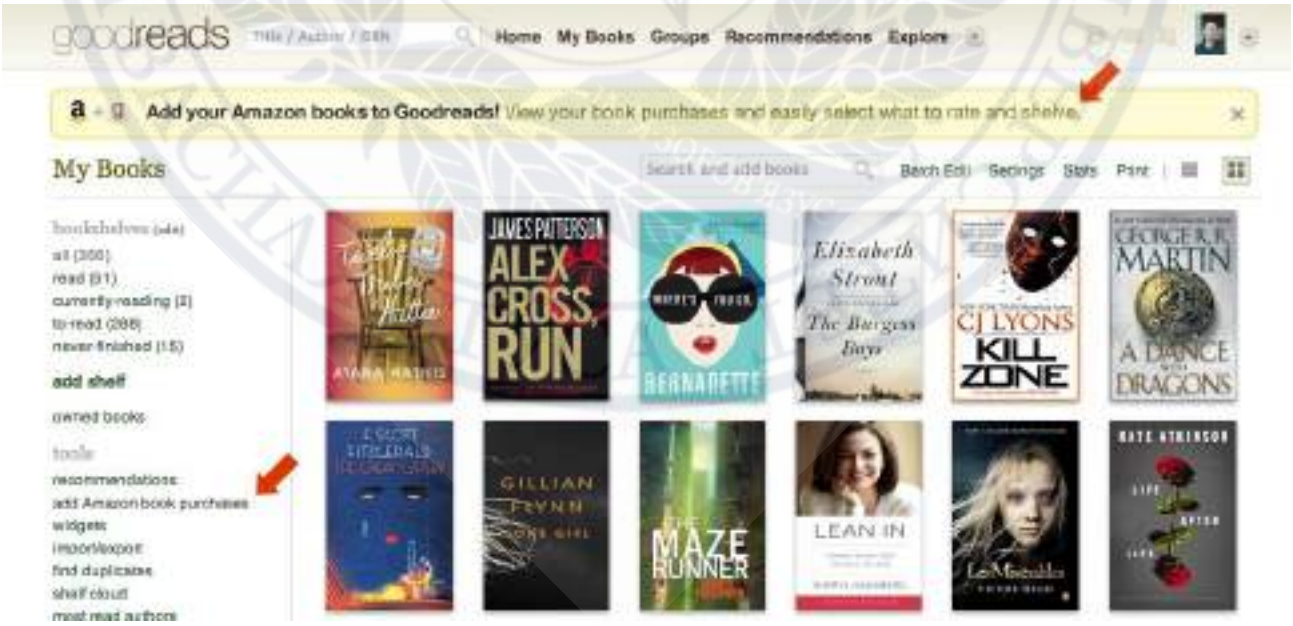


Рисунок 1.2 – Інтерфейс Goodreads із функцією пошуку книг

Джерело: [1]

– Bookly: Мобільний застосунок, орієнтований на персональне відстеження читацької активності та формування звичок (списки, час читання, цілі, статистика). Серверна частина, ймовірно, використовує Firebase (BaaS від Google), що надає сервіси для зберігання даних (Firestore/Realtime Database), автентифікації, серверної логіки (Cloud Functions) та push-повідомлень (FCM). Це спрощує розробку та забезпечує масштабованість. Переваги: фокус на гейміфікації. Недоліки: обмежена підтримка соціальних взаємодій (групові обговорення, коментування книг), що важливо для спільноти. Інтерфейс Bookly зображено на рис. 1.3 [2].

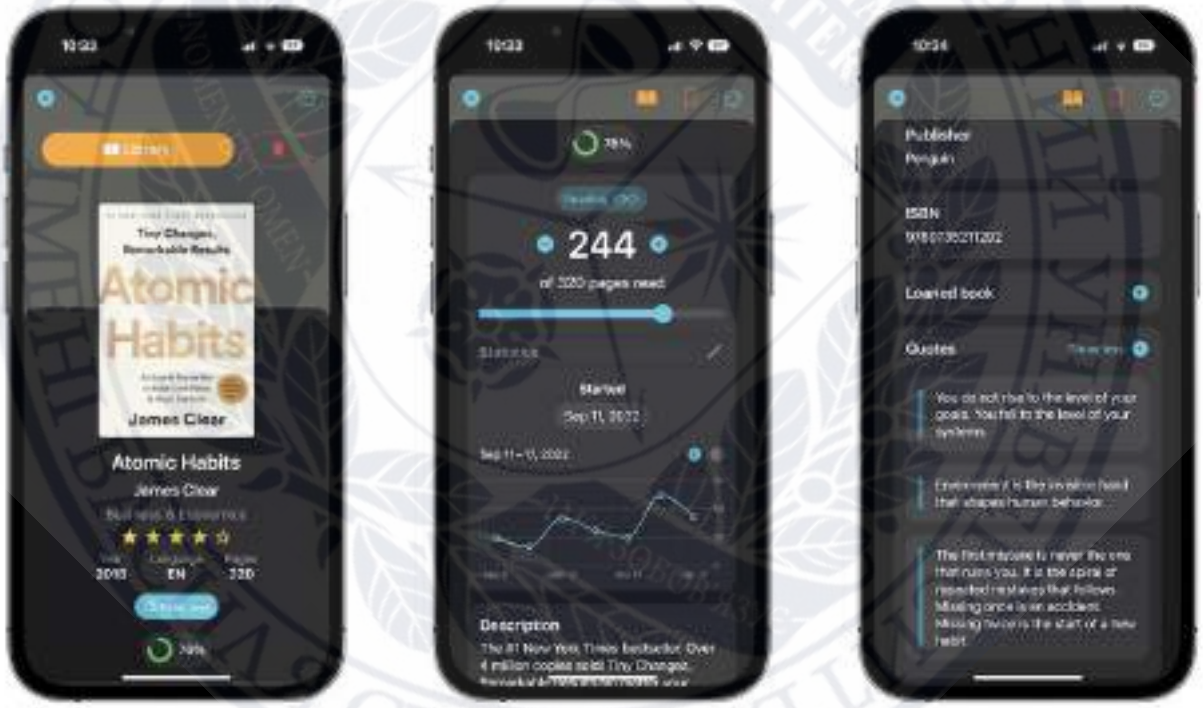


Рисунок 1.3 – Інтерфейс Bookly із функцією відстеження прогресу читання

Джерело: [2]

– Wattpad: Платформа для читання та публікації оригінальних творів користувачами (фанфіки, аматорські романи), з активною спільнотою авторів та читачів, інтерфейс зображено на рис. 1.4. Дозволяє писати, отримувати відгуки,

коментувати. Серверна частина, враховуючи глобальний масштаб та інтенсивність взаємодій, ймовірно, використовує мікросервісну архітектуру та хмарні рішення (можливо, AWS) для високої продуктивності та підтримки потокових даних. Використання потужних хмарних платформ спрощує обробку пікових навантажень. Для даних, ймовірно, гібридний підхід (реляційні БД та NoSQL, наприклад, DynamoDB). Недоліки для класичних книголюбів: сильний ухил в бік аматорської літератури [3].

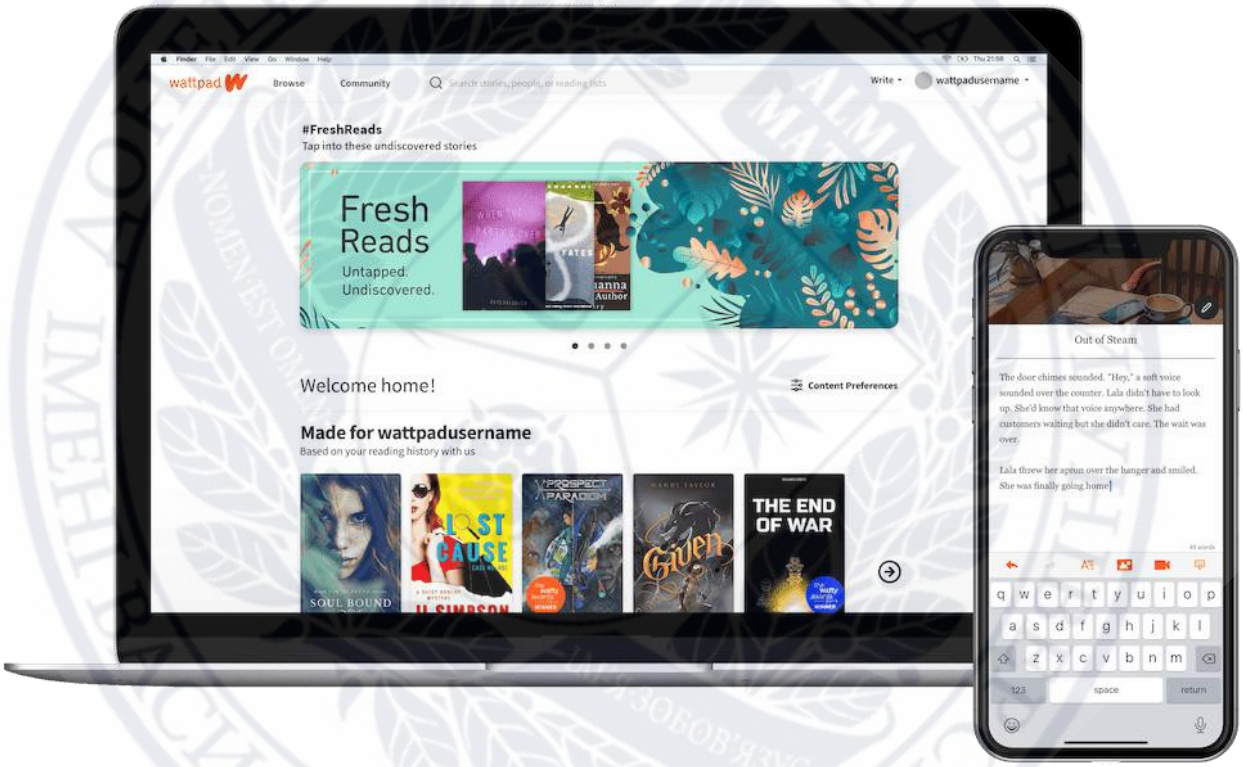


Рисунок 1.4 – Інтерфейс Wattpad

Джерело: [3]

– StoryGraph: Позиціонується як альтернатива Goodreads з акцентом на деталізованому тегуванні книг (настрій, темп) та розширеній статистиці. Функціонал вказує на потребу в гнучкій БД для метаданих та ефективних алгоритмах рекомендацій [4]. Інтерфейс зображено на рис. 1.5.

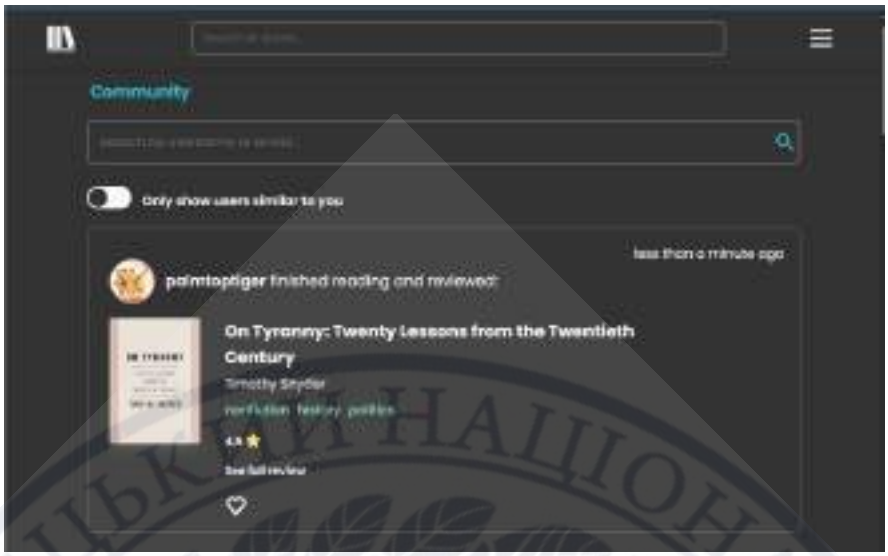


Рисунок 1.5 – Інтерфейс StoryGraph

Джерело: [4]

Аналіз основних параметрів існуючих систем-аналогів включає основний функціонал, технологічний стек, переваги та недоліки, його наведено у табл. 1.1.

Таблиця 1.1

Аналіз основних параметрів існуючих систем-аналогів

Платформа	Основні функції	Ймовірний технологічний стек бекенду	Переваги	Недоліки (для українського ринку)
Goodreads	Пошук книг, віртуальні полиці, відгуки, оцінки, тематичні групи, рекомендації, читацькі челенджі, цитати, книжкові списки.	Мікросервісна архітектура, RESTful API, ймовірно, комбінація реляційних (PostgreSQL/MySQL) та NoSQL баз даних, потужна рекомендаційна система на основі машинного навчання.	Велика база книг та користувачів, розвинені соціальні функції, потужна система рекомендацій.	Обмежена локалізація інтерфейсу, обмежена підтримка україномовного контенту, перевантажений інтерфейс.

Продовження таблиці 1.1

Bookly	Відстеження читацької активності, формування читацьких звичок, гейміфікація	Ймовірно, Firebase (BaaS від Google): Firestore/Realtime Database (зберігання даних), Authentication, Cloud Functions (серверна логіка), FCM (push-повідомлення).	Фокус на персональному відстеженні, гейміфікація, простота у використанні.	Обмежена підтримка соціальних взаємодій.
Wattpad	Платформа для читання та публікації оригінальних творів, активна спільнота авторів та читачів, можливість писати, отримувати відгуки, коментувати.	Ймовірно, мікросервісна архітектура, хмарні рішення (можливо, AWS) для високої продуктивності, гібридний підхід до даних (реляційні БД та NoSQL, наприклад, DynamoDB).	Активна спільнота, можливості для творчості та взаємодії між авторами та читачами.	Сильний ухил в бік аматорської літератури.
StoryGraph	Деталізоване тегування книг (настрій, темп), розширена статистика, персоналізовані рекомендації, відкриття книг за настроєм, читання з друзями, альтернатива Goodreads.	Потребує гнучкої БД для метаданих та ефективних алгоритмів рекомендацій.	Акцент на детальному аналізі та персоналізації, розширена статистика, можливість фільтрувати книги за різними критеріями (настрій, темп, тощо).	Інформація про технологічний стек бекенду не була знайдена.

Джерело: побудовано автором

Порівняльний аналіз існуючих рішень дозволяє зробити кілька ключових узагальнень для проектування власної системи. Щодо API, більшість платформ використовує RESTful підходи, хоча GraphQL виглядає привабливою альтернативою для мобільних пристроїв завдяки гнучкості запитів. У сфері архітектури, мікросервіси є де-факто стандартом для масштабних проектів, тоді як PaaS/BaaS рішення можуть прискорити розробку MVP, але з потенційними обмеженнями гнучкості. Це підводить до висновку, що для нового застосунку доцільним є початковий моноліт з чітким розмежуванням модулів та планом переходу до мікросервісів. Бази даних часто представлені гібридним підходом (реляційні та NoSQL) з обов'язковим використанням систем кешування (Redis, Memcached). Безпека, включаючи HTTPS та відповідність GDPR, є пріоритетом для всіх платформ. Однак, саме локалізація та підтримка україномовного контенту є значною слабкістю більшості глобальних платформ, що відкриває ключову конкурентну перевагу. Нарешті, продуктивність та масштабованість вимагають комплексного підходу, що поєднує ефективну архітектуру, оптимізацію БД та кешування.

З огляду на це, ключові міркування для проектування серверної частини нового застосунку включають створення гнучкого, документованого та версіонованого API (RESTful на старті), та масштабованої архітектури (структурований моноліт з можливістю переходу до мікросервісів). Необхідна надійна система зберігання даних, що базується на реляційній СУБД (PostgreSQL) та інтеграції кешу (Redis). Комплексна безпека має охоплювати сучасні механізми автентифікації, авторизації та захисту. Пріоритетом має стати локалізація та повноцінна підтримка української мови. Крім того, серверна частина повинна забезпечувати підтримку розширених соціальних функцій та персоналізації.

Таким чином, аналіз аналогів демонструє, що успішні серверні рішення мають забезпечувати баланс між функціональністю, продуктивністю, безпекою та адаптацією до потреб аудиторії, включаючи культурний та мовний контекст.

1.3 Визначення функціональних вимог до серверної частини

Функціональні вимоги визначають конкретні функції та можливості, які система повинна надавати для забезпечення ефективної взаємодії користувачів та коректної обробки даних. Вони є прямою конкретизацією потреб спільноти книголюбів та враховують практики, виявлені під час аналізу аналогів. Серверна інфраструктура має забезпечувати управління каталогом книг, соціальні взаємодії, персоналізацію, підтримку подій, безпечну автентифікацію та авторизацію, а також інтеграцію з клієнтом через API.

Серед ключових функціональних можливостей системи, перш за все, варто виділити управління інформацією про книги та доступ до каталогу. Система повинна надавати API для комплексного пошуку книг за різними критеріями, з підтримкою повнотекстового режиму, фільтрації та сортування. Для кожної книги має зберігатися та надаватися детальна інформація, включаючи переклади українською. Передбачається також можливість адміністрування каталогу. Невід'ємною частиною є реалізація соціальних функцій, що дозволяє користувачам залишати відгуки, виставляти рейтинги, коментувати та брати участь у тематичних обговореннях, з підтримкою асинхронних сповіщень. Важливим аспектом є персоналізація користувацького досвіду, що включає збір та аналіз даних для формування персоналізованих рекомендацій книг та надання користувачам інструментів для управління власними списками та профілем. Звісно, система повинна мати надійні механізми автентифікації та авторизації, включаючи реєстрацію, вхід через email/пароль та OAuth 2.0, а також управління профілем відповідно до вимог GDPR.

Критичною для успіху на українському ринку є підтримка локалізації та україномовного контенту. Це означає коректне зберігання, обробку та пошук україномовних даних, а також надання локалізованого контенту через API. Для підвищення залученості спільноти передбачається організація та інформаційна підтримка читацьких подій через відповідний API для управління інформацією та можливістю реєстрації. Для підтримки здорової атмосфери необхідна система

модерації контенту, що надає інструменти для адміністраторів та систему скарг для користувачів. Нарешті, для збагачення каталогу та розширення функціоналу планується інтеграція з зовнішніми сервісами та API, такими як Google Books API, OpenLibrary API, та, можливо, соціальними мережами чи платіжними шлюзами. Аналіз основних вимог до системи структуровано представлено у табл. 1.2

Таблиця 1.2

Аналіз основних вимог до системи

Вимога	Опис	Пріоритет
Управління каталогом книг та пошук	API для CRUD-операцій з книгами (адмінами), розширений пошук, фільтрація, сортування, надання детальної інформації про книги.	Високий
Соціальні функції та взаємодія	API для створення/читання/оновлення/видалення відгуків, рейтингів, коментарів; підтримка групових обговорень/чатів; система сповіщень про соціальну активність.	Високий
Персоналізація та система рекомендацій	API для надання персоналізованих рекомендацій книг на основі аналізу поведінки та вподобань користувача; управління особистими списками книг; налаштування профілю користувача.	Високий
Автентифікація, авторизація та управління профілем	API для безпечної реєстрації, входу (включаючи OAuth 2.0), відновлення пароля; управління токенами доступу (JWT); редагування даних профілю користувача; деактивація/видалення акаунта.	Високий
Підтримка локалізації та україномовного контенту	Забезпечення коректного зберігання, обробки та пошуку україномовних даних; надання локалізованого контенту та системних повідомлень через API.	Високий

Продовження таблиці 1.2

Організація та підтримка читацьких подій	API для створення/перегляду/управління інформацією про події; можливість реєстрації користувачів на події; інтеграція з системою push-повідомлень для нагадувань та анонсів.	Середній
Модерація контенту, створеного користувачами	API для модераторів/адміністраторів для перегляду, схвалення, відхилення, редагування або видалення контенту; управління скаргами користувачів; блокування порушників.	Середній
Інтеграція з зовнішніми сервісами та API	Підтримка взаємодії з зовнішніми API (наприклад, Google Books API) для отримання/збагачення даних; кешування зовнішніх даних; можлива інтеграція з платіжними системами або соціальними мережами для поширення.	Середній

Джерело: побудовано автором

Функції з високим пріоритетом є основою користувацького досвіду та базової функціональності. Функції із середнім пріоритетом можуть бути реалізовані на наступних етапах, але їх архітектурна підтримка має бути закладена. Цей набір вимог слугуватиме основою для подальшого проектування.

1.4 Визначення нефункціональних вимог (продуктивність, безпека, масштабованість)

Нефункціональні вимоги (НФВ) визначають атрибути якості, експлуатаційні характеристики та обмеження системи, що є критично важливими для її успіху. Вони охоплюють такі аспекти, як продуктивність, безпека, масштабованість, доступність, надійність та зручність підтримки. Формування НФВ базується на аналізі потреб спільноти, існуючих рішень та визначених функціональних вимог.

Ключові НФВ включають, перш за все, продуктивність. Система повинна забезпечувати мінімальний час відгуку для ключових API: для операцій читання – менше 200 мс для 95% запитів, а для операцій запису – менше 500 мс. Вона має обробляти щонайменше 1000 одночасних активних сесій (до 100-200 RPS) та виконувати пошук у великому каталозі за 1-2 секунди. Досягнення цих показників планується через оптимізацію запитів до PostgreSQL, ефективне кешування з Redis та використання асинхронної архітектури Node.js. Ризики невиконання цієї вимоги полягають у втраті користувачів та негативних відгуках.

Не менш важливою є масштабованість, що передбачає підтримку вертикального та, пріоритетно, горизонтального масштабування, а також здатність обробляти десятикратне збільшення навантаження без суттєвої переробки архітектури. Це буде реалізовано за допомогою stateless-сервісів, контейнеризації (Docker, Kubernetes), балансування навантаження та архітектурних рішень, що передбачають еволюцію до мікросервісів. Ігнорування цієї вимоги може призвести до деградації продуктивності та високих витрат.

Безпека є абсолютним пріоритетом. Вона включає комунікацію через HTTPS, автентифікацію через JWT, захист від OWASP Top 10 API Security Risks, надійне хешування паролів (bcrypt/Argon2), захист даних у PostgreSQL та відповідність GDPR. Методи досягнення включають використання перевірених бібліотек, валідацію вхідних даних та управління секретами. Нехтування безпекою може призвести до витоку даних та серйозних репутаційних і юридичних наслідків.

Система повинна мати високу доступність з цільовим рівнем 99,9%, що досягається через відмовостійкість, резервування та розгортання в декількох зонах доступності. Надійність забезпечується коректним виконанням функцій, збереженням даних без втрат, регулярним резервним копіюванням (RPO < 1 год, RTO < 4 год) та транзакційністю операцій.

Крім того, важливими є локалізація, що передбачає коректну обробку UTF-8 та гнучкі механізми для української мови; зручність підтримки через добре структурований код, документацію, автоматизоване тестування та моніторинг; та

інтероперабельність завдяки стандартизованому та версіонованому API. Ретельне визначення та дотримання цих НФВ є запорукою створення високоякісного та конкурентоспроможного застосунку.

Чіткий аналіз усіх КНВ наведено у табл. 1.3.

Таблиця 1.3

Аналіз ключових нефункціональних вимог

Вимога	Опис	Пріоритет
Продуктивність	Час відгуку API (читання) < 200 мс (95%), підтримка >1000 одночасних сесій (>100-200 RPS), пошук < 2 сек.	Високий
Масштабованість	Горизонтальне масштабування для підтримки >10 000 одночасних сесій, auto-scaling, масштабована архітектура БД.	Високий
Безпека	HTTPS (TLS 1.2+), JWT, захист від OWASP Top 10 API, хешування паролів (bcrypt/Argon2), відповідність GDPR, шифрування даних at rest (за потреби).	Високий
Доступність	Uptime системи 99,9% (максимальний простій < 8.77 годин/рік), відмовостійкість ключових компонентів.	Високий
Надійність	Регулярне резервне копіювання даних (RPO < 1 год, RTO < 4 год), транзакційність операцій, механізми відновлення після збоїв, моніторинг цілісності даних.	Високий
Локалізація (як НФВ)	Підтримка UTF-8, коректне сортування та форматування для української мови, гнучка архітектура для додавання нових мов, локалізовані системні повідомлення.	Високий

Продовження таблиці 1.3

Зручність підтримки	Модульний, документований код (SOLID, DRY), автоматизоване тестування (>80% покриття для крит. модулів), централізоване логування, інтеграція з системами моніторингу.	Середній
Інтероперабельність	Стандартизований API (JSON, RESTful/GraphQL), версіювання API для зворотної сумісності з мобільними клієнтами.	Середній

Джерело: побудовано автором

Ретельне визначення та дотримання цих НФВ на всіх етапах є запорукою створення високоякісного, надійного та конкурентоспроможного застосунку.

РОЗДІЛ 2
ПРОЕКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ МОБІЛЬНОГО ЗАСТОСУНКУ ДЛЯ СПІЛЬНОТИ КНИГОЛЮБІВ

2.1 Вибір технологічного стеку та розробка архітектури серверної частини

Проектування серверної частини є фундаментальним етапом, що закладає архітектурний кістяк, технологічну основу та ключові механізми функціонування мобільного застосунку. Для платформи, орієнтованої на активну взаємодію спільноти книголюбів, що передбачає обробку значних обсягів даних та реалізацію соціальних функцій, серверна інфраструктура має бути не лише функціональною, але й продуктивною, надійною, безпечною та, що критично важливо, масштабованою.

Формування технологічного стеку та розробка архітектури є взаємопов'язаними завданнями, що визначають довгострокову життєздатність та адаптивність застосунку. Вибір технологій ґрунтувався на комплексному аналізі вимог, серед яких ключовими були висока продуктивність обробки I/O-bound операцій (типових для API), забезпечення відмовостійкості, гарантування безпеки, зручність розробки, а також можливість ефективного горизонтального масштабування.

В якості основної платформи для реалізації серверної логіки було обрано Node.js. Його ключова перевага – асинхронна, керована подіями архітектура неблокуючого вводу-виводу (non-blocking I/O) – дозволяє ефективно обробляти велику кількість одночасних підключень з мінімальними накладними витратами. Це робить Node.js оптимальним для розробки API-сервісів та систем сповіщень, характерних для даної платформи. Додатково, уніфікація JavaScript для клієнтської та серверної розробки сприяє спрощенню процесу та підвищенню продуктивності команди [8].

Для побудови прикладного програмного інтерфейсу (API) у стилі REST на базі Node.js використовується веб-фреймворк Express.js. Його мінімалістичність, гнучкість та розширюваність надають надійний набір інструментів для визначення маршрутів, обробки HTTP-запитів та інтеграції проміжного програмного забезпечення (middleware) для реалізації наскрізної функціональності, такої як логування чи автентифікація. Велика екосистема доступних npm-пакетів дозволяє швидко адаптувати API під специфічні потреби [9].

З метою оптимізації процесу розробки базових операцій управління даними (CRUD) та забезпечення зручного адміністрування контенту, до стеку було інтегровано Strapi – headless Content Management System (CMS) на Node.js, інтерфейс зображено на рис. 2.1. Strapi дозволяє візуально моделювати структуру контенту та автоматично генерувати відповідні RESTful або GraphQL API ендпоінти, що суттєво скорочує час розробки адміністративної панелі та стандартного API. Це дає змогу команді зосередитись на реалізації унікальної бізнес-логіки. Крім того, Strapi надає вбудовані механізми для управління ролями та правами доступу [5].

Рисунок 2.1 - Графічний інтерфейс Strapi

Джерело: [5]

Для забезпечення гнучкості та можливості реалізації складної, нетривіальної логіки, яка виходить за межі стандартних можливостей Strapi (наприклад, алгоритми персоналізованих рекомендацій чи розрахунок комплексних рейтингів), застосовується комбінований підхід. Поряд з API, автоматично згенерованим Strapi для управління контентом, розробляються кастомні ендпоінти на Express.js. Обидві частини функціонують паралельно, звертаючись до єдиної бази даних PostgreSQL, що забезпечує синергію та дозволяє використовувати переваги обох інструментів.

Для зберігання всіх структурованих даних застосунку обрано PostgreSQL [7]. Ця потужна, об'єктно-реляційна СУБД з відкритим кодом відповідає принципам ACID, гарантуючи цілісність даних. Її перевагами є підтримка широкого спектру стандартів SQL, розширених типів даних (включаючи JSONB та масиви), повнотекстового пошуку та складних запитів, що є критичним для функціоналу книжкової платформи. Взаємодія з PostgreSQL з боку Strapi (v4+) здійснюється через конструктор запитів Knex.js, який також рекомендується використовувати і для кастомних модулів на Express.js для уніфікації, хоча можливе застосування й інших популярних ORM, як Sequelize чи TypeORM [11].

З метою суттєвого підвищення загальної продуктивності та зменшення часу відгуку, до стеку інтегрується Redis – високопродуктивне сховище даних типу "ключ-значення". Воно ефективно використовується для кешування результатів запитів до PostgreSQL, списків популярних книг, даних профілів, сесійних даних (наприклад, refresh-токенів) та лічильників, що дозволяє значно знизити навантаження на основну базу даних [10].

Для реалізації системи автентифікації та авторизації використовуються JSON Web Tokens (JWT). Цей відкритий стандарт (RFC 7519) забезпечує безпечний та, що важливо для масштабованості, безстатусний механізм обміну інформацією про ідентифікацію та права доступу між клієнтом та сервером [12].

Щодо архітектури, то на початковому етапі розробки вона реалізується як структурований моноліт з чітким розмежуванням модулів. Такий підхід прискорює розробку MVP та спрощує координацію. Однак, з огляду на потенційне зростання,

архітектура проектується з можливістю поступового переходу до мікросервісного підходу. Мікросервісна архітектура передбачає декомпозицію застосунку на набір невеликих, незалежно розгортаваних сервісів, кожен з яких відповідає за конкретну бізнес-функцію. Це забезпечує високу гнучкість, можливість незалежного масштабування компонентів та підвищує загальну відмовостійкість системи. Плановану архітектуру системи наведена на рис. 2.2. Взаємодія між мікросервісами може здійснюватися через HTTP/REST, gRPC або асинхронні черги повідомлень, а координація запитів – через API Gateway.

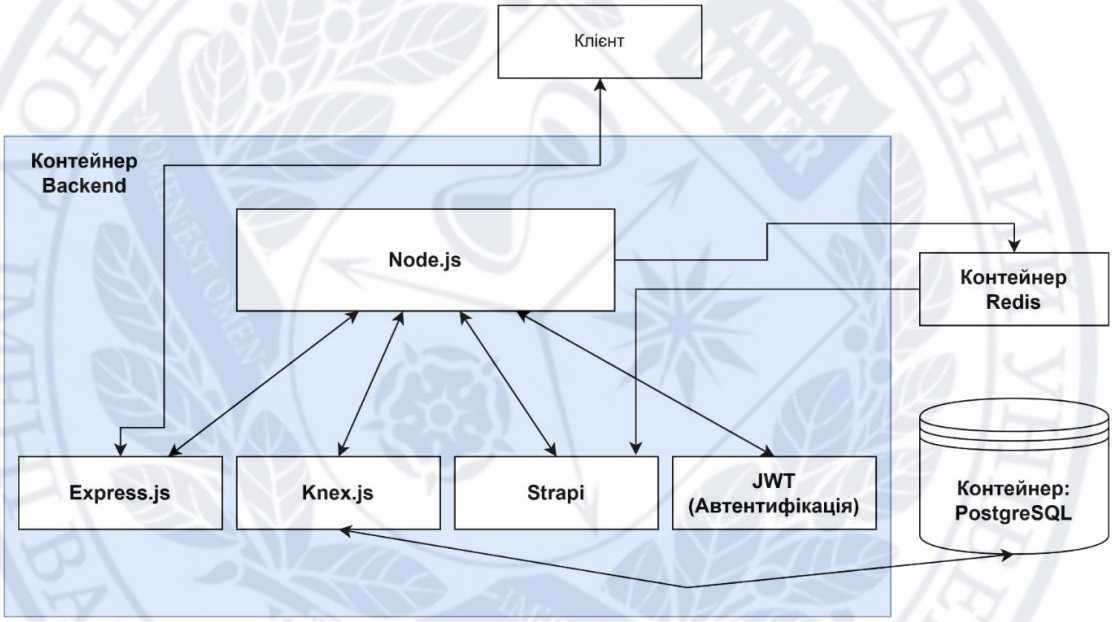


Рисунок 2.2 – Планована архітектура системи

Джерело: побудовано автором

Для повноцінної реалізації функціоналу та забезпечення ефективної взаємодії з користувачами, архітектура передбачає інтеграцію зі спеціалізованими сервісами та підходами. Зокрема, для інформування користувачів про важливі події – нові відгуки, оновлення в обговореннях чи персоналізовані рекомендації – буде використано сервіс Firebase Cloud Messaging (FCM) [25]. Це надійне та кросплатформенне рішення від Google забезпечує ефективну доставку push-повідомлень на мобільні пристрої Android та iOS. Крім того, для виконання

фонових, ресурсоємних або довготривалих завдань, таких як масові розсилки чи обробка зображень, планується інтеграція системи черг повідомлень (наприклад, RabbitMQ або Apache Kafka). Такий підхід до асинхронної обробки дозволяє розвантажити основні сервіси та підвищити чутливість системи, оскільки завдання виконуються фоново, не блокуючи основний потік обробки запитів.

Забезпечення здатності системи витримувати зростаючі навантаження та працювати стабільно є пріоритетом, що досягається через ряд інфраструктурних рішень. Важливим механізмом є горизонтальне масштабування, що передбачає можливість динамічного розгортання декількох ідентичних екземплярів серверного застосунку. Розподіл запитів між ними здійснюється за допомогою балансувальника навантаження (Nginx, HAProxy або AWS ELB), що забезпечує високу доступність.

Для уніфікації середовища розгортання та спрощення управління інфраструктурою використовується технологія контейнеризації Docker [6]. Інтерфейс Docker зображено на рис. 2.3. Вона дозволяє пакувати застосунки та їх залежності у стандартизовані контейнери. Оркестрація цих контейнерів за допомогою системи Kubernetes дозволяє автоматизувати процеси розгортання, масштабування (включаючи автоматичне), моніторингу та управління життєвим циклом, забезпечуючи високу доступність та ефективне використання ресурсів [17].

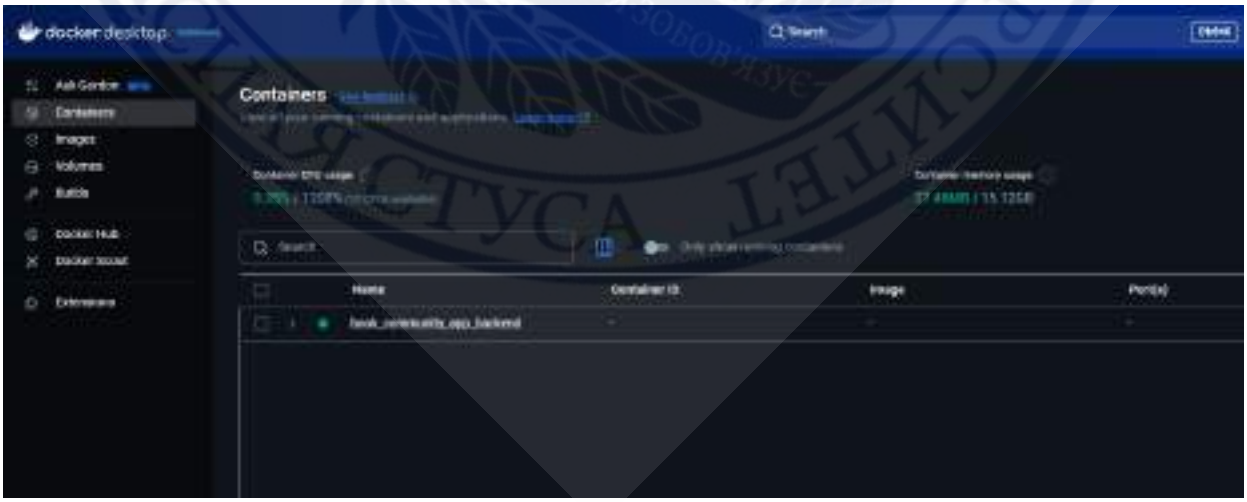


Рисунок 2.3 – Інтерфейс Docker

Джерело: [6]

Безпека розглядається як один із найвищих пріоритетів проектування. Багатошаровий підхід до захисту включає, насамперед, обов'язкове використання протоколу HTTPS (HTTP Secure) для шифрування всього трафіку між клієнтом та сервером, що унеможливлює перехоплення та модифікацію даних. Критично важливою є ретельна валідація всіх вхідних даних на серверній стороні для запобігання атакам типу SQL-ін'єкцій, XSS та інших. Для протидії DDoS-атакам та зловживанням API буде налаштовано обмеження частоти запитів (rate limiting). Крім того, стратегія безпеки передбачає проведення регулярних аудитів вихідного коду та конфігурації, а також своєчасне оновлення всіх програмних компонентів та їх залежностей.

Для забезпечення ефективного управління, діагностики проблем та аналізу продуктивності системи впроваджуються комплексні інструменти моніторингу та логування. Системи моніторингу, такі як Prometheus у поєднанні з Grafana [23], дозволять в реальному часі відстежувати ключові метрики продуктивності (навантаження на CPU, використання пам'яті, час відповіді API). Водночас, централізована система збору та аналізу логів (наприклад, стек ELK або альтернативи як Grafana Loki) забезпечить можливість консолідувати логи з усіх компонентів, що значно спростить аналіз подій та діагностику збоїв.

Узагальнюючи, обраний технологічний стек – Node.js, Express.js, Strapi, PostgreSQL, Redis – у поєднанні з продуманою архітектурою (моноліт з планом еволюції до мікросервісної), механізмами автентифікації на базі JWT, та засобами забезпечення масштабованості й безпеки, формує надійну, гнучку та сучасну основу для серверної частини мобільного застосунку. Ці рішення спрямовані на відповідність актуальним вимогам, забезпечуючи високу продуктивність, безпеку та можливість ефективного масштабування й подальшого розвитку функціоналу. Такий підхід дозволяє швидко адаптуватися до змін ринкових умов та потреб користувачів, мінімізуючи час на розробку нових можливостей. Використання популярних та добре документованих технологій також спрощує процес залучення нових розробників та підтримку кодової бази.

2.2. Моделювання реляційної бази даних на основі PostgreSQL для інформації про книги, користувачів та їх взаємодії

Ефективне моделювання бази даних є наріжним каменем у проектуванні будь-якої інформаційної системи, оскільки від структури даних безпосередньо залежать продуктивність запитів, цілісність інформації, можливості для аналітики та подальшого масштабування. Для мобільного застосунку спільноти книголюбів, що передбачає зберігання значних обсягів структурованих та взаємопов'язаних даних (інформація про користувачів, книги, авторів, їхні відгуки, списки читання, соціальні взаємодії), було прийнято рішення про використання реляційної моделі даних, реалізованої за допомогою системи управління базами даних (СУБД) PostgreSQL. Вибір PostgreSQL обґрунтований її загальновизнаною надійністю, відповідністю стандартам ACID (Atomicity, Consistency, Isolation, Durability), що гарантує транзакційну цілісність, а також багатим набором функціональних можливостей: розширені типи даних (зокрема, JSONB для гнучкого зберігання користувацьких налаштувань або додаткових атрибутів книг, масиви для тегів), потужні механізми для забезпечення цілісності даних (обмеження, тригери), підтримка складних SQL-запитів, повнотекстовий пошук та висока розширюваність. ER-діаграма БД наведена на рис. 2.4.

Для мобільного застосунку спільноти книголюбів, що оперує значними обсягами структурованих даних, було обрано реляційну модель та СУБД PostgreSQL. Цей вибір обґрунтований надійністю PostgreSQL, відповідністю стандартам ACID та багатим функціоналом, що включає розширені типи даних (JSONB, масиви), повнотекстовий пошук та високу розширюваність.

Проектування схеми бази даних ґрунтувалося на ретельному аналізі предметної області та потреб користувачів. Цей процес розпочався з ідентифікації ключових об'єктів системи – Користувачів, Книг, Авторів, Відгуків, списків читання – та їхніх атрибутів. Далі, для кожного атрибута було визначено оптимальний тип даних PostgreSQL, враховуючи природу інформації та вимоги до обробки: VARCHAR(n) для рядків, TEXT для описів, INTEGER або BIGINT для

чисел, TIMESTAMP WITH TIME ZONE для дат, BOOLEAN для логічних значень, JSONB для гнучких структур та TEXT[] для масивів тегів. Важливим етапом стало встановлення ключів: первинних (PK), зазвичай автоінкрементних SERIAL або BIGSERIAL для унікальної ідентифікації, та зовнішніх (FK) для забезпечення посилальної цілісності та логічних зв'язків між таблицями. Нарешті, було застосовано принципи нормалізації, прагнучи досягти щонайменше третьої нормальної форми (3NF), а за можливості – форми Бойса-Кодда (BCNF), з метою усунення надлишковості та запобігання аномаліям даних.

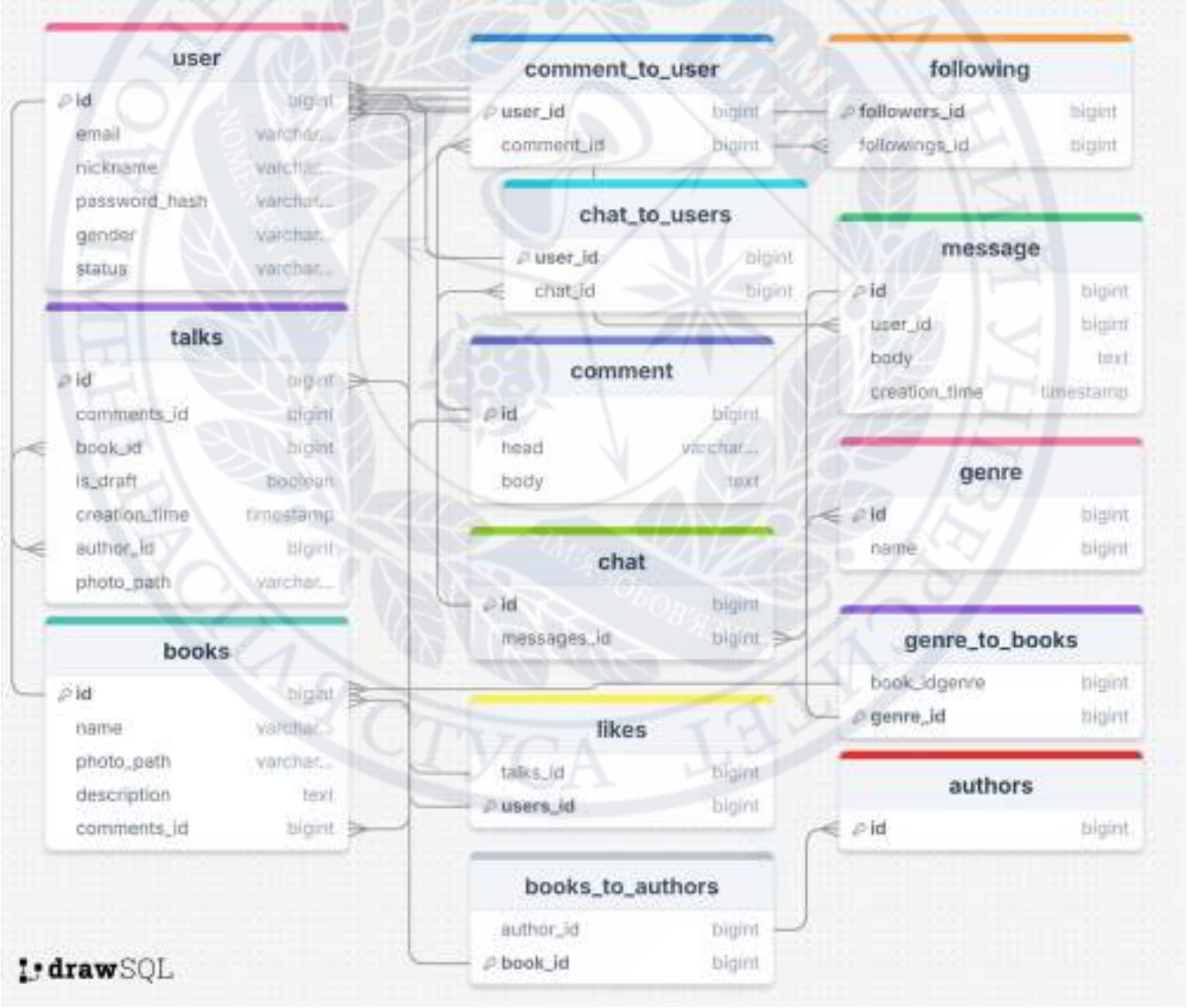


Рисунок 2.4 – ER-діаграма БД

Джерело: авторська розробка

Центральними інформаційними сутностями спроектованої бази даних є таблиці Users, Authors та Books. Таблиця Users зберігає інформацію про користувачів, їхні облікові дані, ролі та налаштування профілю. Authors містить дані про авторів книг. Таблиця Books є ядром каталогу, агрегуючи детальну бібліографічну інформацію, включаючи денормалізовані поля для оптимізації та масиви тегів. Важливу роль відіграє таблиця Reviews, де зберігаються текстові відгуки та числові рейтинги користувачів, пов'язані з конкретними книгами та користувачами, із забезпеченням унікальності (один відгук на книгу від одного користувача) та статусом модерації. Додатково, довідкові таблиці Publishers та Genres слугують для зберігання стандартизованої інформації про видавництва та жанри, що спрощує управління та унеможливлює дублювання.

Для реалізації зв'язків типу "багато-до-багатьох", які є невід'ємною частиною соціальної платформи, було введено проміжні таблиці. BookAuthors пов'язує книги та авторів, дозволяючи одній книзі мати кількох авторів, а одному автору – багато книг. UserBookLists реалізує функціонал персональних списків читання ("прочитані", "бажаю прочитати" тощо), зв'язуючи користувачів, книги та типи списків. Таблиця ReviewLikes відстежує вподобання користувачів щодо відгуків. Повна структура всіх атрибутів, їх типів, обмежень та зв'язків наочно представлена на ER-діаграмі.

Забезпечення функціональності, надійності та ефективності бази даних досягається через низку ключових аспектів. Цілісність даних гарантується на рівні СУБД завдяки використанню первинних та зовнішніх ключів з відповідними правилами, обмежень UNIQUE, NOT NULL, CHECK, а також завдяки транзакційності (принципи ACID). Для підвищення продуктивності застосовуються стратегії індексації: окрім автоматично створюваних індексів для PK та UNIQUE, додаткові B-tree індекси створюються на зовнішні ключі та поля, що часто використовуються у фільтрації та сортуванні. Для повнотекстового пошуку та роботи з масивами використовуються спеціалізовані GIN або GiST індекси. Підтримка локалізації, зокрема української мови, реалізована через додавання

окремих стовпців для локалізованих даних та використання кодування UTF-8. Щодо масштабованості, передбачено можливість вертикального масштабування, використання реплікації (read replicas) та декларативного партиціонування для великих таблиць. Нарешті, візуалізація та документування схеми за допомогою ER-діаграм сприяє кращому розумінню моделі даних та полегшує подальшу розробку.

Таким чином, ретельно продумана модель реляційної бази даних на PostgreSQL, що враховує специфіку предметної області та майбутні потреби, закладає надійний фундамент для стабільної, продуктивної та масштабованої серверної частини.

2.3. Проектування системи автентифікації, авторизації та механізмів забезпечення масштабованості

Проектування надійних систем автентифікації та авторизації, поряд з ефективними механізмами масштабування, становить невід'ємну та критично важливу складову розробки серверної частини будь-якого сучасного мобільного застосунку. Це особливо актуально для платформ, що передбачають активну соціальну взаємодію та обробку персональних даних користувачів. Саме ці компоненти відіграють ключову роль у забезпеченні належного захисту даних, контрольованого доступу до функціоналу та здатності системи ефективно справлятися зі зростаючими навантаженнями.

Центральним елементом системи автентифікації, яка відповідає за перевірку заявленої ідентичності користувача, було обрано JSON Web Tokens (JWT). Цей відкритий промисловий стандарт (RFC 7519) надає механізм для створення токенів доступу. JWT являє собою компактний, самодостатній рядок у форматі JSON, структурований з трьох частин: Заголовка, що визначає тип токена та алгоритм підпису (наприклад, HS256 або RS256); Корисного навантаження, яке містить твердження (claims), такі як ідентифікатор користувача (userId), його роль (role), а також метадані (час видачі iat, термін дії exp); та Підпису, що генерується на сервері за допомогою секретного ключа для гарантування цілісності та автентичності

токена. Ключовою перевагою JWT для даного проекту є його безстатусний характер (statelessness). Це усуває потребу зберігати інформацію про сесії на сервері, що значно спрощує горизонтальне масштабування та зменшує навантаження, що є особливо важливим для мобільних застосунків та потенційної мікросервісної архітектури.

Процес автентифікації з використанням JWT відбувається наступним чином. Спочатку користувач надає свої облікові дані, зазвичай електронну пошту та пароль, через клієнтський мобільний застосунок. Серверна частина отримує ці дані та проводить їх валідацію. Важливим аспектом безпеки є те, що паролі користувачів перед збереженням у базу даних PostgreSQL (таблиця Users) обов'язково хешуються за допомогою криптографічно стійких адаптивних алгоритмів, таких як bcrypt або Argon2. Ці алгоритми використовують унікальну "сіль" (salt) для кожного пароля та мають настроюваний фактор складності, що забезпечує надійний захист від атак перебору та використання райдужних таблиць [14].

У випадку успішної валідації, коли наданий пароль співпадає з хешем у базі даних, сервер генерує access-токен (JWT) і підписує його секретним ключем, відомим лише серверу. Цей згенерований JWT повертається клієнтському застосунку. Клієнт зберігає токен безпечним чином, наприклад, у захищеному сховищі пристрою. Надалі, для кожного запиту до захищених ресурсів API, клієнтський застосунок автоматично додає цей JWT до HTTP-заголовка Authorization, зазвичай використовуючи схему Bearer. Сервер, отримавши такий запит, спочатку перевіряє валідність підпису токена за допомогою свого секретного ключа, а також актуальність терміну його дії. У разі успішної перевірки, сервер витягує необхідну інформацію про користувача безпосередньо з корисного навантаження токена, уникаючи додаткових звернень до бази даних. Це не тільки підвищує швидкість обробки запитів, але й додатково розвантажує систему керування базами даних. Таким чином, кожен запит стає самодостатнім з точки зору автентифікації, що відповідає безстатусному підходу, процес автентифікації користувача з JWT зображено на рис.2.5.

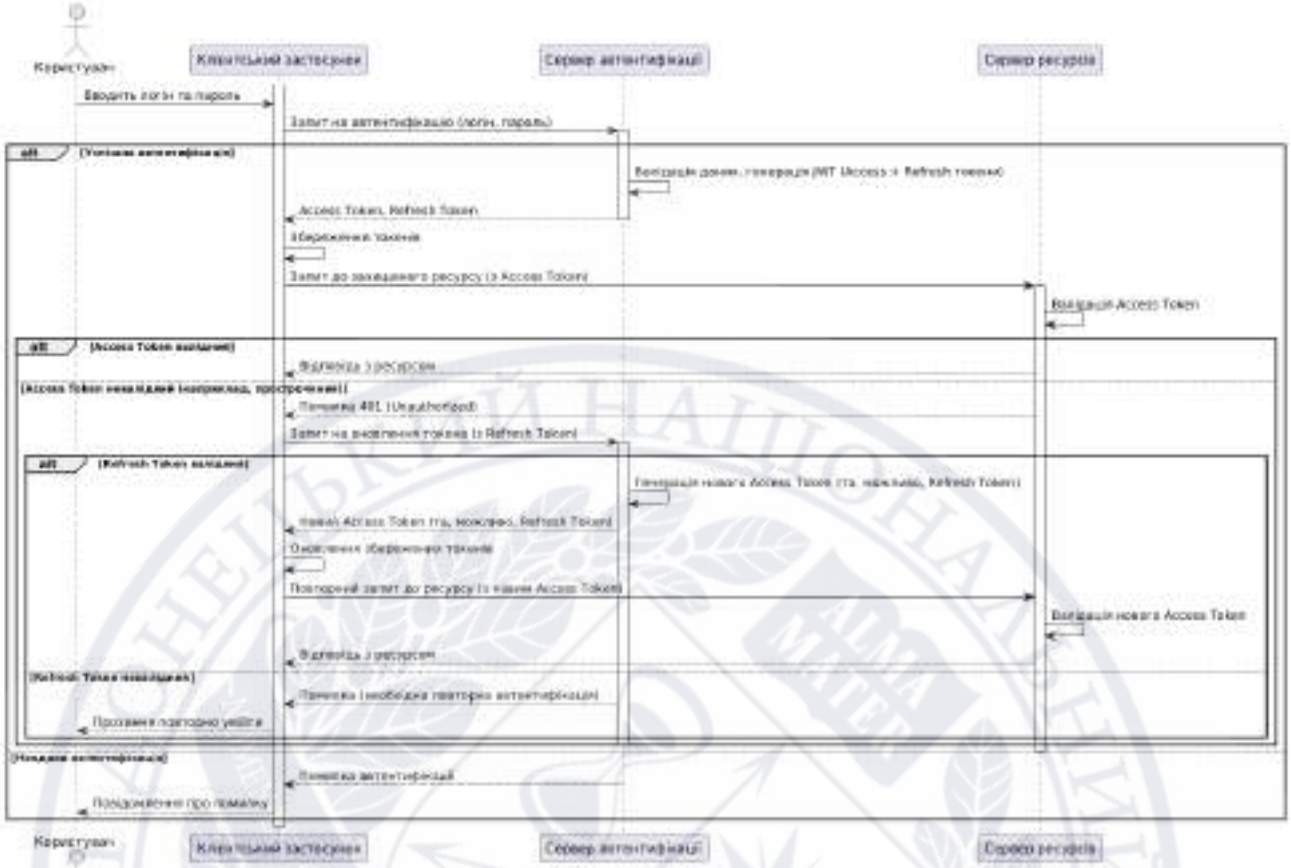


Рисунок 2.5 - Процес автентифікації користувача з JWT

Джерело: побудовано автором

Для забезпечення конфіденційності та цілісності даних, вся комунікація між клієнтським застосунком та сервером здійснюється виключно через захищений протокол HTTPS. Це гарантує шифрування даних під час передачі, надійно захищаючи як облікові дані, так і автентифікаційні токени від потенційного перехоплення.

З метою підвищення безпеки та покращення користувацького досвіду (UX), поряд із короткоживучими access-токенами (термін дії яких обмежений, наприклад, 15-60 хвилинами для мінімізації ризиків компрометації), впроваджується механізм refresh-токенів. Ці токени мають значно довший термін дії, що може сягати від 7 до 30 днів, і надійно зберігаються, переважно на серверній стороні (в Redis або PostgreSQL), асоціюючись із конкретним користувачем та сесією. Коли термін дії access-токена спливає, клієнт може використати refresh-токен для запиту нового

access-токена без необхідності повторного входу, що значно підвищує зручність. Важливо, що зберігання refresh-токенів на сервері дозволяє їх централізовано відкликати у разі потреби, наприклад, при зміні пароля або виявленні підозрілої активності.

Для ще більшої зручності користувачів, система передбачає можливість автентифікації через сторонніх провайдерів, таких як Google або Facebook, за допомогою протоколу OAuth 2.0. Буде реалізовано потік "Authorization Code Grant", що вважається найбільш безпечним для серверних застосунків [15]. Після успішної зовнішньої автентифікації, система створить або оновить локальний обліковий запис та згенерує власний JWT, уніфікуючи подальшу взаємодію з API.

Додатково, для захисту від атак перебору (Brute-Force) на ендпоінти автентифікації впроваджується механізм обмеження частоти запитів (rate limiting). Він обмежує кількість невдалих спроб входу з однієї IP-адреси або для одного акаунту, з можливим тимчасовим блокуванням. Надійне зберігання секретних ключів для підпису JWT є ще одним пріоритетом.

Після успішної автентифікації в дію вступає система авторизації, яка визначає, які дії та ресурси доступні конкретному користувачеві. В основі цієї системи лежить Рольова модель доступу (Role-Based Access Control - RBAC), що є поширеним та добре зрозумілим підходом [16].

Користувачам призначаються певні ролі, а кожній ролі – відповідний набір прав (permissions) на виконання операцій або доступ до ресурсів (рис. 2.6). Інформація про роль зберігається у профілі користувача в PostgreSQL і може включатися до JWT для швидкої перевірки.

У контексті застосунку визначено три ключові ролі. Базова роль – user (звичайний користувач) – надає право переглядати каталог, шукати книги, залишати власні відгуки та оцінки, керувати своїм профілем та особистими списками книг. Роль moderator успадковує всі права користувача, але додатково отримує інструменти для модерації контенту (відгуків, коментарів), забезпечуючи дотримання правил спільноти та якість контенту. Найвищий рівень доступу має

admin (адміністратор), який отримує повний контроль над системою: управління каталогом, користувачами, модерація всього контенту та доступ до адміністративної панелі для налаштування системи та перегляду статистики.

Рисунок 2.6 – Use-Case діаграма

Джерело: побудовано автором

Практична реалізація рольової моделі відбувається через механізм перевірки прав доступу при кожному запиті до захищених ресурсів API. Зазвичай, це реалізується через спеціалізоване проміжне програмне забезпечення (middleware) у фреймворку Express.js.

Таке middleware перехоплює запит, витягує JWT з заголовка Authorization, валідує його підпис та термін дії, а потім аналізує інформацію про роль користувача, що міститься в токені. На основі цієї ролі та попередньо визначених правил доступу для кожного ендпоінта та HTTP-методу (наприклад, право на виклик DELETE /api/v1/users/{id} може мати лише admin), система приймає рішення про надання або

заборону доступу. У випадку відмови, клієнту повертається відповідний HTTP-статус помилки, найчастіше 403 Forbidden.

Варто зазначити, що для ендпоінтів, керованих CMS Strapi, можуть застосовуватися її власні вбудовані інструменти управління ролями та правами. При цьому архітектура передбачає гнучкість: у майбутньому, за потреби, система авторизації може бути розширена до більш гранулярних моделей, таких як списки контролю доступу (ACL) або атрибутний контроль доступу (ABAC), якщо проста рольова модель виявиться недостатньою.

Поряд із забезпеченням безпеки та контролю доступу, ключовою нефункціональною вимогою є здатність системи ефективно обробляти зростаючу кількість одночасних користувачів, збільшення обсягу даних та інтенсивності запитів без суттєвої деградації продуктивності. Для досягнення цієї мети застосовуються різноманітні механізми забезпечення масштабованості.

Основним підходом є горизонтальне масштабування серверних застосунків, рис.2.7. Воно полягає у розгортанні декількох ідентичних екземплярів серверного застосунку (Node.js/Express.js та Strapi) на різних серверах або в окремих контейнерах. Завдяки безстатусності JWT-автентифікації, що була описана раніше, запити від одного й того ж користувача можуть ефективно оброблятися будь-яким доступним екземпляром сервера, оскільки кожен запит містить всю необхідну інформацію.

Розподіл вхідних запитів між цими екземплярами здійснюється за допомогою балансувальника навантаження – це може бути Nginx, HAProxy або відповідні хмарні сервіси, як-от AWS Elastic Load Balancer.

Такий підхід запобігає перевантаженню окремих вузлів, забезпечує високу доступність системи та дозволяє динамічно адаптувати обчислювальні потужності відповідно до поточних потреб.

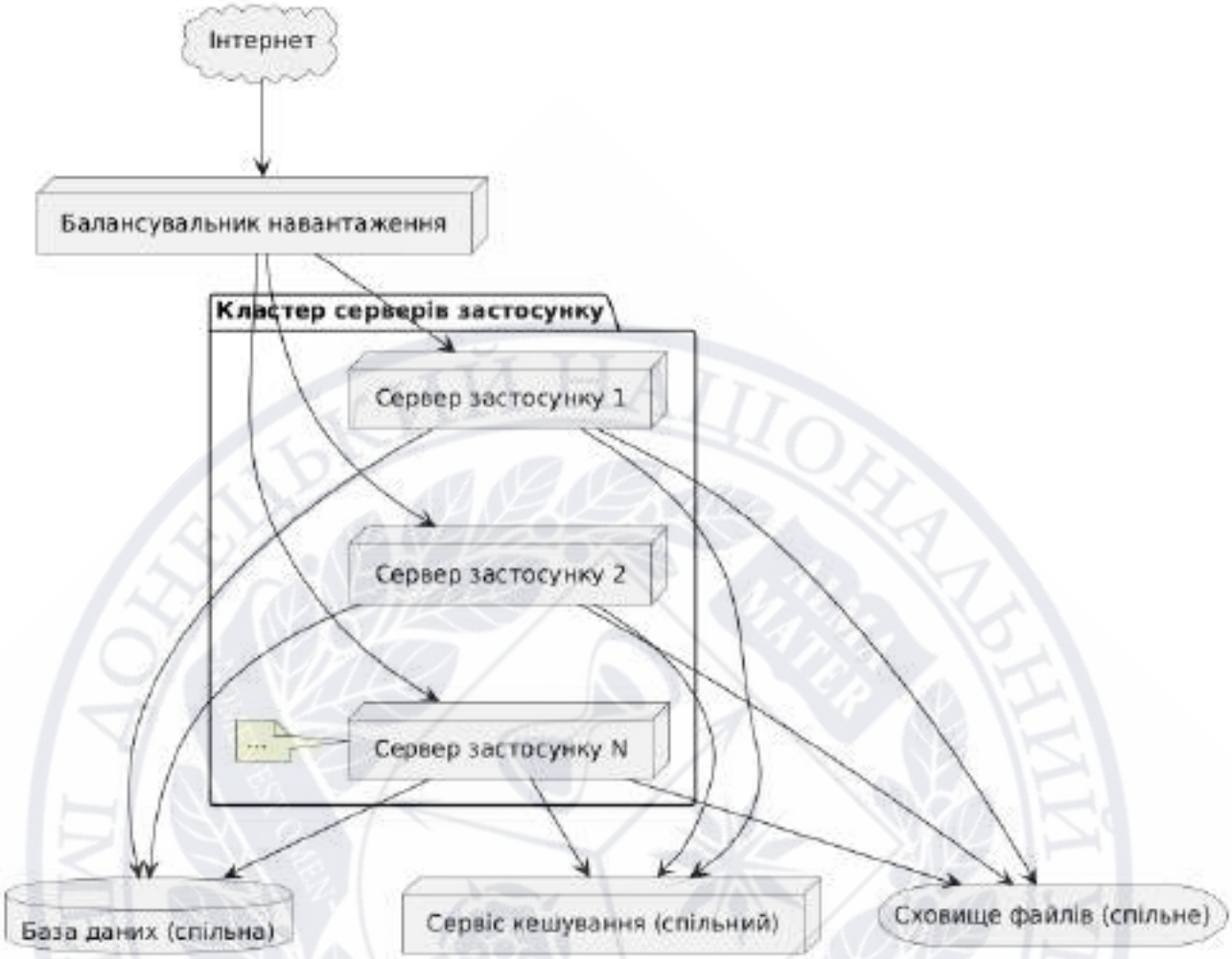


Рисунок 2.7 – Горизонтальне балансування з балансувальником навантаження

Джерело: побудовано автором

Довгострокова стратегія розвитку системи, як зазначалося раніше, передбачає еволюцію від початкової монолітної архітектури до мікросервісної. Це дозволить декомпозувати систему на набір невеликих, слабкозв'язаних та незалежно розгортаваних сервісів, кожен з яких відповідатиме за конкретну бізнес-функцію – управління користувачами, каталог книг, відгуки, рекомендації чи сповіщення. Такий підхід надає значні переваги: кожен мікросервіс може масштабуватися окремо відповідно до специфічного навантаження, що є більш ефективним з точки зору використання ресурсів. Навіть CMS Strapi може функціонувати як окремий мікросервіс. Водночас, важливо враховувати й виклики, пов'язані з мікросервісами, зокрема складність управління, забезпечення узгодженості даних та моніторингу.

Технологічною основою для гнучкого розгортання та управління як монолітом, так і майбутніми мікросервісами, є контейнеризація за допомогою Docker та оркестрація за допомогою Kubernetes.

Docker дозволяє пакувати застосунки та їх залежності у стандартизовані контейнери, забезпечуючи консистентність середовища та ізоляцію. Kubernetes, у свою чергу, автоматизує процеси розгортання, управління, моніторингу, самовідновлення та масштабування цих контейнерів. Зокрема, механізм Horizontal Pod Autoscaler (HPA) в Kubernetes дозволяє динамічно змінювати кількість запущених екземплярів сервісів залежно від поточного навантаження, наприклад, за показниками використання CPU або пам'яті [6].

Масштабованість на рівні зберігання даних забезпечується можливостями PostgreSQL. Це включає як вертикальне масштабування (збільшення ресурсів сервера БД), так і горизонтальне – через налаштування реплікації та створення ведених серверів (read replicas) для розподілу навантаження на операції читання. Для дуже великих таблиць, як-от Reviews, передбачається можливість застосування партиціонування за певним критерієм (наприклад, за датою) для покращення продуктивності та спрощення обслуговування.

Невід'ємним компонентом підвищення продуктивності та зменшення навантаження на базу даних є ефективне кешування даних за допомогою Redis. Ця розподілена система кешування в оперативній пам'яті використовується для зберігання часто запитуваних, але рідко змінюваних даних: списків популярних книг, даних профілів, конфігураційних параметрів або відповідей API.

Застосування стратегій кешування, таких як cache-aside, у поєднанні з механізмами інвалідації кешу при зміні даних у PostgreSQL, дозволяє значно зменшити кількість запитів до основної бази даних, скоротити час відповіді сервера та підвищити загальну пропускну здатність системи.

Для завдань, що не потребують негайної відповіді – ресурсоємних або довготривалих операцій (масові розсилки, генерація звітів, обробка файлів), – застосовується асинхронна обробка через черги повідомлень, такі як RabbitMQ або

Apache Kafka [26]. Завдання поміщаються в чергу і обробляються окремими фоновими процесами (воркерами). Це розвантажує основні потоки, покращує чутливість API та забезпечує надійне виконання фонових операцій.

Окрім механізмів автентифікації, авторизації та масштабування, проектування враховує додаткові заходи безпеки на рівні застосунку. Перш за все, це ретельна валідація всіх вхідних даних на сервері. Перевірка на відповідність формату, типу та діапазону значень є першою лінією захисту від SQL-ін'єкцій, XSS та інших атак, пов'язаних із введенням шкідливих даних.

Також застосовуються найкращі практики захисту API ендпоінтів згідно з рекомендаціями OWASP API Security Top 10 [13], включаючи уникнення передачі чутливих даних в URL та належну обробку HTTP-станів. Критично важливим є безпечне управління секретами: ключі JWT, паролі до БД та ключі доступу до сторонніх сервісів зберігаються за допомогою спеціалізованих інструментів (Kubernetes Secrets, HashiCorp Vault) або змінних середовища з обмеженим доступом.

Таким чином, застосований комплексний підхід, що охоплює сучасні стандарти безпеки, гнучку рольову модель, стратегії масштабування (горизонтальне, вертикальне, мікросервісне), контейнеризацію, оркестрацію, кешування та асинхронну обробку, формує надійну, безпечну та високопродуктивну основу для серверної частини.

Ці рішення спрямовані на забезпечення найвищого рівня захисту даних, контрольованого доступу та здатності системи стабільно адаптуватися до зростаючих навантажень, що є критичним для довгострокового успіху застосунку.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ

3.1. Налаштування середовища розробки та розгортання серверної частини

Після завершення етапів аналізу, формування вимог та детального проектування архітектури й бази даних, наступним логічним кроком є практична реалізація спроектованих рішень та їх подальше розгортання в операційне середовище. Цей розділ присвячений саме цим аспектам, охоплюючи ключові моменти налаштування середовища розробки, програмної реалізації основних модулів, а також стратегії та інструменти для розгортання, забезпечення безпеки, моніторингу та всебічного тестування системи.

Ретельне налаштування середовища розробки та визначення стратегії розгортання має фундаментальне значення для забезпечення стабільного, відтворюваного та ефективного процесу. Некоректно налаштоване середовище може призвести до розбіжностей між умовами розробки та продуктивним оточенням, ускладнюючи відлагодження. Описані підходи спрямовані на максимальне задоволення вимог до масштабованості, доступності, надійності та безпеки.

Основою для написання, модифікації та відлагодження коду слугує локальне середовище розробки (Local Development Environment). Ключовою вимогою до нього є максимальна відповідність продуктивному оточенню (Dev/Prod Parity), що мінімізує проблеми при перенесенні коду. Для цього використовується стандартизований набір компонентів та інструментів. Основою слугує актуальна LTS версія Node.js, управління якою може здійснюватися через NVM. Для роботи з проєктними залежностями застосовуються npm або Yarn, що забезпечують детерміновану збірку через package-lock.json або yarn.lock. Розробка ведеться з використанням Express.js для кастомної бізнес-логіки та Strapi для управління контентом. Локально встановлені екземпляри PostgreSQL та Redis дозволяють

тестувати взаємодію з даними та кешем, а управління міграціями схеми здійснюється через Knex.js. Як IDE переважно використовується Visual Studio Code з набором розширень для статичного аналізу, форматування та роботи з Docker, PostgreSQL та Redis.

Для забезпечення ізоляції залежностей та відтворюваності середовища застосовується контейнеризація за допомогою Docker та Docker Compose. Кожен сервіс або монолітний застосунок пакується у власний образ за допомогою Dockerfile, який містить інструкції для побудови, включаючи базовий образ, копіювання файлів, встановлення залежностей та команду запуску приклад Dockerfile наведено у лістингу 3.1. Для оптимізації використовуються multi-stage builds.

```
FROM node:18-alpine AS base
WORKDIR /usr/src/app
COPY package*.json ./
FROM base AS dependencies
RUN npm ci --only=production
FROM dependencies AS build
COPY . .
RUN npm install --only=development
# RUN npm run build
FROM base AS release
WORKDIR /usr/src/app
COPY --from=dependencies /usr/src/app/node_modules
./node_modules
COPY --from=build /usr/src/app/dist ./dist
COPY --from=build /usr/src/app/public ./public
COPY --from=build /usr/src/app/package.json ./
EXPOSE 3000
CMD [ "node", "dist/server.js" ]
```

Лістинг 3.1 – **приклад Dockerfile**

Управління мультиконтейнерними застосунками здійснюється декларативно через Docker Compose та файл docker-compose.yml. Цей файл описує сервіси (застосунок, PostgreSQL, Redis), їхні образи, залежності, налаштування мережі та томів (volumes) для забезпечення персистентності даних., docker-compose.yml наведено у лістингу 3.2.

```
services:
 app:
  build:
   context: .
   dockerfile: Dockerfile
  container_name: my_book_app
  ports:
   - "3000:3000"
  environment:
   - NODE_ENV=development
   -
DATABASE_URL=postgresql://app_user:app_password@db:5432/book_db
   - REDIS_URL=redis://cache:6379
  depends_on:
   - db
   - cache
  volumes:
   - .:/usr/src/app
   - /usr/src/app/node_modules
  restart: unless-stopped
 db:
  image: postgres:14-alpine
  container_name: my_postgres_db
  environment:
   POSTGRES_DB: book_db
   POSTGRES_USER: app_user
   POSTGRES_PASSWORD: app_password
  ports:
   - "5432:5432"
  volumes:
   - postgres_data:/var/lib/postgresql/data
  restart: unless-stopped
 cache:
  image: redis:7-alpine
  container_name: my_redis_cache
  ports:
   - "6379:6379"
  restart: unless-stopped
volumes:
 postgres_data:
```

Лістинг 3.2 – **приклад docker-compose.yml**

Docker Compose дозволяє запустити все середовище командою docker-compose up -d, результати виконання наведено на рис.3.1.

Рисунок 3.1 - Результати виконання docker-compose up -d

Джерело: побудовано автором

Для прискорення циклу розробки часто використовується монтування локальної директорії з кодом всередину контейнера, що забезпечує миттєве відображення змін без необхідності перебудови образу.

Невід'ємною частиною локальної розробки є автоматизоване тестування та базовий моніторинг. Для автоматизованого тестування застосовується фреймворк Jest, що дозволяє проводити юніт-тестування ізольованих частин коду [20]. Інтеграційне тестування API-ендпоінтів та їх взаємодії з базою даних і кешем реалізується за допомогою бібліотеки Supertest у поєднанні з Jest. Важливою практикою є налаштування збору метрик тестового покриття з цільовим показником понад 80% для критичних модулів. Щодо локального моніторингу та логування, можливе розгортання екземплярів Prometheus та Grafana [23] для аналізу метрик

продуктивності, а використання бібліотек Winston або Pino забезпечує структуруване логування, що полегшує діагностику.

Для розгортання, управління та масштабування Docker-контейнерів у продуктивному середовищі використовується керована служба Amazon Elastic Kubernetes Service (EKS). Вона тісно інтегрується з іншими сервісами AWS, такими як Elastic Load Balancing (ELB) для розподілу трафіку, IAM для управління доступом та VPC для мережевої ізоляції. Kubernetes забезпечує декларативне управління, автоматичне масштабування (HPA) та самовідновлення сервісів. Зберігання та версіонування Docker-образів здійснюється за допомогою приватного реєстру Amazon Elastic Container Registry (ECR), що інтегрований з EKS та CI/CD.

Інфраструктура також спирається на керовані сервіси AWS для баз даних та кешу. Amazon RDS для PostgreSQL надає керований екземпляр бази даних, беручи на себе адміністрування, резервне копіювання та забезпечуючи високу доступність. Amazon ElastiCache for Redis, у свою чергу, надає керований екземпляр Redis для розподіленого кешування, що покращує продуктивність.

Як єдина точка входу для клієнтських запитів може виступати Amazon API Gateway. Цей сервіс виконує маршрутизацію до EKS, автентифікацію і авторизацію, обмеження частоти запитів та захист через AWS WAF. Для безпечного зберігання та управління чутливими даними, такими як паролі до БД чи ключі API, використовуються сервіси AWS Secrets Manager або AWS Systems Manager Parameter Store.

Весь процес від внесення змін до коду до розгортання в продуктивне середовище автоматизується за допомогою комплексного CI/CD пайплайну. Для цього можуть використовуватися такі інструменти, як GitHub Actions, AWS CodePipeline, AWS CodeBuild та AWS CodeDeploy, що забезпечує швидкість, надійність та консистентність процесу розгортання, схему високорівневої архітектури розгортання на AWS наведено на рис. 3.2.

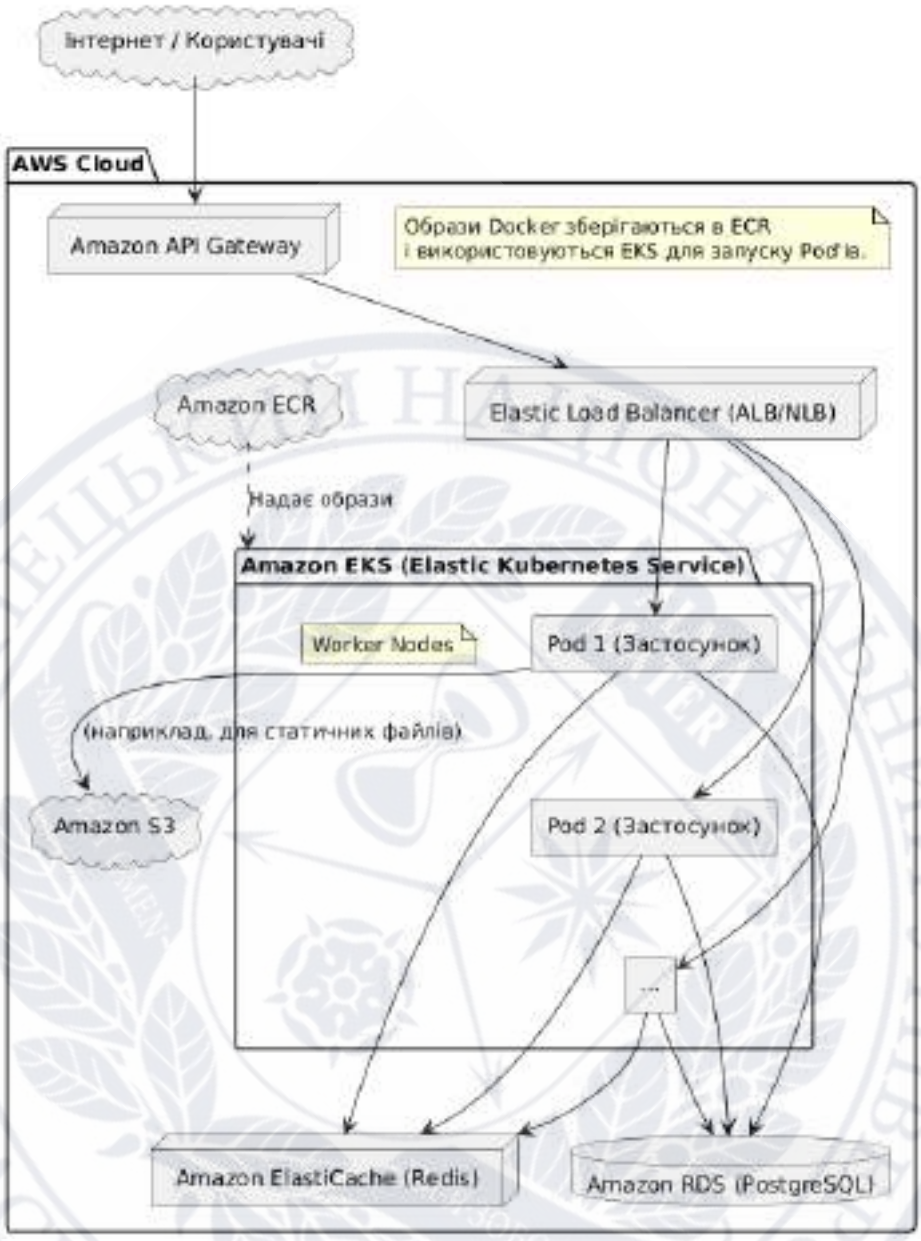


Рисунок 3.2 - Схему високорівневої архітектури розгортання на AWS

Джерело: побудовано автором

Типовий сценарій CI/CD: Розробник надсилає зміни до Git -> автоматичний запуск юніт/інтеграційних тестів -> у разі успіху, збірка Docker-образу -> тегування та завантаження образу в ECR -> ініціація оновлення конфігурацій Kubernetes (через Helm або Kustomize) для розгортання нової версії на кластері EKS. Застосовуються безпечні стратегії розгортання (Blue/Green, Canary releases) для мінімізації ризиків, CI/CD конвеєр наведено на рис. 3.3.

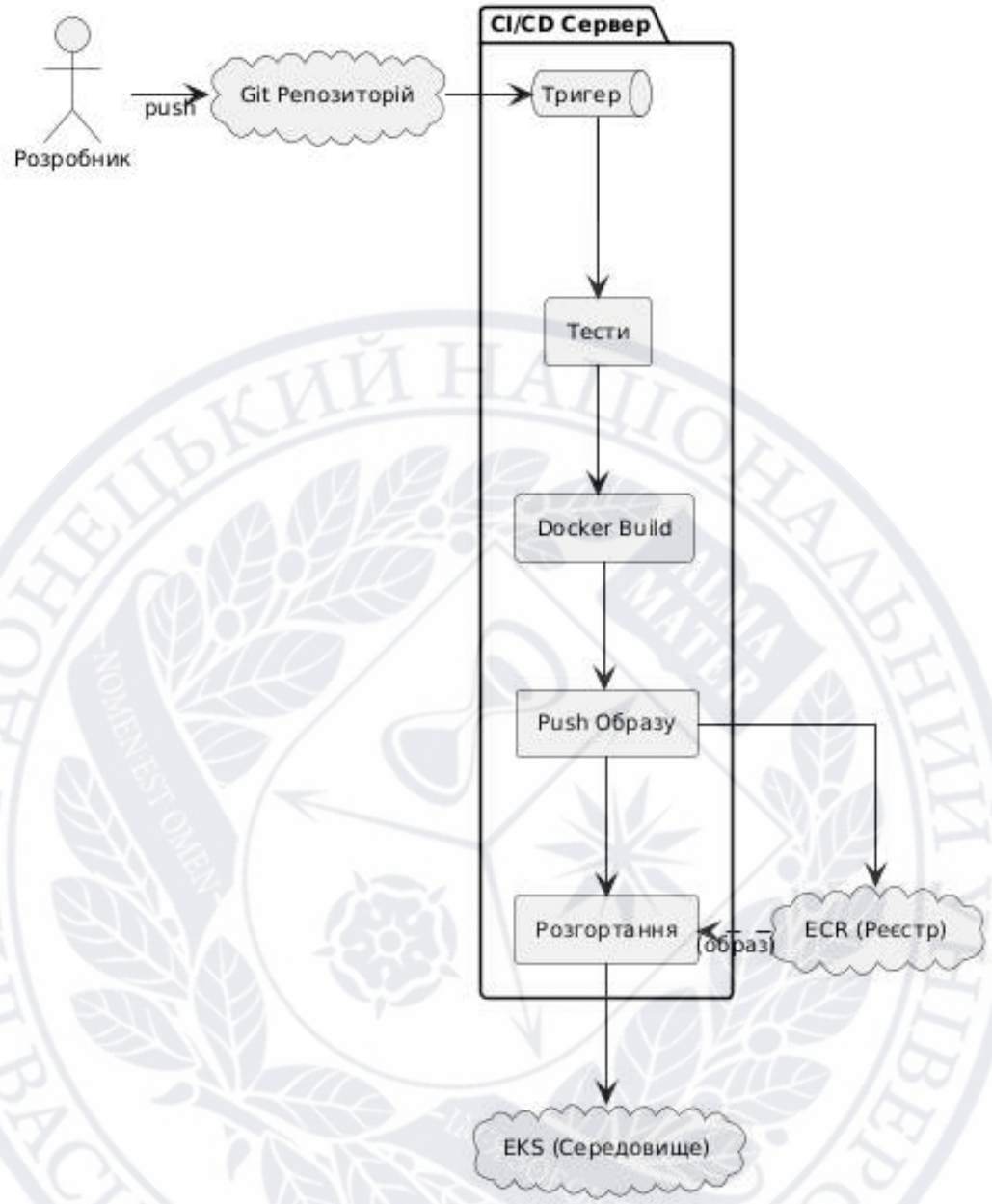


Рисунок 3.3 - CI/CD конвеєр

Джерело: побудовано автором

У продуктивному середовищі ключову роль відіграє комплексний моніторинг та логування. Основним інструментом в інфраструктурі AWS є Amazon CloudWatch, що надає сервіси для збору та аналізу логів (CloudWatch Logs), моніторингу метрик (CloudWatch Metrics) та налаштування автоматичних сповіщень (CloudWatch Alarms). Для більш глибокого збору та візуалізації

специфічних метрик застосунку можуть бути додатково розгорнуті Prometheus та Grafana, що інтегруються з експортерами метрик, як-от prom-client.

Хоча основний фокус зроблено на AWS, варто зазначити, що як альтернативна платформа для MVP або команд з обмеженими ресурсами може розглядатися Heroku (PaaS). Вона значно спрощує розгортання, але може мати обмеження гнучкості та вищу вартість при масштабуванні порівняно з AWS EKS.

Незалежно від обраної платформи, архітектура розгортання керується фундаментальними принципами, що забезпечують якість та надійність. Висока доступність (High Availability) досягається розгортанням компонентів у кількох ізольованих зонах доступності та використанням health checks. Це тісно пов'язано з відмовостійкістю (Fault Tolerance), тобто здатністю системи продовжувати роботу при збоях окремих компонентів завдяки резервуванню та автоматичному відновленню, як це реалізовано в Kubernetes. Масштабованість (Scalability) забезпечує гнучку адаптацію ресурсів до навантаження, бажано з автоматичними механізмами. Всі ці аспекти впроваджуються з дотриманням комплексних заходів безпеки (Security), що охоплюють мережевий рівень, шифрування даних та управління доступом. Нарешті, регулярне та надійне резервне копіювання даних з тестуванням процедур відновлення гарантує їх збереження.

Таким чином, продумане налаштування локального середовища з використанням Docker та Docker Compose, у поєднанні з обґрунтованим вибором хмарної платформи, такої як AWS EKS, та впровадженням CI/CD, забезпечують стандартизацію, якість, масштабованість, високу доступність та безпеку серверної частини на всіх етапах життєвого циклу.

3.2. Розробка API-ендпоінтів та інтеграція з базою даних

Розробка чітко визначених, функціонально насичених та ефективних API-ендпоінтів, разом з їх глибокою та оптимізованою інтеграцією з СУБД PostgreSQL, є центральним етапом практичної реалізації серверної частини. Цей процес трансформує функціональні вимоги – управління каталогом, профілями, відгуками

– у конкретні програмні інтерфейси, забезпечуючи при цьому дотримання нефункціональних вимог, таких як продуктивність, безпека та масштабованість. Далі описується методологія проектування RESTful API та ключові аспекти інтеграції. Незалежно від того, чи генеруються базові CRUD API за допомогою Strapi, чи розробляється кастомна логіка на Node.js/Express.js, результатом є набір RESTful API-ендпоінтів, що формують контракт взаємодії між сервером та клієнтом.

Проектування та реалізація API ґрунтується на дотриманні архітектурного стилю REST (Representational State Transfer), що забезпечує простоту, стандартизацію та масштабованість. Ключовим принципом є безстанність (Statelessness): кожен HTTP-запит від клієнта містить всю необхідну інформацію для обробки, включаючи JWT для автентифікації, а сервер не зберігає контекст клієнта між запитами, що значно спрощує горизонтальне масштабування. Застосовується ресурсно-орієнтований підхід, де функціональність та дані представлені як логічні ресурси (наприклад, /books, /authors) з унікальними URI. Взаємодія з ними відбувається через стандартні HTTP-методи (GET, POST, PUT, PATCH, DELETE), а результати операцій повідомляються через стандартні HTTP-коди стану (200 OK, 201 Created, 404 Not Found тощо). Основним форматом обміну даними обрано JSON через його легкість та широку підтримку. Для забезпечення плавної еволюції API без порушення роботи існуючих клієнтів впроваджується стратегія версіювання, наприклад, через включення номера версії до URI (/api/v1/books).

Для розробки ендпоінтів застосовується комбінований підхід. Strapi використовується для швидкої генерації значної частини базових CRUD-операцій над моделями даних, визначеними через адмінпанель. Це включає вбудовані можливості фільтрації, сортування та пагінації. Strapi також дозволяє кастомізувати ці ендпоінти та створювати нові через розробку власних контролерів та сервісів, що забезпечує гнучкість у керуванні контентом приклад вигляду наповненої БД у strapi наведено на рис. 3.4.

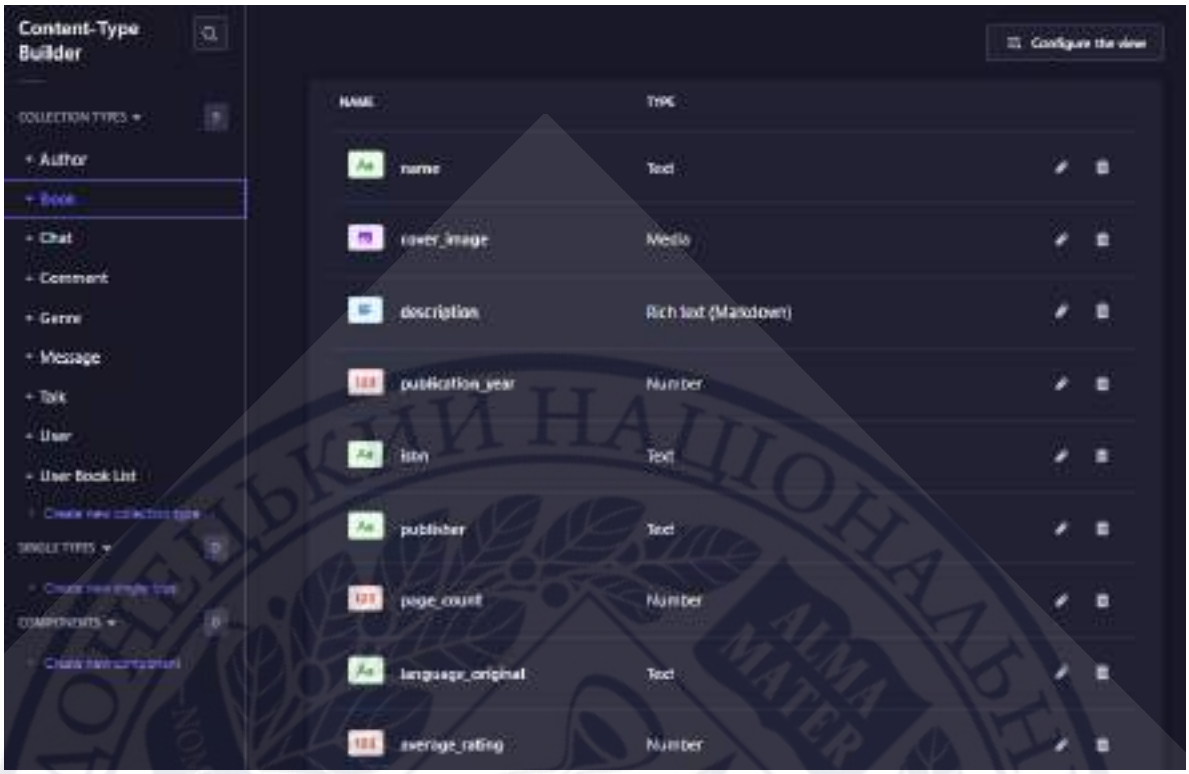


Рисунок 3.4 - Приклад вигляду наповненої БД у strapi

Джерело: побудовано автором на основі [5]

Для реалізації кастомної логіки та потенційних мікросервісів застосовується розробка на Node.js/Express.js. У цьому підході кожен модуль або мікросервіс реалізує власний набір ендпоінтів, дотримуючись структури, що включає маршрутизацію (використовуючи express.Router()), контролери (відповідальні за валідацію, виклик сервісів та формування відповідей), сервісний шар (що містить бізнес-логіку) та шар доступу до даних (DAL), реалізований через ORM. Middleware в Express.js активно використовується для наскрізних завдань: автентифікації, авторизації, логування та обробки помилок.

Важливим аспектом є документація API. Для цього використовується стандарт OpenAPI Specification (Swagger), що дозволяє ретельно документувати всі ендпоінти, їх параметри, відповіді, моделі даних та механізми безпеки. На основі цієї специфікації автоматично генерується інтерактивна HTML-документація (Swagger UI, ReDoc), що значно полегшує інтеграцію з клієнтськими застосунками та тестування.

Ефективна та безпечна інтеграція з СУБД PostgreSQL є критично важливою. Для взаємодії з базою даних з Node.js використовуються ORM (Object-Relational Mapper) або Query Builders. У проекті перевага надається Knex.js (який також використовується в Strapi v4+), що забезпечує гнучкий конструктор запитів, підтримку міграцій та "seed", а також захист від SQL-ін'єкцій. Альтернативно можуть застосовуватися ORM, як-от Sequelize чи TypeORM. Використання цих інструментів зменшує шаблонний код, підвищує продуктивність розробки та спрощує реалізацію складних запитів та транзакцій.

Для досягнення високої продуктивності API застосовуються різноманітні техніки оптимізації запитів до бази даних. По-перше, проводиться аналіз планів виконання запитів за допомогою EXPLAIN та EXPLAIN ANALYZE в PostgreSQL для виявлення повільних операцій. По-друге, ключовим є правильне проектування та використання індексів (B-tree, GIN, GiST) на полях, що використовуються у фільтрації, сортуванні та об'єднаннях. Важливо також уникати проблеми "N+1 запитів" шляхом застосування "жадібного завантаження" (eager loading) пов'язаних даних. Крім того, активно використовуються вбудовані можливості PostgreSQL для складних аналітичних запитів та повнотекстового пошуку. Для операцій, що вимагають атомарної зміни даних у декількох таблицях, обов'язково використовуються транзакції, що гарантує цілісність даних, а ORM/Knex.js надають зручний синтаксис для управління ними.

Ще одним кроком для зменшення навантаження на PostgreSQL та прискорення часу відповіді є інтеграція з системою кешування Redis. Найбільш поширеною стратегією кешування є Cache-Aside Pattern (Lazy Loading). Відповідно до неї, дані, що часто запитуються, але не змінюються надто динамічно (наприклад, списки популярних книг або деталі профілів), зберігаються в Redis, патерн зображено на рис. 3.5.

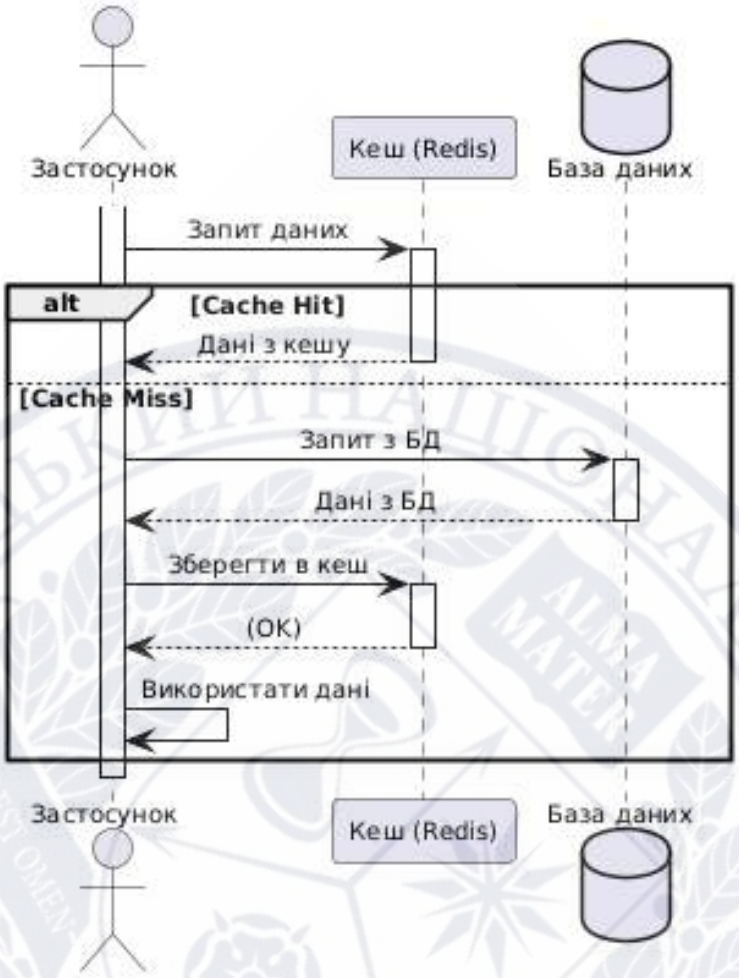


Рисунок 3.5 - Cache-Aside Pattern

Джерело: побудовано автором

Механізм роботи Cache-Aside полягає в тому, що перед запитом до PostgreSQL сервер перевіряє наявність даних у Redis за унікальним ключем. У випадку "cache hit", дані швидко повертаються з кешу. Якщо ж стався "cache miss", виконується запит до PostgreSQL, отримані дані зберігаються в Redis з попередньо визначеним часом життя (TTL), що залежить від частоти їх оновлення, і лише потім повертаються клієнту. При зміні даних у PostgreSQL вкрай важливо забезпечити своєчасну інвалідацію кешу (видалення або оновлення записів у Redis), щоб уникнути повернення застарілої інформації. Для цього можуть застосовуватися різні стратегії, як-от write-through, write-around або інвалідація за ключем. Для роботи з Redis використовуються різноманітні типи даних (Strings, Hashes, Lists,

Sets) та клієнтські бібліотеки для Node.js (redis, ioredis), що підтримують асинхронні команди.

Для операцій, що не вимагають негайної відповіді, впроваджується асинхронна обробка ресурсоємних завдань за допомогою черг повідомлень, наприклад, RabbitMQ. Застосовується архітектура "виробник-споживач": API-ендпоінт формує повідомлення із завданням та відправляє його до черги, швидко повертаючи клієнту відповідь (наприклад, 202 Accepted).

Окремі фонові процеси (воркери) асинхронно вибирають та виконують ці завдання. Це покращує продуктивність API, підвищує надійність фонових операцій та дозволяє незалежно масштабувати воркерів. Взаємодія з чергами з Node.js відбувається через спеціалізовані бібліотеки, як-от amqplib.

Для розширення функціональності та збагачення даних передбачено взаємодію із зовнішніми сервісами. Інтеграція з Google Books API (або OpenLibrary API) дозволяє автоматично отримувати детальну інформацію про книги. Серверна частина виконує HTTP-запити, обробляє та кешує отримані дані.

Для надсилання push-сповіщень на мобільні пристрої використовується Firebase Cloud Messaging (FCM) через Firebase Admin SDK, що дозволяє реалізувати цільові сповіщення.

На етапі реалізації особлива увага приділяється ключовим аспектам надійності та безпеки API. Кожен захищений ендпоінт проводить автентифікацію та авторизацію, перевіряючи валідність JWT та роль користувача. Здійснюється сувора валідація всіх вхідних даних від клієнта за допомогою бібліотек, як express-validator або joi, для запобігання помилкам та атакам.

Система проектується з урахуванням відповідності GDPR та іншим нормам приватності, надаючи API для реалізації прав користувачів. Також забезпечується локалізація на рівні API, що включає коректну обробку UTF-8 та повернення контенту українською мовою.

Отже, ретельна реалізація функціональних, безпечних та високопродуктивних API-ендпоінтів, інтегрованих з PostgreSQL, Redis, чергами

повідомлень та зовнішніми сервісами, створює надійний програмний інтерфейс, що є технологічною основою для успішного функціонування мобільного застосунку.

3.3. Впровадження безпеки (шифрування, захист від атак) та тестування (юніт, інтеграційні, навантажувальні)

Заключними, але критично важливими етапами реалізації серверної частини є комплексне впровадження багатошарових заходів безпеки та проведення ретельного, всебічного тестування. Ці етапи, інтегровані протягом усього життєвого циклу розробки, спрямовані на забезпечення максимального захисту даних користувачів, цілісності системи, стійкості до кіберзагроз, а також на валідацію функціональної коректності та відповідності нефункціональним вимогам.

Впровадження комплексних заходів безпеки базується на багатошаровому підході "defense in depth". Безпека реалізується на рівнях інфраструктури, застосунку та даних, охоплюючи захист інформації як під час зберігання ("at rest"), так і під час передачі ("in transit"), протидію поширеним атакам та відповідність нормативним вимогам, зокрема GDPR.

Одним із фундаментальних аспектів є шифрування даних (Data Encryption)**.** Особлива увага приділяється захисту паролів користувачів: вони ніколи не зберігаються у відкритому вигляді.

Перед збереженням у PostgreSQL паролі піддаються односторонньому криптографічному хешуванню з використанням стійкого та адаптивного алгоритму, такого як bcrypt, що дозволяє налаштовувати складність ("work factor").

До кожного пароля автоматично додається унікальна "сіль" (salt), яка зберігається разом із хешем, що значно ускладнює атаки за допомогою райдужних таблиць та перебору за словником [14], даний процес зображено у схемі на рис. 3.6.

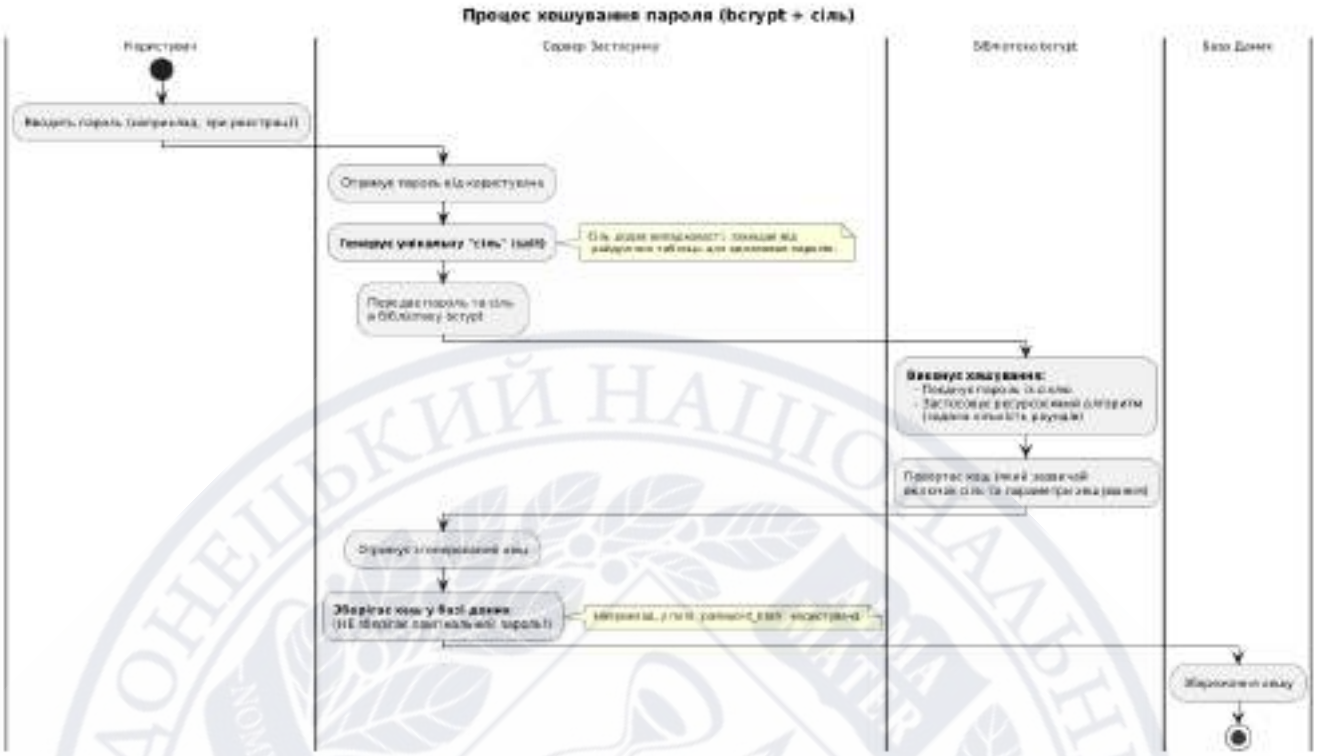


Рисунок 3.6 - Схему процесу хешування пароля

Джерело: побудовано автором

Важливим елементом є шифрування даних під час передачі (Encryption in Transit). Уся комунікація між клієнтом та API, а також між внутрішніми сервісами, здійснюється виключно за захищеним протоколом HTTPS (TLS 1.2/1.3), що забезпечує конфіденційність та цілісність за допомогою TLS-сертифікатів. Не менш важливим є захист конфігураційних даних та секретів (Secrets Management). Чутливі дані, такі як ключі JWT чи паролі до БД, ніколи не зберігаються у вихідному коді; натомість використовуються спеціалізовані сервіси (AWS Secrets Manager) або змінні середовища. Окремо розглядається безпека автентифікаційних токенів (JWT Security): access-токени мають короткий термін дії, підписуються стійкими алгоритмами, а refresh-токени зберігаються на сервері з можливістю їх відкликання.

Значна увага приділяється захисту від поширених веб-атак та загроз API, орієнтуючись на рекомендації OWASP Top 10. Для запобігання SQL-ін'єкціям використовуються ORM та параметризовані запити. Для запобігання XSS дані від

користувачів ретельно валідуються та санітизуються перед відображенням. Захист від CSRF для RESTful API з JWT є менш критичним, але для веб-інтерфейсів використовуються стандартні механізми. Крім того, налаштовуються HTTP-заголовки безпеки (через helmet) та політики CORS для обмеження доступу до API лише з довірених джерел. Невід'ємною частиною є забезпечення відповідності нормативним вимогам, зокрема GDPR, шляхом реалізації прав користувачів та дотримання принципів "privacy by design".

Додаткові важливі практики безпеки включають регулярне оновлення всіх залежностей та програмного забезпечення для усунення відомих вразливостей, а також надійне резервне копіювання даних PostgreSQL з регулярним тестуванням процедур відновлення. Планується проведення регулярних аудитів безпеки та тестування на проникнення. Нарешті, для гарантування якості та відповідності вимогам, застосовується всебічне та багаторівневе тестування серверної частини. Використовується багаторівневий підхід ("піраміда тестування") [19] для ефективного виявлення дефектів на якомога ранніх стадіях розробки, піраміду тестування відображено на рис. 3.7.

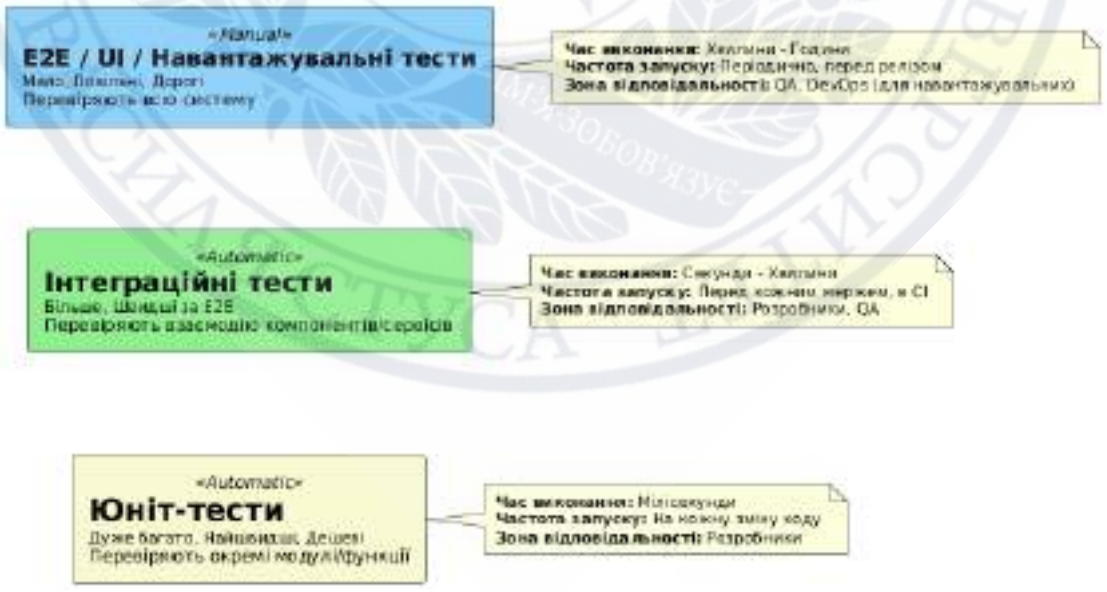


Рисунок 3.7 – Піраміда тестування

Джерело: побудовано автором

Першим рівнем "піраміди тестування" є юніт-тести (Unit Tests). Їхня мета полягає у перевірці коректності роботи найменших логічно ізольованих компонентів коду – функцій, модулів, методів класів – в повній ізоляції від зовнішніх залежностей, таких як база даних, файлова система чи мережеві сервіси. Для цього використовується фреймворк Jest [20, 21], а зовнішні залежності замінюються мок-об'єктами. Об'єктами тестування виступають функції валідації вхідних даних (наприклад, формату email чи діапазону рейтингу), компоненти бізнес-логіки (розрахунок середнього рейтингу, генерація/перевірка JWT), функції хешування паролів та різноманітні утиліти. Під час тестування перевіряються як позитивні, так і негативні сценарії, а також граничні випадки (edge cases). Значення юніт-тестів полягає у їх швидкому виконанні, що дозволяє виявляти помилки на ранніх етапах, спрощувати рефакторинг та забезпечувати стабільність кодової бази. Цільовим показником покриття коду юніт-тестами для критичних модулів визначено не менше 85%. Тестування API за допомогою фреймворку Jest показано на рис. 3.8.

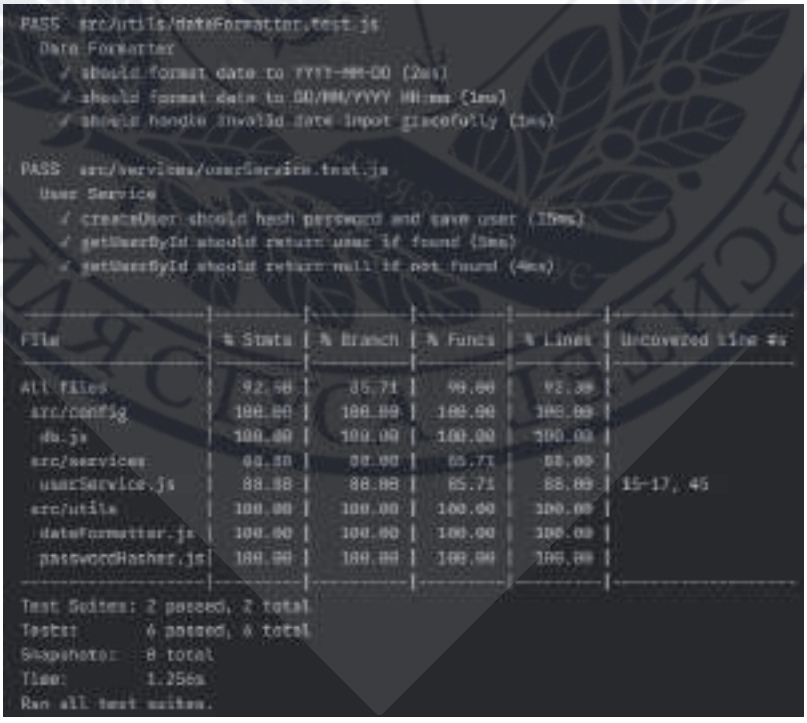


Рисунок 3.8 – Тестування API за допомогою фреймворку Jest

Джерело: побудовано автором

Наступним рівнем є інтеграційні тести (Integration Tests). Їхня мета полягає у перевірці коректності взаємодії між різними компонентами та модулями системи, що вже пройшли юніт-тестування. Це включає перевірку взаємодії API-ендпоінтів з базою даних PostgreSQL та кешем Redis, а також взаємодію між сервісами всередині застосунку або між мікросервісами.

Як інструмент використовується програмний інструментарій Postman, що дозволяє імітувати HTTP-запити до розроблених API-ендпоінтів та ретельно перевіряти їхні відповіді, включаючи статус-коди, тіло та заголовки [27]. Об'єктами тестування виступають ключові сценарії використання системи: від реєстрації користувача та його подальшої автентифікації до створення контенту (наприклад, книги адміністратором) та соціальних взаємодій (публікація відгуку).

Наприклад, при тестуванні POST-запиту на /api/v1/books/{bookId}/reviews перевіряється не лише валідація JWT та вхідних даних, але й коректність збереження відгуку в PostgreSQL, оновлення середнього рейтингу книги та інвалідація відповідних записів у кеші Redis.

Для виконання інтеграційних тестів використовується спеціально підготовлене тестове середовище, максимально наближене до продуктивного, часто з використанням Docker-контейнерів для PostgreSQL та Redis, з обов'язковим очищенням або заповненням бази даних тестовими даними перед кожним запуском. Такий підхід дозволяє виявити проблеми на стиках взаємодії компонентів, які неможливо відловити на рівні юніт-тестів.

Інтеграційні тести також підтверджують, що дані коректно передаються та трансформуються між різними частинами системи. Завдяки цьому значно підвищується впевненість у стабільності системи перед переходом до більш високорівневих видів тестування.Тестування API у програмному інструменті Postman показано на рис. 3.9.

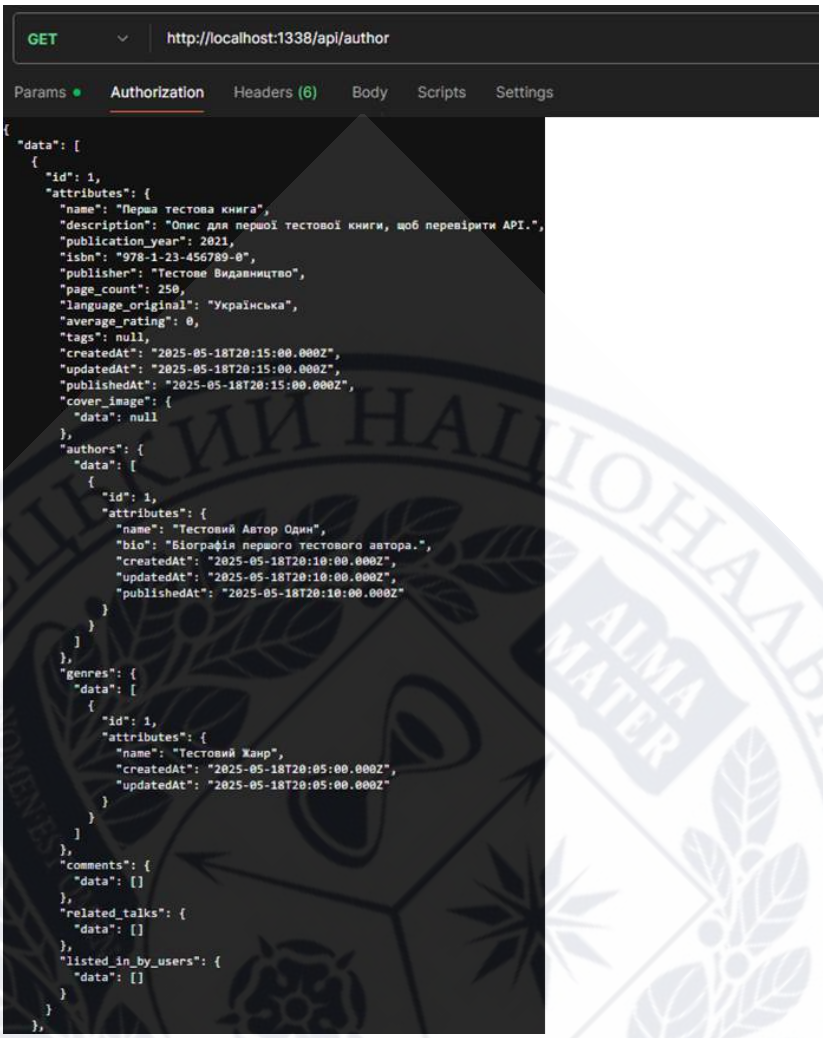


Рисунок 3.9 – Тестування API у програмному інструменті Postman

Джерело: побудовано автором

Вершиною "піраміди тестування" є навантажувальні тести (Load Tests) та тести продуктивності (Performance Tests). Їхня мета полягає в оцінці ключових нефункціональних атрибутів – продуктивності, стабільності, надійності та масштабованості серверної частини – під високим, реалістичним або навіть піковим навантаженням. Ці тести дозволяють визначити максимальну кількість одночасних користувачів або запитів за секунду (RPS), виміряти час відгуку та виявити потенційні "вузькі місця" в системі. Для їх проведення використовуються спеціалізовані інструменти, такі як Artillery.io, k6, Apache JMeter або Gatling [22,23]. Сценарії тестування ретельно розробляються для імітації поведінки великої кількості віртуальних користувачів, що виконують типові операції: перегляд книг,

пошук, створення відгуків чи автентифікацію. Застосовуються різні типи тестів: стандартні навантажувальні, стрес-тести (для визначення меж системи), тести на витривалість (soak/endurance) та на сплески (spike). Аналіз результатів базується на ключових метриках, що включають час відгуку API (середній, медіанний та перцентилі), пропускну здатність (RPS/TPS), відсоток помилок, а також показники використання ресурсів сервера, бази даних та кешу (CPU, RAM, I/O, мережа). Приклад навантажувального тестування представлено на рис. 3.10.

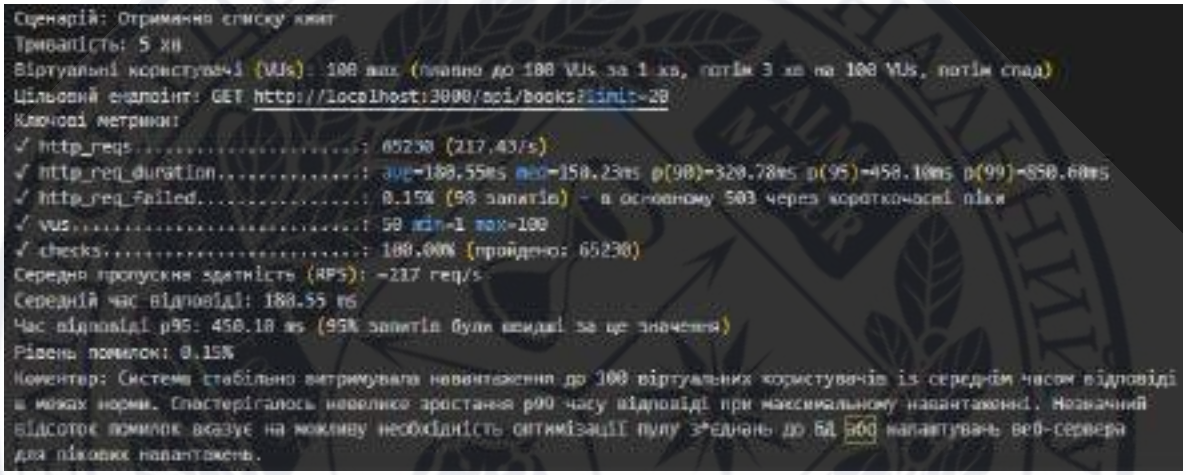


Рисунок 3.10 – Навантажувальне тестування

Джерело: побудовано автором

Результатом проведення таких тестів є виявлення "вузьких місць" (bottlenecks), як-от неефективний код, неоптимальні запити до бази даних чи неправильне кешування. Це також дозволяє перевірити ефективність механізмів масштабування, наприклад, auto-scaling в Kubernetes. На основі отриманих результатів та їх відповідності нефункціональним вимогам (наприклад, досягнення цільових показників RPS та часу відгуку) приймаються обґрунтовані рішення щодо оптимізації коду, запитів, кешування або масштабування інфраструктури.

Для забезпечення безперервності та надійності процесу контролю якості впроваджується автоматизація тестування. Усі автоматизовані тести, як юніт-, так і інтеграційні, інтегруються в CI/CD пайплайн (з використанням інструментів, як

GitHub Actions або AWS CodePipeline). Це забезпечує їх автоматичний запуск при кожній зміні коду, реалізуючи підхід "shift-left testing", та автоматичну генерацію звітів. Результати всіх видів тестування, включаючи навантажувальні, ретельно аналізуються та систематизуються. Для забезпечення прозорості контролю якості узагальнена інформація може бути представлена у структурованому вигляді, наприклад, у таблиці, що відображає тип тесту, кількість кейсів, покриття та виявлені проблеми.

Таким чином, впровадження надійних заходів безпеки та проведення всебічного, багаторівневого тестування є запорукою створення стабільного, захищеного, надійного та високопродуктивного серверного рішення, готового до інтеграції та довготривалої експлуатації.

ВИСНОВКИ

У ході виконання дипломної роботи було успішно досягнуто поставлену мету – здійснено комплексне проектування, обґрунтування вибору технологій та описано ключові аспекти реалізації та тестування серверної частини мобільного застосунку для спільноти книголюбів, з акцентом на високу продуктивність, масштабованість, безпеку та підтримку україномовного контенту.

Проведено аналіз предметної області, який дозволив зрозуміти специфічні потреби сучасної спільноти книголюбів, що включають легкий доступ до інформації про книги, розвинені соціальні функції, персоналізацію, підтримку україномовного контенту та можливість участі в тематичних подіях. Аналіз існуючих аналогів виявив їхні сильні та слабкі сторони, що дозволило сформулювати унікальну ціннісну пропозицію та уникнути повторення відомих недоліків, зокрема в частині локалізації для українського ринку. На основі цього аналізу було сформульовано детальний перелік функціональних та нефункціональних вимог, які стали основою для подальшого проектування.

Обґрунтовано вибір сучасного та ефективного технологічного стеку, ядром якого є Node.js у поєднанні з фреймворком Express.js для реалізації кастомної бізнес-логіки та RESTful API, та headless CMS Strapi для управління контентом та швидкої генерації базових CRUD-операцій. Для зберігання даних обрано потужну реляційну СУБД PostgreSQL, а для кешування – високопродуктивне сховище Redis. Розроблено гнучку архітектуру, що поєднує переваги швидкої розробки з Strapi та потужності кастомної реалізації на Express.js, із закладеною можливістю еволюції від початкового структурованого моноліту до мікросервісної архітектури для забезпечення довгострокової масштабованості.

Спроектовано детальну та нормалізовану реляційну модель бази даних на основі PostgreSQL, що враховує всі ключові сутності (користувачі, книги, автори, відгуки, списки читання, вподобання) та їхні взаємозв'язки, включаючи реалізацію зв'язків "багато-до-багатьох" через проміжні таблиці. Визначено типи даних,

первинні та зовнішні ключі, обмеження цілісності, стратегії індексації та механізми підтримки локалізації на рівні бази даних. Розглянуто аспекти масштабованості СУБД, включаючи реплікацію та партиціонування.

Розроблено надійну систему автентифікації та авторизації, що базується на використанні JSON Web Tokens (JWT) для безстатусної автентифікації та включає механізм refresh-токенів для підвищення безпеки та зручності. Передбачено можливість інтеграції з OAuth 2.0 провайдерами. Система авторизації реалізована на основі рольової моделі доступу (RBAC) з чітко визначеними ролями (користувач, модератор, адміністратор) та їхніми правами.

Описано ключові аспекти реалізації та розгортання серверної частини. Детально розглянуто налаштування локального середовища розробки з використанням Docker та Docker Compose для забезпечення консистентності та відтворюваності. Представлено комплексну архітектуру розгортання на хмарній платформі AWS, що включає Amazon EKS для оркестрації контейнерів, Amazon RDS для PostgreSQL, Amazon ElastiCache для Redis, Amazon API Gateway, а також інструменти CI/CD (GitHub Actions, AWS CodePipeline) та моніторингу (CloudWatch, Prometheus, Grafana).

Визначено комплексний підхід до розробки API-ендпоінтів, що дотримується принципів REST, та їх інтеграції з базою даних PostgreSQL, системою кешування Redis, чергами повідомлень (RabbitMQ/Kafka) для асинхронних завдань та зовнішніми API (Google Books API, FCM).

Запропоновано та обґрунтовано багатошарову стратегію забезпечення безпеки, що охоплює шифрування даних (паролів, даних "в дорозі" та "в спокої", секретів), захист від поширених атак (OWASP Top 10), відповідність GDPR, регулярне оновлення ПЗ, резервне копіювання та моніторинг безпеки. Розроблено детальну методологію тестування, що включає юніт-тести (Jest), інтеграційні тести (Jest, Supertest) та навантажувальні тести (Artillery.io, k6), з визначенням цільових показників покриття та продуктивності.

Практична цінність виконаної роботи полягає у створенні детального

проекту серверної частини, готового до реалізації, який може слугувати основою для розробки повноцінного мобільного застосунку для української спільноти книголюбів. Запропоновані архітектурні та технологічні рішення забезпечують необхідний баланс між швидкістю розробки, функціональністю, продуктивністю, безпекою, масштабованістю та можливістю подальшого розвитку проекту.

Напрямками подальших досліджень та розробок можуть бути: безпосередня програмна реалізація всіх описаних компонентів серверної частини, розробка клієнтської частини мобільного застосунку, поглиблене дослідження та впровадження більш складних алгоритмів для системи персоналізованих рекомендацій, розширення функціоналу для організації онлайн-подій та інтерактивних читацьких клубів, а також проведення масштабного користувацького тестування та збору зворотного зв'язку для подальшого вдосконалення платформи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Goodreads. *Goodreads*. URL: https://www.goodreads.com/ (date of access: 9.02.2025).
2. Bookly - особистий досвід користування застосунком 🕰️. *Друкарня*. URL: https://drukarnia.com.ua/articles/bookly-osobistii-dosvid-koristuvannya-zastosunkom-c913M (дата звернення: 18.02.2025).
3. Wattpad - Where stories live. *Wattpad - Where stories live*. URL: https://www.wattpad.com/ (date of access: 18.02.2025).
4. StoryGraph - . StoryGraph – Your own paragraph and favorite story. URL: https://www.storygraph.com/ (date of access: 18.02.2025).
5. Strapi 5 Docs | Strapi 5 Documentation. *Strapi 5 Docs | Strapi 5 Documentation*. URL: https://docs.strapi.io/ (date of access: 19.02.2025).
6. Home. *Docker Documentation*. URL: https://docs.docker.com/ (date of access: 19.02.2025).
7. PostgreSQL: Documentation. *PostgreSQL: The world's most advanced open source database*. URL: https://www.postgresql.org/docs/ (date of access: 19.02.2025).
8. Node.js – Run JavaScript Everywhere. *Node.js – Run JavaScript Everywhere*. URL: https://nodejs.org/ (date of access: 19.02.2025).
9. Express - Node.js web application framework. *Express - Node.js web application framework*. URL: https://expressjs.com/ (date of access: 20.02.2025).
10. Redis. *Docs*. URL: https://redis.io/docs/latest/ (date of access: 28.02.2025).
11. SQL Query Builder for Javascript | Knex.js. *SQL Query Builder for Javascript | Knex.js*. URL: https://knexjs.org/ (date of access: 28.02.2025).
12. RFC 7519: JSON Web Token (JWT). *IETF Datatracker*. URL: https://datatracker.ietf.org/doc/html/rfc7519 (date of access: 28.02.2025).
13. OWASP API Security Project | OWASP Foundation. *OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation*. URL: https://owasp.org/www-project-api-security/ (date of access: 8.03.2025).

14. Password Storage - OWASP Cheat Sheet Series. *Introduction - OWASP Cheat Sheet Series*. URL:https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (date of access: 8.03.2025).

15. RFC 6749: The OAuth 2.0 Authorization Framework. *IETF Datatracker*. URL: https://datatracker.ietf.org/doc/html/rfc6749 (date of access: 8.03.2025).

16. Role Based Access Control | CSRC. *NIST Computer Security Resource Center | CSRC*. URL: https://csrc.nist.gov/Projects/role-based-access-control (date of access: 18.03.2025).

17. Kubernetes Documentation. *Kubernetes*. URL: https://kubernetes.io/docs/home/ (date of access: 18.03.2025).

18. Початок роботи з Amazon EKS. Amazon Web Services. *Amazon Web Services, Inc.* URL: https://aws.amazon.com/en/eks/getting-started/ (дата звернення: 24.03.2025).

19. The Practical Test Pyramid. *martinfowler.com*. URL: https://martinfowler.com/articles/practical-test-pyramid.html (date of access: 28.03.2025).

20. Jest. *Jest Delightful JavaScript Testing*. URL: https://jestjs.io (date of access: 28.03.2025).

21. GitHub - ladjs/supertest: 🕷 Super-agent driven library for testing node.js HTTP servers using a fluent API. Maintained for @forwardemail, @ladjs, @spamscanner, @breejs, @cabinjs, and @lassjs. *GitHub*. URL: https://github.com/ladjs/supertest (date of access: 6.04.2025).

22. Welcome – Artillery Docs. *Artillery Docs*. URL: https://www.artillery.io/docs (date of access: 6.04.2025).

23. Grafana k6 | Grafana k6 documentation. *Grafana Labs*. URL: https://grafana.com/docs/k6/latest/ (date of access: 11.04.2025).

24. Using the API | Google Books APIs | Google for Developers. *Google for Developers*. URL: https://developers.google.com/books/docs/v1/using?hl=ua (date of access: 25.04.2025).

25. Firebase Cloud Messaging. *Firebase*. URL: https://firebase.google.com/docs/cloud-messaging?hl=ru (date of access: 28.04.2025).

26. RabbitMQ Tutorials | RabbitMQ. *RabbitMQ: One broker to queue them all | RabbitMQ*. URL: https://www.rabbitmq.com/tutorials (date of access: 14.05.2025).

27. Postman: The World's Leading API Platform | Sign Up for Free. *Postman API Platform*. URL: https://www.postman.com/ (date of access: 14.05.2025).

ДОДАТКИ

ДОДАТОК А - Фрагмент коду з прикладом запиту до БД через Knex.js

```
// file: services/bookService.js
import knexInstance from '../config/knexConfig.js'; // Налаштований
екземпляр Knex

async function getBookWithDetails(bookId, includeAuthor = false,
reviewsLimit = 5) {
 try {
  const bookQuery = knexInstance('books')
   .select(
    'books.id as book_id',
    'books.title',
    'books.publication_year',
    'books.isbn',
    'books.description',
    'books.page_count',
    'books.cover_image_url',
    'books.average_rating', // Денормалізоване поле
    'publishers.name as publisher_name' // Приклад JOIN з видавцями
   )
   .leftJoin('publishers', 'books.publisher_id', 'publishers.id')
   .where('books.id', bookId)
   .first(); // Очікуємо один запис

  const book = await bookQuery;

  if (!book) {
   return null; // Книгу не знайдено
  }

  const result = { ...book };

  if (includeAuthor) {
   const authors = await knexInstance('authors')
    .select('authors.id as author_id', 'authors.name', 'authors.bio')
    .join('book_authors', 'authors.id', 'book_authors.author_id')
    .where('book_authors.book_id', bookId);
   result.authors = authors; // Книга може мати декількох авторів
  }

  // Отримання відгуків з лімітом
  const reviewsQuery = knexInstance('reviews')
   .select(
     'reviews.id as review_id',
     'reviews.rating',
     'reviews.review_text',
     'reviews.created_at',
     'users.username as reviewer_username' // Ім'я користувача, що
```

```
залишив відгук
    )
    .join('users', 'reviews.user_id', 'users.id')
    .where('reviews.book_id', bookId)
    .orderBy('reviews.created_at', 'desc')
    .limit(reviewsLimit);

  result.reviews = await reviewsQuery;

  // Отримання загальної кількості відгуків для пагінації (якщо
потрібно)
  const totalReviewsCount = await knexInstance('reviews')
    .where('book_id', bookId)
    .count('id as total')
    .first();
  result.reviews_meta = {
    limit: reviewsLimit,
    total: totalReviewsCount ? parseInt(totalReviewsCount.total, 10)
: 0
  };


  return result;
 } catch (error) {
  console.error('Error fetching book with details:', error);
  throw new Error('Could not retrieve book details.');
 }
}

// Приклад виклику:
    // getBookWithDetails('some-book-uuid', true, 3).then(data =>
                        console.log(data));
```

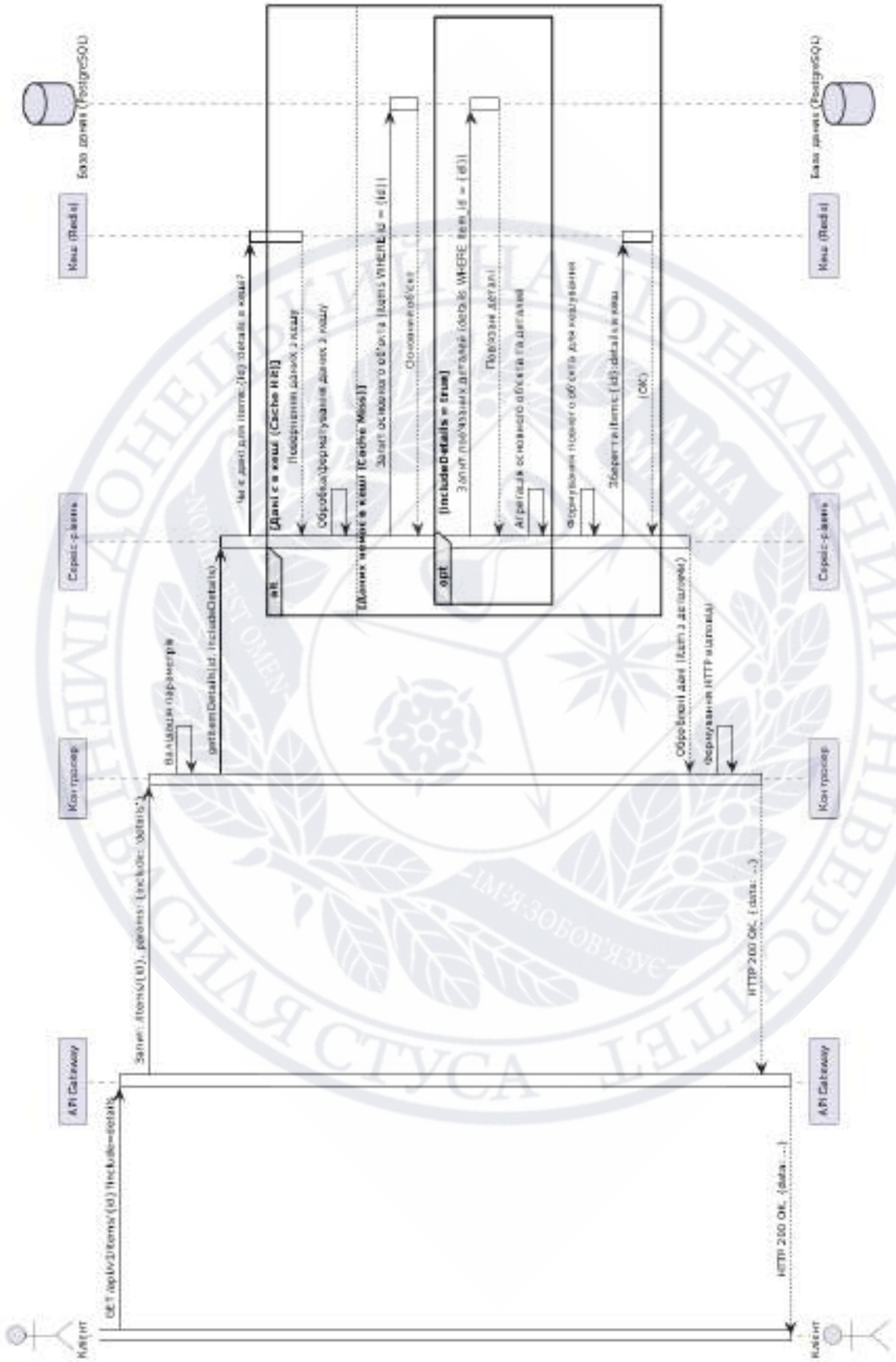
ДОДАТОК Б - Фрагмент коду з визначенням маршруту в Express.js

```
// file: routes/bookRoutes.js
import express from 'express';
import bookController from '../controllers/bookController.js';
import { authenticateToken } from '../middleware/authMiddleware.js';
// Приклад middleware автентифікації
import { validateBookQuery } from
'../middleware/validationMiddleware.js'; // Приклад middleware
валідації

const router = express.Router();

/**
 * @swagger
 * /books:
 * get:
 * summary: Отримати список книг
 * description: Повертає список книг з можливістю фільтрації та
пагінації.
 * parameters:
 * - in: query
 * name: page
 * schema:
 * type: integer
 * description: Номер сторінки
 * - in: query
 * name: limit
 * schema:
 * type: integer
 * description: Кількість елементів на сторінці
 * - in: query
 * name: genre
 * schema:
 * type: string
 * description: Фільтр за ID жанру
 * responses:
 * 200:
 * description: Список книг.
 * content:
 * application/json:
 * schema:
 * type: object
 * properties:
 * data:
 * type: array
 * items:
 * $ref: '#/components/schemas/Book' # Посилання на схему Book в
Swagger
 * pagination:
 * type: object
```

```
 * properties:
 * currentPage:
 * type: integer
 * totalPages:
 * type: integer
 * totalItems:
 * type: integer
 * tags:
 * - Books
 */
router.get('/', validateBookQuery, bookController.getAllBooks);

/**
 * @swagger
 * /books/{id}:
 * get:
 * summary: Отримати книгу за ID
 * description: Повертає детальну інформацію про книгу, включаючи
автора та відгуки.
 * parameters:
 * - in: path
 * name: id
 * required: true
 * schema:
 * type: string
 * description: ID книги
 * - in: query
 * name: include
 * schema:
 * type: string
 * description: Включити пов'язані дані (наприклад, "author,reviews")
 * security:
 * - bearerAuth: [] # Позначення, що маршрут вимагає JWT
 * responses:
 * 200:
 * description: Детальна інформація про книгу.
 * 401:
 * description: Не авторизований.
 * 404:
 * description: Книгу не знайдено.
 * tags:
 * - Books
 */
router.get('/:id', authenticateToken, bookController.getBookById);

/**
 * @swagger
 * /books:
 * post:
 * summary: Створити нову книгу
 * description: Додає нову книгу до бази даних (потрібні права
```

```
модератора/адміна).
 * security:
 * - bearerAuth: []
 * requestBody:
 * required: true
 * content:
 * application/json:
 * schema:
 * $ref: '#/components/schemas/NewBookPayload' # Посилання на схему
тіла запиту
 * responses:
 * 210:
 * description: Книгу успішно створено.
 * 400:
 * description: Некоректні дані.
 * 403:
 * description: Заборонено (недостатньо прав).
 * tags:
 * - Books
 */
router.post('/', authenticateToken, /* authorizeRoles('moderator',
'admin'), */ bookController.createBook);

export default router;
```

ДОДАТОК В – Компактний механізм обробки складного API запиту



ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____
_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»; що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації; що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів; що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)