Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система для організації та управління міською бібліотекою
(тема)

Виконав:
здобувач ___4____ року навчання
групи ПЗПІ-21-9

Владислав РЯБКО
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник ст.викл. каф. ПІ Олександр ОЛІЙНИК
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

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

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Рябку Владиславу Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Програмна система для організації та управління міською бібліотекою

Затверджена наказом по університету від «19» травня 2025 р. № 397Ст

2. Термін подання студентом роботи до екзаменаційної комісії 11.06.2025

3. Вихідні дані до роботи Розробити вебзастосунок для автоматизації міської бібліотеки з підтримкою ролей (читач, бібліотекар, адміністратор). Передбачити реєстрацію, пошук, бронювання, керування контентом за ролями. Back-end реалізувати на Python з Django REST Framework, front-end реалізувати на React з Axios. Як СУБД використати PostgreSQL.

4. Перелік питань, що потрібно опрацювати в роботі
Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	22.05.2025	*виконано*
2	Створення специфікації ПЗ	24.05.2025	*виконано*
3	Проектування ПЗ	27.05.2025	*виконано*
4	Розробка ПЗ	29.05.2025	*виконано*
5	Тестування ПЗ	30.05.2025	*виконано*
6	Оформлення пояснювальної записки	31.05.2025	*виконано*
7	Підготовка презентації та доповіді	01.06.2025	*виконано*
8	Попередній захист	03.06.2025	*виконано*
9	Нормоконтроль, рецензування	05.06.2025	*виконано*
10	Здача роботи у електронний архів	08.06.2025	*виконано*
11	Допуск до захисту у зав. кафедри	09.06.2025	*виконано*

Дата видачі завдання «20» травня 2025 р.

Здобувач  Владислав РЯБКО
(підпис)

Керівник роботи _____ ст.викл. каф. ПІ Олександр ОЛІЙНИК
(підпис) (посада, Власне ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра: 77 стр., 49 рис., 17 джерел.

AXIOS, DJANGO, DJANGO REST FRAMEWORK, MATERIAL UI, POSTGRESQL, REACT, SIMPLEJWT, TAILWINDCSS

Об’єкт розробки – система, для ефективного управління та автоматизації процесів в міській бібліотеці. Система охоплює клієнтську частину (frontend), та серверну частину (backend), забезпечуючи зручний доступ до каталогу книг, їх бронювання та додання в базу, а також перегляд подій та заходів.

Мета розробки – створення сучасного та зручного веб-додатку, який дозволяє автоматизувати процеси бібліотеки, спростити обслуговування відвідувачів і надати персоналу інструменти для керування книжковим фондом та заходами. Таке рішення має об'єднати всі ключові функції бібліотеки в одному інтерфейсі й полегшити роботу як працівників, так і користувачів.

Метод рішення – використання фреймворку Django для побудови серверної частини, що відповідає за обробку запитів, керування даними та авторизацію користувачів. Для створення API застосовано Django REST Framework у зв’язці з PostgreSQL як основною базою даних. Реалізовано підтримку автентифікації на основі JWT з розмежуванням прав доступу для різних ролей (читач, бібліотекар, адміністратор) за допомогою SimpleJWT. Фронтенд-частина реалізована з використанням React, що дозволило створити динамічний та інтерактивний інтерфейс. TailwindCSS і Material UI використано для побудови адаптивного дизайну, а Axios – для комунікації між клієнтською частиною та сервером.

Результатом розробки стала зручна, гнучка та масштабована система, яку можна використовувати не тільки в бібліотеках, а й в інших схожих закладах – наприклад, у школах чи культурних центрах.

ABSTRACT

AXIOS, DJANGO, DJANGO REST FRAMEWORK, MATERIAL UI, POSTGRESQL, REACT, SIMPLEJWT, TAILWINDCSS

The object of development is a system for efficient management and automation of processes in a municipal library. The system includes a client-side (frontend) and a server-side (backend), providing convenient access to the book catalog, book reservations, adding books to the database, as well as viewing events and activities.

The purpose of the development is to create a modern and user-friendly web application that automates library processes, simplifies visitor service, and provides staff with tools to manage the book collection and events. This solution is intended to combine all key library functions into a single interface and facilitate the work of both staff and users.

The solution method involves using the Django framework to build the server-side, which handles request processing, data management, and user authentication. Django REST Framework is used to build the API in conjunction with PostgreSQL as the main database. JWT-based authentication with role-based access control (reader, librarian, administrator) is implemented using SimpleJWT. The frontend is developed using React, which enabled the creation of a dynamic and interactive interface. TailwindCSS and Material UI are used to build a responsive design, and Axios is used for communication between the client-side and the server.

As a result, the system is convenient, flexible, and scalable, suitable not only for libraries but also for similar institutions such as schools or cultural centers. Thanks to its simple interface, the system facilitates the work of staff, improves visitor service, and enhances library management.

ЗМІСТ

ВСТУП

Сьогоднішній етап розвитку цифрових технологій має суттєвий вплив на освітні та культурні установи, серед яких особливе місце займають бібліотеки. У зв'язку зі зростанням кількості користувачів та збільшенням обсягу інформаційних ресурсів, усе актуальнішою стає потреба в автоматизації ключових процесів. Це стосується як обслуговування читачів, так і працівників бібліотеки.

З огляду на це, метою цієї роботи стало створення інтерактивного веб-застосунку для міської бібліотеки, який б поєднував зручність використання, гнучку структуру ролей та можливість масштабування. Система передбачає як функціонал для читачів – пошук і бронювання книг, перегляд історії позичань, перегляд заходів, – так і інструменти для бібліотекарів та адміністраторів, зокрема керування каталогом, модерацію користувачів і обробку запитів.

У ході реалізації було проаналізовано особливості існуючих систем та їх обмеження, після чого спроєктовано архітектуру, яка включає клієнтську частину (frontend), серверну частину (backend) і базу даних. Особливу увагу приділено ролям користувачів – з чітким розмежуванням доступу для читача, бібліотекаря та адміністратора. Таке рішення дозволяє зберігати гнучкість системи, а також легко доповнювати її новими функціями в майбутньому.

Запровадження такої системи може помітно покращити обслуговування відвідувачів, полегшити роботу працівників і зробити бібліотеку більш доступною та зручною для користувачів в онлайн форматі. Розробка буде корисною не лише для міських бібліотек, а й для шкіл чи культурних установ, де важливо впорядкувати роботу з інформацією та заохотити людей долучатися до заходів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

При дослідженні існуючих рішень для управління бібліотеками був застосований метод конкурентного бенчмаркінгу, який дозволяє провести порівняльний аналіз наявних на ринку продуктів та визначити їх сильні та слабкі сторони. Для об'єктивної оцінки були обрані ключові характеристики, важливі для бібліотечних систем, які оцінювалися за десятибальною шкалою.

Серед існуючих рішень для автоматизації бібліотек було проаналізовано кілька систем, які найбільш повно відповідають потребам сучасної міської бібліотеки.

Перший продукт – Koha.

Переваги системи:

- відкритий код, що дозволяє адаптувати систему під конкретні потреби;
- повноцінна підтримка міжнародних бібліотечних стандартів;
- наявність модуля для міжбібліотечного обміну;
- багатомовний інтерфейс;
- розвинена система звітності;
- можливість роботи з штрих-кодами та RFID-мітками.

Недоліки системи:

- складність налаштування та впровадження;
- надмірна функціональність для малих бібліотек;
- потреба у кваліфікованому технічному персоналі;
- застарілий користувацький інтерфейс;
- відсутність мобільного додатку;
- обмежені можливості для організації культурних заходів.

Аналіз показує, що хоча Koha є потужним інструментом для управління бібліотечним фондом, вона має певні обмеження з точки зору сучасних вимог до користувацького досвіду та мобільності. Це створює можливості для розробки більш сучасного та орієнтованого на користувача рішення. На рисунку 1.1

представлений Інтерфейс системи Koha.

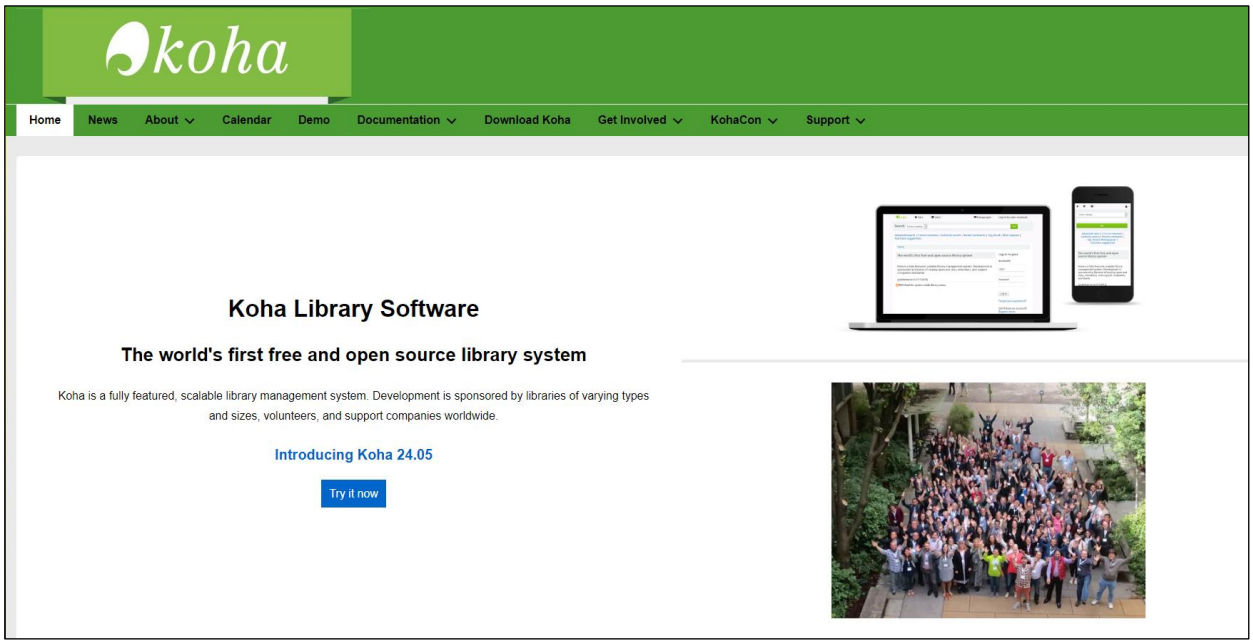


Рисунок 1.1 – Інтерфейс сайту Koha (за даними [1])

Другий продукт – Evergreen. Це ще одна система з відкритим кодом, що активно використовується в публічних і академічних бібліотеках. Вона підтримує управління каталогами, обробку запитів користувачів, резервування, управління обліковими записами та звітність. Її відмінною рисою є масштабованість, завдяки чому Evergreen підходить для великих бібліотечних систем з багатьма філіями.

Переваги системи:

– висока масштабованість для роботи з мережею бібліотек;
– потужна система каталогізації;
– підтримка консорціумів бібліотек;
– гнучка система пошуку;
– розвинені можливості для формування звітності;
– надійна система резервування книг.

Недоліки системи:

– високі вимоги до серверної інфраструктури;
– складний процес встановлення та налаштування;
– надмірна складність інтерфейсу для користувачів;
– обмежені можливості для взаємодії з читачами через сучасні канали

комунікації;

– застарілий дизайн інтерфейсу.

Аналіз Evergreen демонструє, що система найкраще підходить для великих бібліотечних мереж, але може бути занадто складною та ресурсомісткою для окремих міських бібліотек. Система також не відповідає сучасним вимогам до користувацького досвіду та соціальної взаємодії.

Порівнюючи Evergreen з Koha, можна відзначити, що обидві системи мають схожі проблеми з точки зору користувацького інтерфейсу та складності впровадження, але Evergreen має переваги в масштабованості та роботі з великими мережами бібліотек, тоді як Koha більш гнучка в налаштуванні та має кращу підтримку спільноти. На рисунку 1.2 показана головна сторінка застосунку.

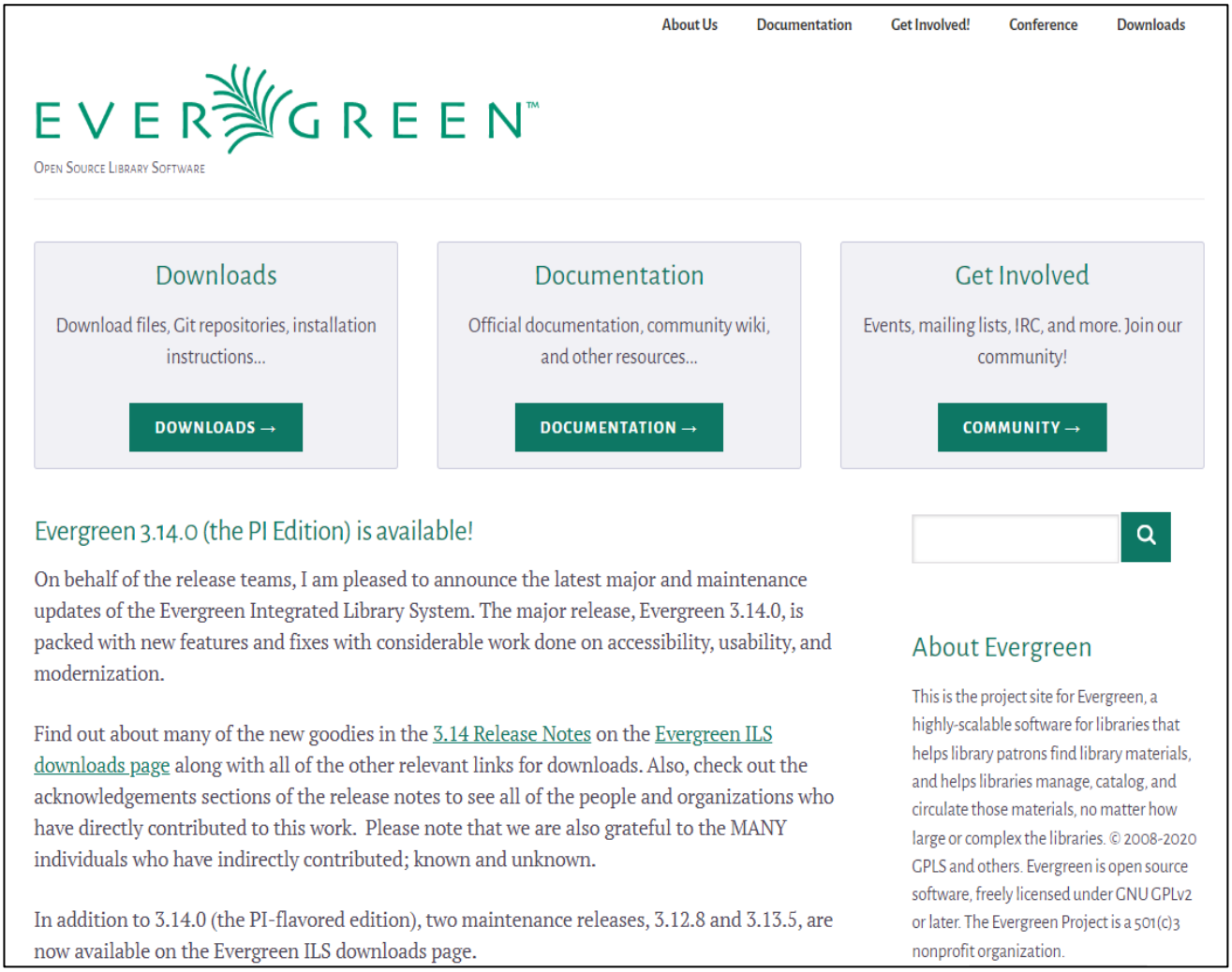


Рисунок 1.2 – Сайт Evergreen (за даними [2])

На рисунку 1.3 продемонстровано, як здійснюється пошук у

користувацькому інтерфейсі системи Evergreen ILS.

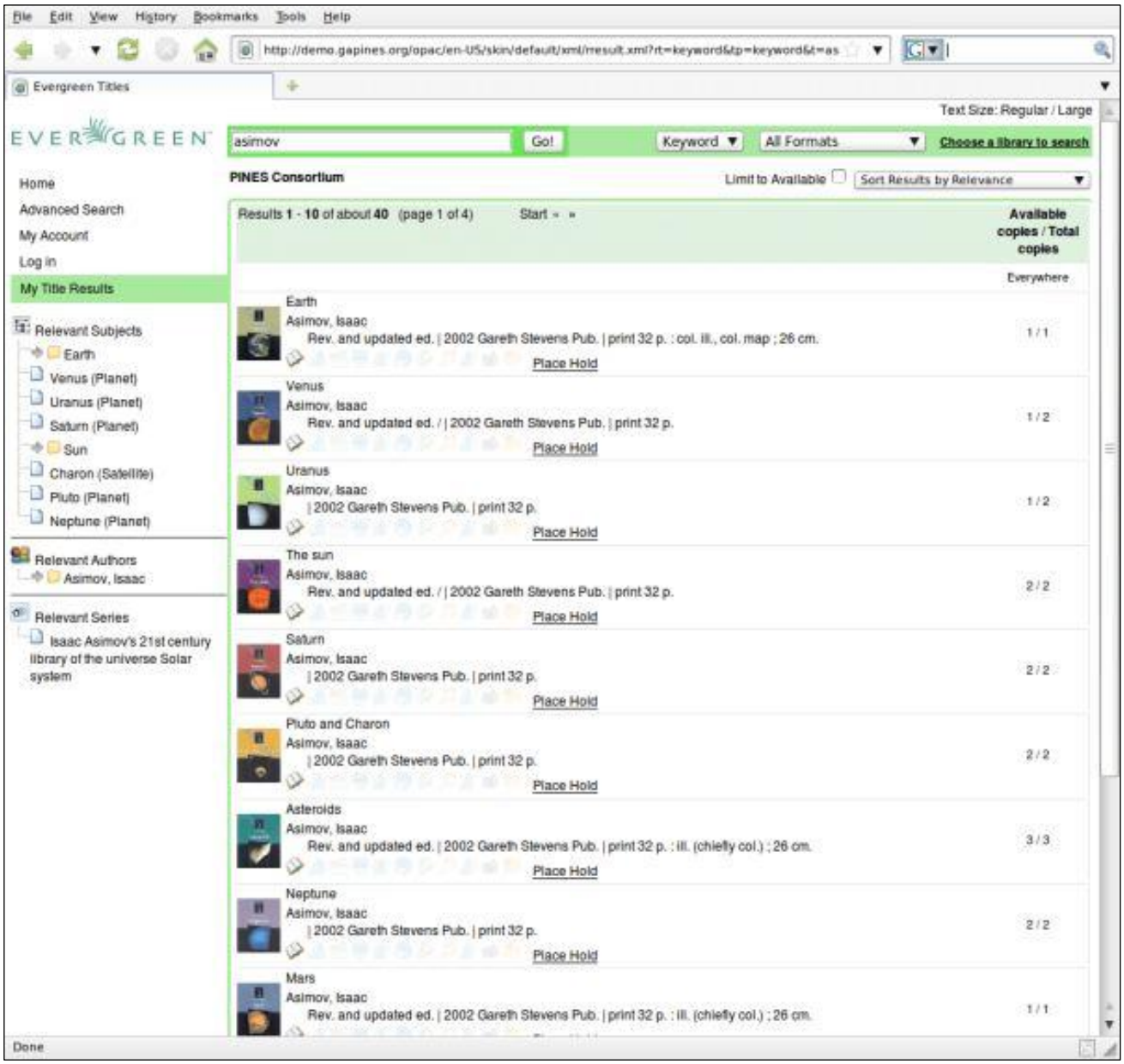


Рисунок 1.3 – Приклад взаємодії з Evergreen (за даними [3])

Виконано пошук за словом "Asimov", результатом чого є перелік назв, згрупованих за автором та тематикою. У лівій панелі користувач може обрати фільтри, наприклад, релевантних авторів (Asimov, Isaac) або предмети (наприклад, Planets), щоб уточнити результати. На рисунку 1.4 продемонстровано, як здійснюється пошук у користувацькому інтерфейсі системи Evergreen ILS з використанням фільтрів.

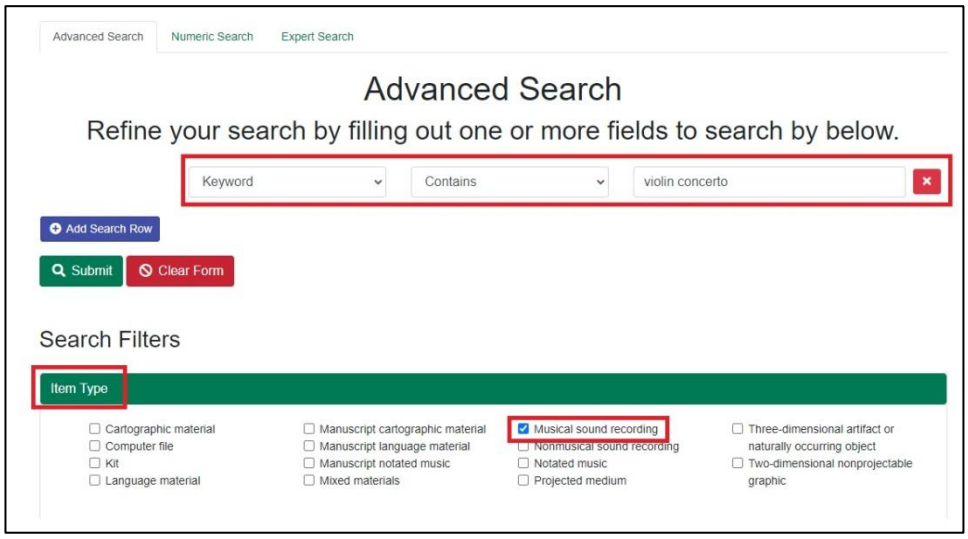


Рисунок 1.4 – Поглиблений пошук в Evergreen (за даними [4])

Третій продукт – Alma. Це комерційна бібліотечна система, популярна серед великих університетів. Вона пропонує розширені інструменти для каталогізації, управління електронними ресурсами, аналітики використання та інтеграції з академічними системами. Хоча Alma дуже функціональна, її вартість і складність можуть стати перешкодою для невеликих бібліотек.

Переваги системи:

– потужна хмарна платформа для управління бібліотечними ресурсами;
– комплексна система роботи з електронними та друкованими виданнями;
– глибока аналітика використання ресурсів;
– розвинені інструменти для каталогізації;
– інтеграція з академічними інформаційними системами;
– підтримка міжнародних стандартів каталогізації;
– автоматизована система поповнення фондів.

Недоліки системи:

– висока вартість впровадження та утримання;
– надмірна складність для невеликих бібліотек;
– складний інтерфейс користувача;
– орієнтація переважно на academic-середовище;
– обмежена гнучкість для публічних бібліотек;

– додаткова плата за розширені функції;

– складність кастомізації під локальні потреби.

Alma чітко демонструє тренд переходу бібліотечних систем у хмарні рішення з потужним аналітичним інструментарієм. Водночас система має суттєві обмеження для невеликих міських бібліотек через складність та високу вартість. На рисунку 1.5 зображено сайт даної системи.

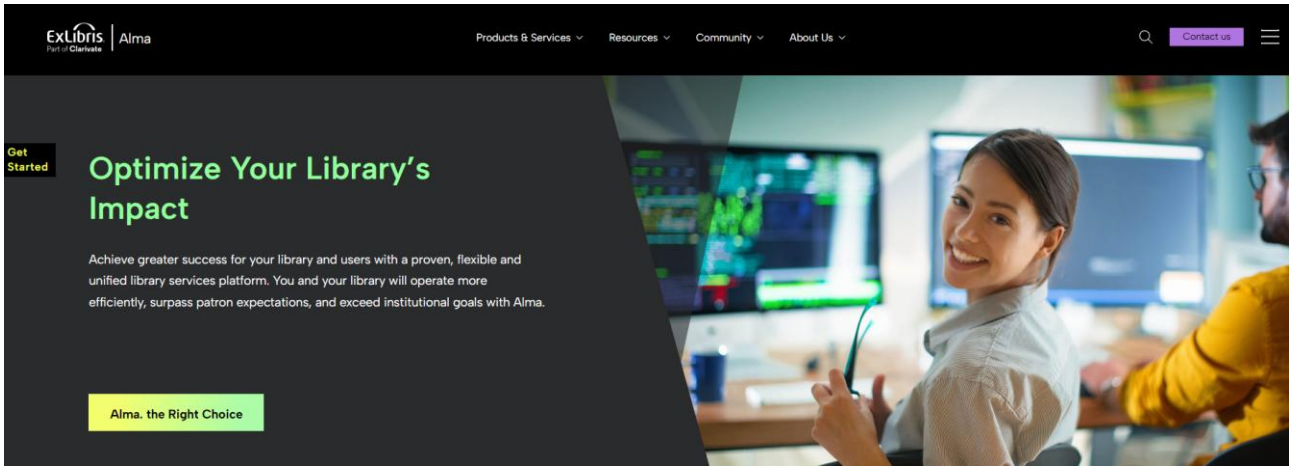


Рисунок 1.5 – Сайт Alma (за даними [5])

Наступний крок – створення порівняльної таблиці бібліотечних систем, обраних для аналізу. Це дозволить наочно продемонструвати сильні та слабкі сторони кожної системи та визначити найбільш перспективні напрямки для розробки власного рішення. Таблиця міститиме ключові характеристики, важливі для бібліотек, та бальну оцінку за кожним критерієм (див. рис. 1.6).

Порівняльна таблиця бібліотечних систем

	Характеристики	Koha	Evergreen	Alma	Наш продукт
1	Функціональність управління фондом	9	9	9	10
2	Зручність користувацького інтерфейсу	5	4	5	8
3	Простота впровадження	4	3	3	8
4	Гнучкість кастомізації	8	5	6	8
5	Інтеграція з сучасними технологіями	6	6	9	8
6	Підтримка мобільних пристроїв	3	3	6	8
7	Організація бібліотечних заходів	4	4	5	8
8	Аналітичні можливості	7	7	9	8
9	Вартість володіння	5	4	2	8

Рисунок 1.6 – Порівняльна матриця систем-конкурентів і моєї системи (рисунок виконано самостійно)

Проведений аналіз показує, що на ринку існує потреба у створенні сучасної, зручної та доступної системи управління бібліотеками, яка поєднає найкращі

практики існуючих рішень. Такий продукт має бути орієнтованим на потреби читачів, забезпечувати ефективну організацію внутрішніх процесів бібліотеки та надавати потужні аналітичні інструменти для прийняття управлінських рішень.

1.2 Виявлення проблем та актуалізація рішень

Наявні проблеми в існуючих бібліотечних системах, таких як Koha, Evergreen, та Alma (Ex Libris), включають складності в управлінні великим книжковим фондом, недостатню зручність інтерфейсу для користувачів і відсутність повноцінної інтеграції з додатковими сервісами. Бібліотеки потребують засобів для швидкого оновлення даних про книги, організації бібліотечних заходів та персоналізації користувацького досвіду.

Аналіз показав, що системи не в змозі задовольнити всі вимоги сучасної міської бібліотеки, зокрема щодо оптимального управління запитами, інтерактивних платформ для заходів та аналітики використання ресурсів. Це зумовлює потребу у створенні програмного продукту, який забезпечить зручне керування книжковим фондом, обробку запитів користувачів на резервування та повернення книг, а також функціонал для проведення заходів і звітності.

Основні виклики:

- управління книжковим фондом. Традиційні системи мають обмежені можливості з оновлення бібліотечного фонду в режимі реального часу, відстеження вибуття книг та контролю над резервами;
- обробка користувацьких запитів. Існуючі рішення часто не підтримують швидку обробку запитів на книги, що впливає на задоволеність користувачів. Крім того, автоматичне оновлення статусів книг під час позичання та повернення є важливим елементом для зменшення навантаження на бібліотекарів;
- збір та аналіз статистики використання. Наявні системи обмежено аналізують популярність книг та частоту запитів, що важливо для формування асортименту та збереження актуальності бібліотечного фонду;

– інтерактивність і зручність інтерфейсу. Багато бібліотечних систем мають недостатньо зручний та зрозумілий інтерфейс, що ускладнює процес пошуку та позичання книг для користувачів. Інтерактивна платформа для подій та презентацій може покращити взаємодію бібліотеки з відвідувачами.

Ці проблеми вказують на необхідність створення програмного забезпечення, яке може інтегрувати сучасні технології управління даними, надати зручний і доступний інтерфейс для користувачів та автоматизувати рутинні операції бібліотеки.

1.3 Постановка задачі

Задача розроблюваного продукту полягає в автоматизації процесів міської бібліотеки для оптимізації роботи бібліотекарів, покращення обслуговування користувачів та забезпечення ефективного управління ресурсами. Програмний продукт передбачає створення єдиного електронного каталогу книг, який дозволить бібліотекарям швидко та ефективно оновлювати інформацію про наявність книг у бібліотеці, а користувачам – бачити актуальні дані про доступність книг. Функціонал каталогу також включатиме автоматизацію обліку надходження та вибуття книг, що полегшить контроль за станом книжкового фонду та оптимізує роботу бібліотекарів.

Система також має включати функцію обробки запитів користувачів, яка дозволить автоматично приймати заявки на резервування, позичання та повернення книг з подальшим оновленням статусу кожної книги. Це значно знизить навантаження на персонал бібліотеки, підвищуючи оперативність обслуговування і створюючи зручніший процес для користувачів.

Окрім основного функціоналу, система має забезпечити інструменти для адміністратора, а саме можливість редагувати та видаляти користувачів, та змінняти системні змінні, наприклад зміна тривалості терміну позичання.

Для розширення функціоналу взаємодії з користувачами система забезпечить інтерактивну платформу для організації бібліотечних заходів, таких як презентації,

літературні зустрічі та майстер-класи. Це сприятиме залученню громади до культурного життя бібліотеки та покращить комунікацію між бібліотекою та її користувачами.

Програмний продукт складатиметься з серверної і клієнтської частин. Я відповідатиму за реалізацію обох. Серверна частина включатиме набір ендпоінтів для взаємодії з електронним каталогом, обробки запитів користувачів на резервування, управління особистими кабінетами користувачів, та додання та редагування книг в фонду бібіліотеки. Контролери серверної частини будуть оптимізовані для швидкої обробки інформації з мінімальним навантаженням на систему, забезпечуючи передачу необхідних даних до клієнтської частини або бази даних.

Клієнтська частина програмного продукту забезпечить зручні веб-інтерфейси для всіх ролей – адміністратора, бібліотекаря та читача. Адміністратор матиме доступ до панелі керування, де зможе налаштовувати систему, керувати обліковими записами та доступом. Бібліотекар матиме інструменти для додавання та редагування книг, управління та додання заходів, та перевірки історії бронювань користувачів. Користувачі зможуть здійснювати пошук і фільтрацію книг, переглядати їхній статус, резервувати та повертати книги, а також реєструватися на події. Інтерфейс для гостей дозволить переглядати каталог і події без доступу до персональних функцій.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Постановка мети

Основною метою створення цієї програмної системи є надання зручного інструменту для повного управління міською бібліотекою – від обслуговування читачів до адміністрування книжкового фонду, подій і користувачів.

У системі передбачено чіткий поділ ролей: звичайний користувач (читач), бібліотекар і адміністратор. Кожен із них має власні повноваження та доступ до відповідних функцій.

Ще одним важливим акцентом у розробці є простота й інтуїтивність інтерфейсу [6]. Сайт має бути доступним і зручним у користуванні як на комп'ютері, так і на мобільних пристроях. Каталог книг структурований, реалізований пошук і фільтри – за жанрами, авторами, доступністю [7] . Кожна книга містить окрему сторінку з детальним описом і інформацією про кількість доступних примірників.

Фронтенд реалізований за допомогою React, з використанням TailwindCSS та Material UI для створення адаптивного, сучасного дизайну. Для запитів до сервера використовується Axios. За вхід у систему та безпеку відповідає JWT-аутентифікація на базі Django REST Framework [8] і SimpleJWT [9].

Уся інформація зберігається в базі даних PostgreSQL – це дозволяє впорядковано обробляти дані про книги, користувачів, замовлення, події та інші дії. REST API забезпечує чітку структуру взаємодії між клієнтською та серверною частинами, а також відкриває можливість подальшої інтеграції з іншими сервісами.

Окрема увага приділяється обробці помилок та перевірці прав доступу – щоб користувач бачив зрозумілі повідомлення й не стикався з технічними бар'єрами. У перспективі система легко може бути розширена: наприклад, додати модулі зі статистикою, систему рекомендацій книжок або чат із бібліотекарем.

2.2 Загальний опис системи

Розроблена програмна система призначена для цифрового управління процесами у межах міської бібліотеки. Вона охоплює обидві частини веб-додатку

– клієнтську (frontend) та серверну (backend), забезпечуючи повний цикл взаємодії користувача з бібліотечними ресурсами. Система дозволяє здійснювати автентифікацію користувачів, переглядати каталог книжок, та заходи, бронювати книжки, а також адмініструвати вміст бібліотеки.

Завдяки архітектурі, побудованій на сучасних веб-технологіях, система підтримує масштабованість та зручну інтеграцію з іншими інформаційними сервісами. Основною метою було створити гнучку та зручну платформу, яку можна застосовувати не лише у міських бібліотеках, але й у культурних та освітніх закладах.

2.3 Основний функціонал системи

Функціональні можливості системи поділяються залежно від ролі користувача:

Гості (незареєстровані користувачі):

- ознайомлення з переліком подій;
- перегляд доступних книжок;
- реєстрація/вхід до системи.

Зареєстровані користувачі (читачі):

- авторизація через токен;
- перегляд каталогу книг та пошук книг;
- доступ до актуального списку подій;
- перегляд профілю;
- перегляд історії позичань.

Бібліотекарі:

- керування подіями;
- редагування книжкового фонду;
- перегляд інформації користувача;
- перегляд історії його бронювань користувача

Адміністратори: перегляд інформації користувачів; видалення користувачів; зміна параметрів системи.

Кожна роль отримує доступ лише до релевантного функціоналу, що реалізовано через систему прав доступу на основі JWT (JSON Web Token). Усі запити реалізовані через REST API, що забезпечує прозору взаємодію між клієнтом і сервером.

2.4 Загальні обмеження

Програмна система має наступні обмеження:

- для коректної роботи веб-додатку необхідна наявність стабільного доступу до мережі Інтернет;
- конфіденційна інформація користувачів зберігається у зашифрованому вигляді та недоступна жодному з адміністраторів або бібліотекарів (пароль, токен);
- кожна книга в базі даних має унікальний запис, ідентифікований за допомогою ID, і не може бути продубльована без зміни метаданих;
- бібліотекар не має права змінювати облікові дані інших працівників або користувачів – такі дії доступні виключно адміністратору;

2.5 Припущення та залежності

Наявні припущення та залежності:

- користувач може створити бронювання книги лише за умови, що доступна хоча б одна копія книги;
- бронювання прив’язується до конкретного користувача, і лише він і бібліотекарь можуть змінити статус бронювання;
- у випадку видалення книги або користувача, пов’язані з ними записи бронювань зберігаються в архіві для цілей аудиту.

3 АРХІТЕКТУРА ТА ПРОЕКТУВАННЯ ПЗ

3.1 UML проєктування ПЗ

У межах проєктування програмної системи для автоматизації міської бібліотеки було створено діаграми прецедентів (use case diagrams), які демонструють функціональні можливості для різних ролей користувачів: читачів, бібліотекарів та адміністраторів [10]. Ці діаграми відображають взаємодію користувача із системою через клієнтський інтерфейс, зокрема дії, пов'язані з пошуком книг, переглядом подій, бронюванням та керуванням обліковим записом.

Функціональність для ролі читача подано на рисунку 3.1.

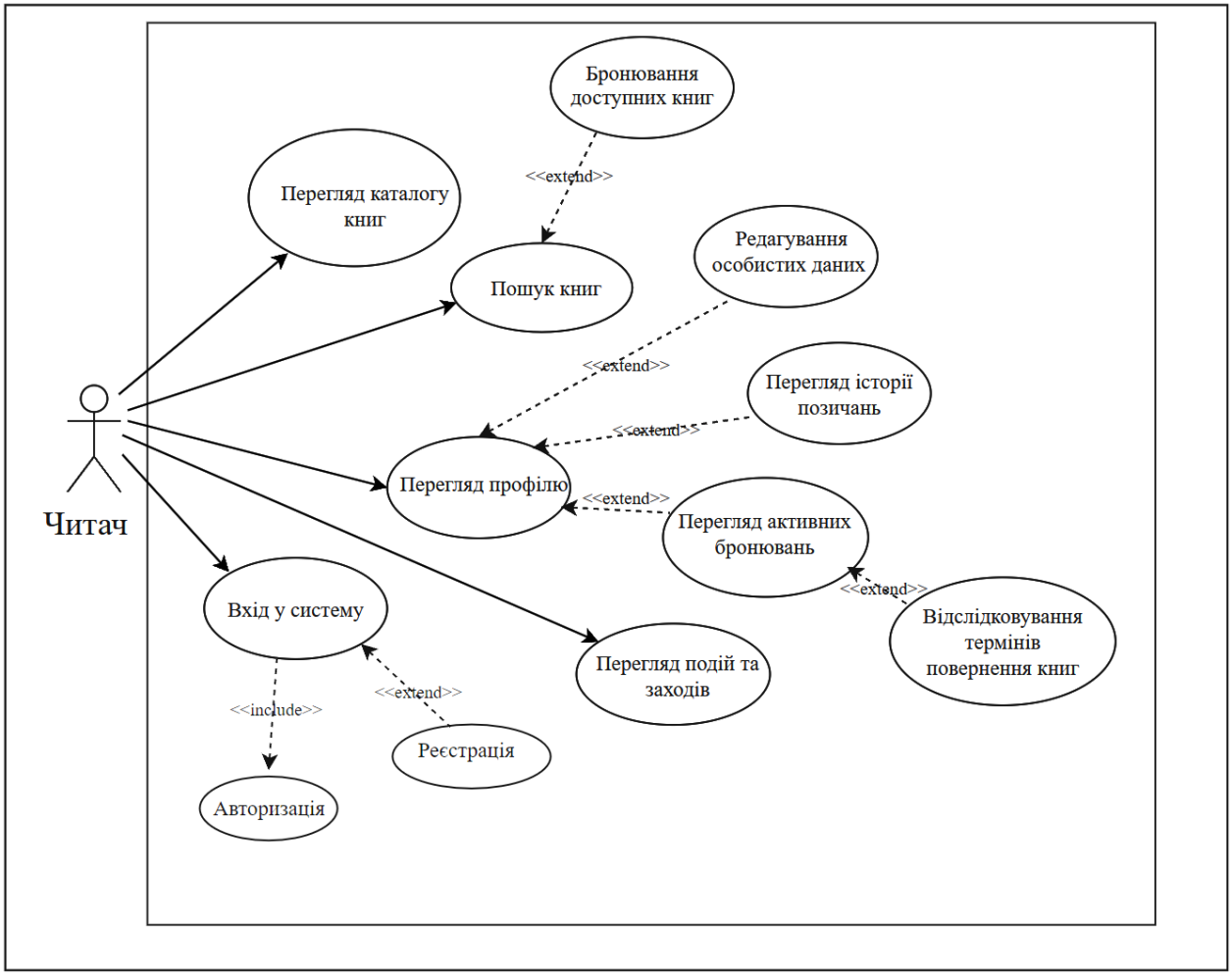


Рисунок 3.1 – Use Case діаграма для ролі Читача (рисунок виконано самостійно)

Читач має такі можливості як:

– перегляд каталогу книг;

– пошук книг за автором, назвою або жанром;

– перегляд подій та заходів;

– реєстрація та вхід у систему;

– бронювання доступних книг;

– перегляд власного профілю та його редагування;

– перегляд історії позичань та активних бронювань;

– відстеження термінів повернення книг;

– вихід із облікового запису.

Бібліотекар, має розширений набір функцій, необхідних для обслуговування читачів і підтримки актуальності бібліотечного фонду. Його функціональні можливості показані на рисунку 3.2.

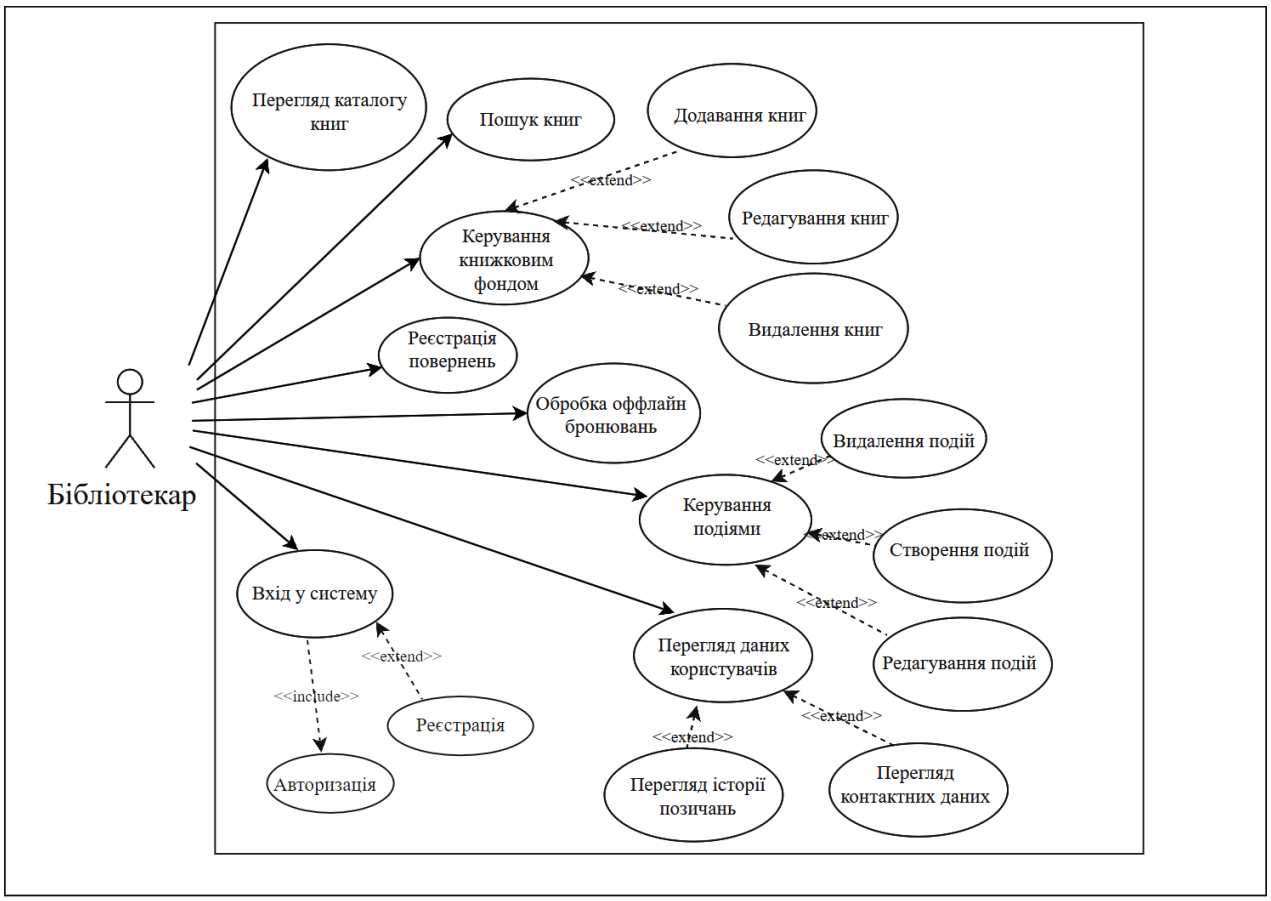


Рисунок 3.2 – Use Case діаграма для ролі Бібліотекаря
(рисунок виконано самостійно)

Бібліотекар має такі можливості:

– вхід у систему та авторизація;

- реєстрація нового облікового запису;
- перегляд та пошук книг у каталозі;
- керування книжковим фондом (додавання, редагування, видалення книг);
- реєстрація повернень книг;
- обробка офлайн-бронювань користувачів;
- керування подіями (створення, редагування, видалення заходів);
- перегляд контактних даних користувачів;
- перегляд історії позичань читачів;

Цей набір дій дозволяє бібліотекарю зручно та швидко виконувати дії, пов'язані з обігом книг і подіями бібліотеки, а також надавати базову підтримку читачам. Така роль забезпечує баланс між функціональністю й обмеженим доступом до критично важливої інформації, яку може змінювати тільки адміністратор.

Адміністратор має повний доступ до функціональності системи, включно з усіма можливостями, які доступні бібліотекарю. Він може працювати з книжковим фондом, обробляти повернення, керувати подіями та переглядати інформацію про користувачів. Окрім цього, адміністратор володіє розширеними правами, що дають змогу керувати налаштуваннями системи на глобальному рівні, а також здійснювати повноцінне управління обліковими записами користувачів і їхніми правами доступу. Його функціонал представлено на рис. 3.3.

Адміністратор має такі можливості:

- редагування даних користувача (контакну інформацію та роль);
- видалення користувачів;
- зміна ролі користувача (читач, бібліотекар, адміністратор);
- налаштування глобальних параметрів системи (термін позичання, графік роботи бібліотеки).

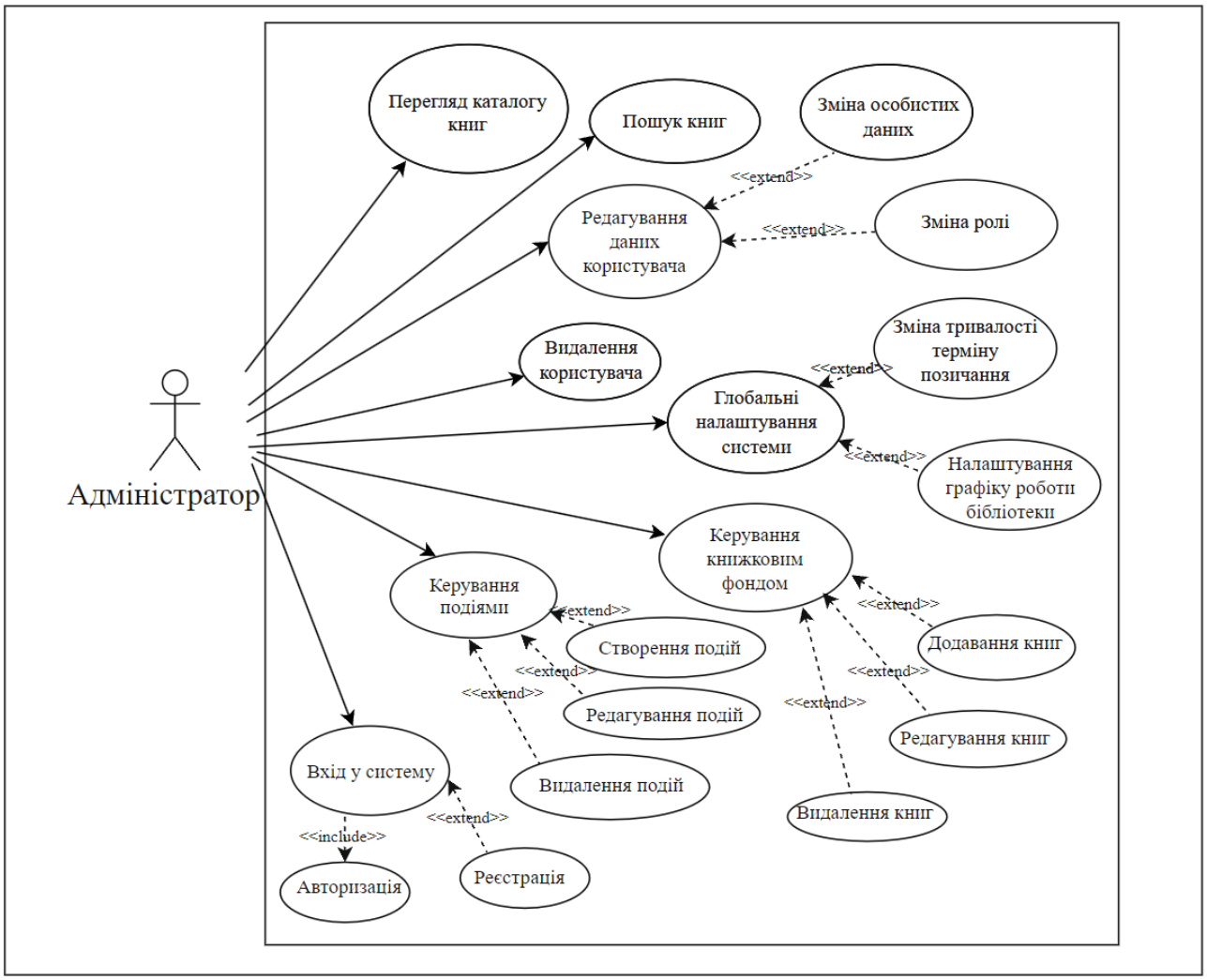


Рисунок 3.3 – Use Case діаграма для ролі Адміністратора (рисунок виконано самостійно)

3.2 Проєктування архітектури ПЗ

Для розробки серверної частини програмного забезпечення централізованої системи управління міською бібліотекою було обрано трирівневу архітектуру, що складається з трьох логічно відокремлених шарів: рівня презентації, рівня бізнес-логіки та рівня доступу до даних. Цей підхід є стандартом у розробці складних веб-застосунків завдяки своїм перевагам у гнучкості, масштабованості, зручності супроводу та чіткому розподілу функціональних обов'язків між компонентами системи [11].

Однією з головних переваг трирівневої архітектури є те, що кожен із рівнів може розроблятись та розширюватись незалежно. Наприклад, у разі зміни в логіці обробки бронювань або в процесах керування подіями бібліотеки, ці зміни можуть

бути реалізовані без необхідності втручання у код, який відповідає за інтерфейс користувача або доступ до бази даних. Це значно знижує ризики появи помилок під час оновлень і спрощує супровід та тестування системи. Крім того, при зростанні навантаження можна масштабувати окремі рівні системи, наприклад, окремо базу даних або логіку, не торкаючись решти частин застосунку.

Розмежування рівнів сприяє підвищенню безпеки системи. Користувачі не мають прямого доступу до бази даних, оскільки всі запити проходять через рівень бізнес-логіки, де виконується валідація вхідних даних, перевірка прав доступу та застосування інших захисних механізмів. Це дозволяє гнучко налаштовувати обмеження доступу для різних ролей без зміни загальної структури системи. Схему спроектованої архітектури представлено на рисунку 3.4.

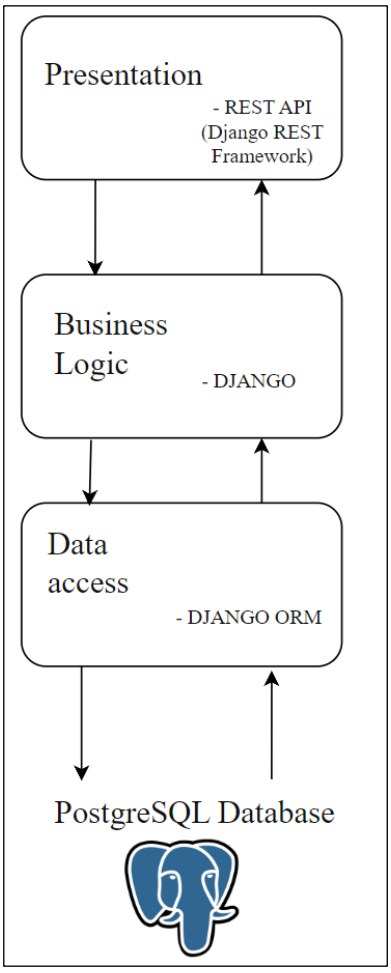


Рисунок 3.4 – Схема архітектури програмного забезпечення (рисунок виконаний самостійно)

На рівні доступу до даних реалізовано взаємодію з базою даних PostgreSQL – обраною через її стабільність, відкритість, зручність роботи з транзакціями та

високу продуктивність при роботі зі складними запитами. Комунікація з базою даних побудована за допомогою інструментів Django ORM, що дозволяє зручно працювати з моделями, не занурюючись у написання сирих SQL-запитів. Це пришвидшує розробку, полегшує підтримку коду і дозволяє зосередитися на вирішенні прикладних задач, а не технічних нюансах запитів до бази.

Середовище Django забезпечує зручну організацію бізнес-логіки – тобто тієї частини, яка відповідає за правила роботи системи. Саме тут розміщені механізми бронювання книг, обробки повернень, управління подіями, визначення доступу до певного функціоналу в залежності від ролі користувача (читач, бібліотекар, адміністратор). На цьому ж рівні реалізовано перевірку коректності введених даних, обробку помилок та логіку взаємодії з іншими частинами системи.

Фронтенд системи реалізований на базі React, що дозволяє створити інтерактивний, динамічний та зручний інтерфейс користувача. Для побудови запитів до серверної частини використовується бібліотека Axios, яка забезпечує швидку та зручну взаємодію з RESTful API. Вся взаємодія між клієнтом та сервером організована згідно з принципами REST-архітектури: запити надсилаються через HTTP, ресурси ідентифікуються за унікальними URI, використовується чітка структура запитів із методами GET, POST, PUT, DELETE. Такий підхід робить систему більш гнучкою, прозорою та такою, що легко підтримується.

REST API забезпечує незалежність між частинами застосунку: фронтенд лише надсилає запити та обробляє відповіді, не втручаючись у логіку роботи сервера. Це дозволяє легко змінювати чи оновлювати інтерфейс без ризику порушити бекенд, і навпаки – додавати нові функції на сервері без впливу на клієнтську частину [12].

Узагальнено, вибір такої архітектури дозволив побудувати систему, що легко масштабуються, розширюється та обслуговується. Вона чітко розділяє відповідальність між рівнями, підвищує безпеку, зменшує ризики при внесенні змін та забезпечує високу якість кінцевого продукту. Такий підхід виявився найбільш ефективним для проєкту міської бібліотеки, де важливо підтримувати

стійку роботу системи, враховувати різні ролі користувачів та забезпечити інтуїтивно зрозумілий доступ до усіх функцій.

3.3 Проєктування архітектури ПЗ

У процесі розроблення централізованої системи управління бібліотекою ключовим етапом стало створення моделі зберігання даних, яка забезпечує надійність, високу продуктивність і безпеку обробки інформації. Для реалізації було обрано PostgreSQL – сучасну об'єктно-реляційну систему управління базами даних (СУБД), що характеризується стабільністю та широким застосуванням у реальних проєктах.

Вибір PostgreSQL зумовлений її відповідністю принципам ACID (атомарність, узгодженість, ізольованість, довговічність), що гарантує цілісність даних у критичних операціях, таких як обробка бронювань або повернення книг [13]. СУБД ефективно обробляє великі обсяги даних і надає розширені можливості для роботи з індексами, зв’язками між таблицями та оптимізації складних SQL-запитів, що забезпечує високу продуктивність бізнес-процесів бібліотечної системи.

Додатковими перевагами PostgreSQL є її розширюваність, підтримка розширень і активна спільнота розробників, що полегшує вирішення нестандартних завдань. Система також забезпечує високий рівень безпеки завдяки вбудованим механізмам шифрування даних, контролю доступу на основі ролей і захисту мережевих з’єднань, що є критично важливим для збереження конфіденційності персональних даних користувачів.

На етапі логічного проєктування було розроблено реляційну модель бази даних, що охоплює основні сутності: користувачі, книги, бронювання, профілі користувачів, заходи, жанри, ролі та фото. Структура моделі оптимізована для забезпечення швидкого доступу до даних, мінімізації надлишковості та підтримки масштабованості. Взаємозв’язки між сутностями спроєктовані з урахуванням нормалізації для уникнення дублювання даних і забезпечення ефективної обробки запитів.

Розроблена модель бази даних є надійною основою для реалізації функціоналу системи, включаючи управління книжковим фондом, обслуговування користувачів, організацію подій та менеджментом користувачів. Використання PostgreSQL забезпечує стабільність, безпеку та гнучкість для подальшого розвитку системи. Було розроблено логічну модель сутностей системи (див. рис. 3.5).

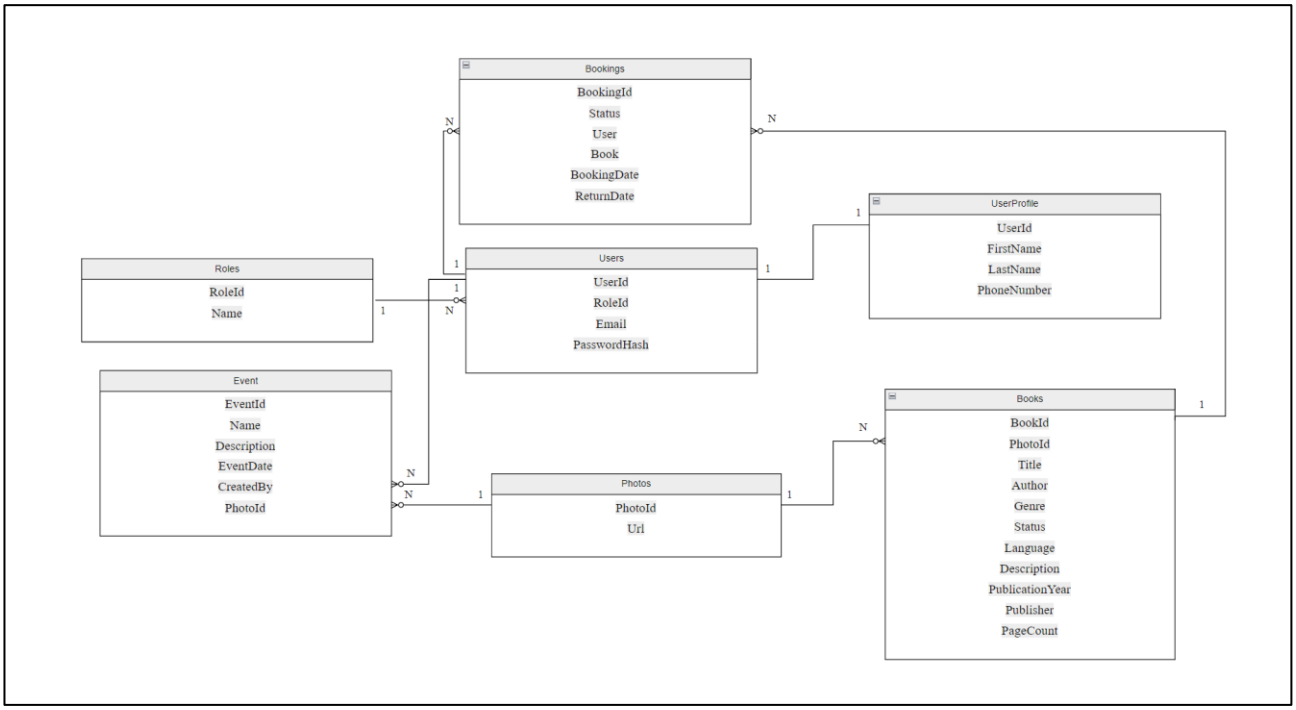


Рисунок 3.5 – ER-діаграма програмної системи (рисунок виконаний самостійно)

3.4 Приклади найцікавіших алгоритмів та методів

У межах реалізованої бібліотечної інформаційної системи однією з ключових функцій для забезпечення гнучкості в управлінні стала можливість динамічного редагування графіку роботи бібліотеки. Ця функція реалізована виключно для користувачів із роллю адміністратора, і дозволяє оперативно оновлювати години відкриття та закриття установи без внесення змін у код чи перезапуску сервера.

Особливістю реалізації є те, що змінений розклад відразу відображається на головній сторінці веб-інтерфейсу, доступного для всіх зареєстрованих та неавторизованих користувачів. Завдяки цьому користувачі завжди бачать актуальний графік роботи бібліотеки без необхідності звертатися до персоналу або перевіряти розклад на інших ресурсах.

Функціонал реалізовано через наступні ключові елементи: модель SystemSettings у Django, яка містить поля для зберігання часу роботи у форматі: початок та час праці для будніх та вихідних днів, форма редагування налаштувань на окремій сторінці адміністратора, яка дозволяє швидко внести зміни, REST API-ендпоїнт, через який оновлені дані передаються на фронтенд та фронтенд-компонент, який при кожному відкритті головної сторінки запитує сервер на наявність актуального розкладу та автоматично його виводить у нижній частині сайту.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Загальна структура додатку

У межах кваліфікаційної роботи було реалізовано сучасну систему автоматизації роботи міської бібліотеки, яка базується на архітектурі «клієнт-сервер». Основними критеріями під час вибору технологій були: швидкість розробки, простота підтримки та масштабованість. Для реалізації серверної частини було обрано мову програмування Python і фреймворк Django, а для побудови клієнтської частини – React.js. Передача даних між клієнтом і сервером здійснюється у форматі JSON через RESTful API.

Таке поєднання дозволяє ефективно реалізовувати функціональність бібліотеки, зокрема роботу з книжками, подіями, бронюваннями та користувачами. Серверна частина відповідає за обробку запитів, авторизацію та логіку бізнес-процесів, а клієнтська – за взаємодію з користувачем через зручний інтерфейс. Завдяки використанню Axios на фронтенді, обмін даними із сервером здійснюється асинхронно, без повного перезавантаження сторінок.

4.2 Архітектура програмної системи

Під час реалізації програмного забезпечення для системи управління міською бібліотекою було використано трирівневу архітектуру, яка є прикладом багаторівневого підходу до побудови вебзастосунків. Така структура дозволяє розділити систему на незалежні логічні шари, що спрощує її масштабування, і розвиток.

Архітектура передбачає такі рівні:

– Рівень презентації (клієнтська частина, фронтенд);

– Рівень бізнес-логіки (серверна логіка);

– Рівень доступу до даних (ORM + СУБД).

Рівень презинтації було реалізовано з використанням JavaScript-бібліотеки React, що дозволяє створювати динамічні інтерфейси користувача. Структура фронтенду передбачає розподіл логіки на компоненти, кожен з яких відповідає за

окрему частину интерфейса: перегляд книг, подій, оформлення бронювань, перегляд данних користувача тощо.

Зв’язок із сервером забезпечується через HTTP-запити за допомогою бібліотеки Axios, що дозволяє передавати дані у форматі JSON без перезавантаження сторінки. Структуру клієнтської частини наведено на рисунку 4.1.

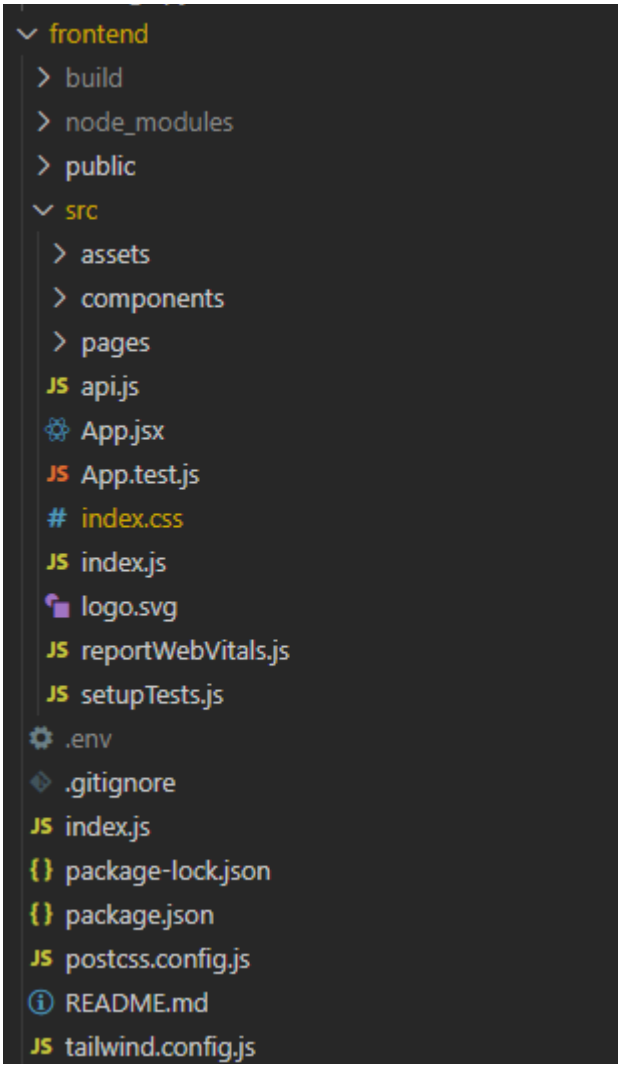


Рисунок 4.1 – Структура компонентів клієнтської частини системи [17] (рисунок виконано самостійно)

Обробка запитів, аутентифікація, перевірка прав доступу та виконання дій у відповідь на запити реалізовані на рівні бізнес-логіки. Для цього використано Django REST Framework, який забезпечує побудову RESTful API з підтримкою серіалізації, авторизації, обробки помилок, пагінації та ролей користувачів. Запити

обробляються за допомогою APIView, ViewSet, а логіка реалізується у відповідних методах – зокрема, create(), retrieve(), update() та інших.

Рівень доступу до даних відповідає за збереження, вибірку та модифікацію даних у базі. За доступ до даних відповідають моделі Django, які визначають структуру таблиць у СУБД. В якості засобу роботи з базою використано ORM Django, що дозволяє працювати з даними без написання SQL-запитів [14].

Зв'язки між сутностями (наприклад, книга – бронювання, фото – подія) реалізовано через поля типу ForeignKey. Серіалізатори (serializers.py) відповідають за перетворення даних у формат JSON для подальшої взаємодії з клієнтом. Загальну структуру модулів серверної частини системи наведено на рисунку 4.2.

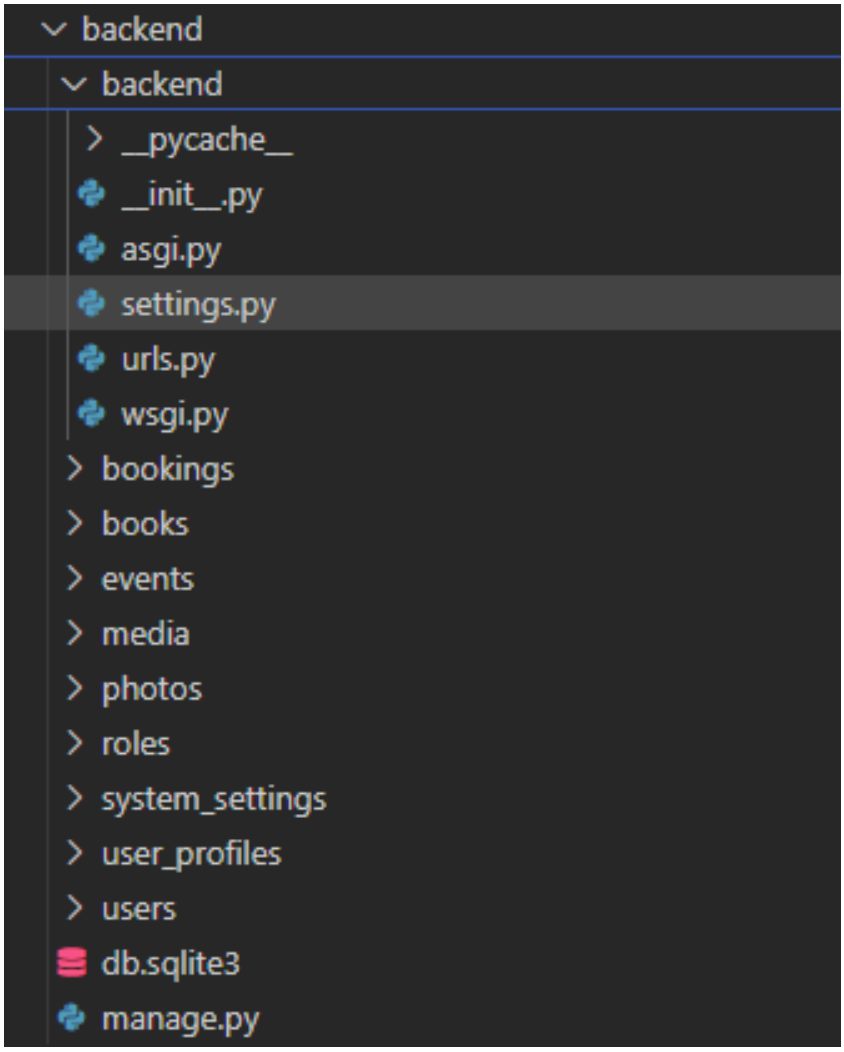


Рисунок 4.2 – Структура серверної частини проєкту [17] (рисунок виконано самостійно)

4.3 Взаємодія з базою даних через ORM

У якості системи керування базами даних у проєкті використано PostgreSQL – сучасну реляційну СУБД, що підтримує транзакції, зберігання даних у форматі JSONB, добре масштабується та забезпечує цілісність інформації відповідно до принципів ACID [13]. Такий вибір пояснюється потребою у надійному і стабільному збереженні даних.

Для роботи з базою застосовано вбудовану ORM Django, це робить розробку швидшою і знижує ризик помилок. Зміни у структурі бази вносяться через систему міграцій, яка автоматично відстежує зміни в моделях і створює відповідні оновлення. Приклад виконання команди міграції для таблиць додатку books, зображено на рисунку 4.3.

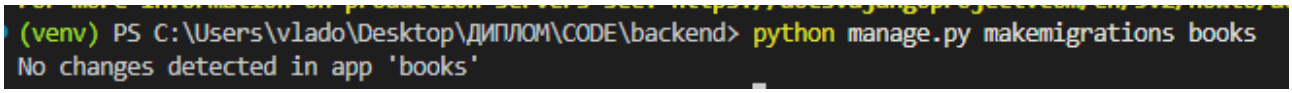


Рисунок 4.3 – Приклад міграції моделей для додатку books. (рисунок виконано самостійно)

4.4 Архітектура серверної частини

Серверну частину системи реалізовано з дотриманням принципів модульності та чіткого розділення відповідальностей. Основні функціональні блоки винесено в окремі Django-додатки – books, events, photos, users, bookings тощо. Кожен з модулів відповідає за окрему частину логіки, що значно полегшує підтримку, тестування та подальший розвиток системи.

Взаємодія між клієнтською частиною і сервером відбувається через REST API, побудоване за допомогою Django REST Framework. Для кожного додатку реалізовано відповідні APIView або ViewSet, які забезпечують обробку типових HTTP-запитів (GET, POST, PUT, DELETE). Передача даних здійснюється у форматі JSON через серіалізатори, що дозволяє фронтенду, написаному на React, легко інтегруватись із бекендом.

Кожен додаток має модель, яка відповідає за структуру збереження даних для окремих сутностей предметної області. Це дає змогу зменшити зв'язність між компонентами та забезпечити незалежну розробку.

Фрагмент моделі Book визначає структуру для збереження основної інформації про книгу: назву, автора, жанр, мову, опис, рік видання, кількість сторінок, статус та зображення книжки. Модель Book наведено на рисунку 4.4.

Рисунок 4.4 – Модель даних Book у додатку books [17] (рисунок виконано самостійно)

Зображення для книги завантажується окремо через спеціальну модель Photo, яка дозволяє зручно керувати файлами. Модель Photo наведено на рисунку 4.5.

Рисунок 4.5 – Модель Photo [17] (рисунок виконано самостійно)

Зображення фізично зберігаються у файловій системі, тоді як у базі даних зберігається лише шлях до файлу. Це дозволяє зменшити навантаження на БД та забезпечити швидший доступ до медіа.

Завантаження зображень реалізовано через окремий API-ендпоінт /api/photos/upload/, який входить до складу модуля photos. Цей ендпоінт приймає зображення у форматах JPEG, PNG та інших, зберігає файл у файловій системі, а також, за наявності ідентифікатора книги (book_id), автоматично прив'язує фото до відповідного запису в базі даних.

Обробка запиту, аутентифікація користувача та логіка збереження реалізовані у класі PhotoUploadView, що базується на APIView фреймворку Django REST Framework. На рисунку 4.6 зображено реалізацію ендпоінту та логіку прикріплення зображення до книги.

backend > photos > views.py > ...

```python
from rest_framework.views import APIView
from rest_framework.parsers import MultiPartParser
from rest_framework.response import Response
from rest_framework import status, permissions
from .models import Photo
from books.models import Book

class PhotoUploadView(APIView):
    parser_classes = [MultiPartParser]
    permission_classes = [permissions.IsAuthenticated]

    def post(self, request):
        uploaded_file = request.FILES.get('photo')
        book_id = request.data.get('book_id')

        if not uploaded_file:
            return Response({"error": "Файл не надіслано"}, status=status.HTTP_400_BAD_REQUEST)

        photo = Photo.objects.create(image=uploaded_file)

        if book_id:
            try:
                book = Book.objects.get(pk=book_id)
                book.photo = photo
                book.save()
            except Book.DoesNotExist:
                return Response({"error": "Книга не знайдена"}, status=status.HTTP_404_NOT_FOUND)

        return Response({"message": "Фото завантажено", "photo_id": photo.id}, status=status.HTTP_201_CREATED)
```

Рисунок 4.6 – Обробка запиту на завантаження фото та прив'язка до книги [17] (рисунок виконано самостійно)

Особливістю реалізації є підтримка прикріплення фотографій як до книг, так і до подій. Для подій передбачено окремий ендпоінт /api/events/upload-photo/, який приймає ID події разом із зображенням, після чого зберігає файл та прив'язує його до відповідного запису в базі.

У системі передбачено чітке розмежування прав доступу залежно від ролі користувача. Бібліотекар може керувати книжками, додавати або редагувати події, переглядати дані користувачів та їхню історію бронювань, а також обробляти офлайн-бронювання. Адміністратор, у свою чергу, має ті самі можливості, що й бібліотекар, а додатково може управляти користувачами та змінювати системні налаштування. Звичайний користувач, тобто читач, має змогу переглядати каталог доступної літератури, ознайомлюватися зі списком заходів, змінити свої конфіденціальні дані, бронювати книги та переглядати історію своїх позичань. Модель користувача можна побачити на рисунку 4.7.

backend > users > models.py > ...

```python
from django.db import models
from django.contrib.auth.models import AbstractBaseUser, BaseUserManager, PermissionsMixin
from roles.models import Role

class CustomUserManager(BaseUserManager):
    def create_user(self, email, password=None, **extra_fields):
        if not email:
            raise ValueError("Email is required")
        email = self.normalize_email(email)
        user = self.model(email=email, **extra_fields)
        user.set_password(password)
        user.save()
        return user

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault('is_staff', True)
        extra_fields.setdefault('is_superuser', True)
        return self.create_user(email, password, **extra_fields)

class User(AbstractBaseUser, PermissionsMixin):
    email = models.EmailField(unique=True)
    role = models.ForeignKey(Role, on_delete=models.SET_NULL, null=True, related_name='users')

    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = []

    objects = CustomUserManager()

    def __str__(self):
        return self.email
```

Рисунок 4.7 – Модель користувача [17] (рисунок виконано самостійно)

Роль користувача в системі зберігається у вигляді зовнішнього ключа до окремої моделі Role, яка розташована в модулі roles. Зберігання конфіденційної

інформації, такої як email, ім'я та прізвище, реалізовано через кастомну модель UserProfile, що розширює можливості базової моделі користувача Django.

Контроль доступу до окремих ендпоінтів системи реалізовано через механізм перевірки ролей, що відбувається автоматично під час обробки кожного запиту за допомогою permission-класів Django REST Framework або кастомної логіки. Це забезпечує надійне розмежування прав і дозволяє, наприклад, надавати можливість змінювати ролі або видаляти користувачів лише адміністраторам. На рисунку 4.8 показано реалізацію такого механізму контролю доступу на прикладі представлень модуля users.

```python
from rest_framework import generics, permissions
from rest_framework.exceptions import PermissionDenied
from .models import User
from .serializers import (
    UserSerializer,
    ChangeUserRoleSerializer,
    RegisterSerializer,
)

class UserListView(generics.ListAPIView):
    queryset = User.objects.all()
    serializer_class = UserSerializer
    permission_classes = [permissions.IsAuthenticated]


class ChangeUserRoleView(generics.UpdateAPIView):
    queryset = User.objects.all()
    serializer_class = ChangeUserRoleSerializer
    permission_classes = [permissions.IsAuthenticated]

    def dispatch(self, request, *args, **kwargs):
        if request.user.role.name != 'Administrator':
            raise PermissionDenied("Тільки адміністратор може змінювати ролі")
        return super().dispatch(request, *args, **kwargs)


class DeleteUserView(generics.DestroyAPIView):
    queryset = User.objects.all()
    permission_classes = [permissions.IsAuthenticated]

    def dispatch(self, request, *args, **kwargs):
        if request.user.role.name != 'Administrator':
            raise PermissionDenied("Лише адміністратор може видаляти користувачів")
        return super().dispatch(request, *args, **kwargs)

class RegisterView(generics.CreateAPIView):
    queryset = User.objects.all()
    serializer_class = RegisterSerializer
```

Рисунок 4.8 – Реалізація перевірки ролей користувачів у представленні модуля модуля users [17] (рисунок виконано самостійно)

Лише адміністратор і бібліотекар мають доступ до створення подій або редагування книг. Для цього використовуються кастомні permission-класи (IsAdminOrLibrarian) або умовна логіка у ViewSet'ах.

Механізм бронювання книжок у системі реалізований через окрему модель Booking, яка відповідає за зберігання інформації про користувача, книгу, дату бронювання, дату повернення та статус ("Booked", "Returned" або "Canceled"). Ця модель встановлює зв'язки з таблицями користувачів і книг, що дозволяє відслідковувати, хто саме забронював той чи інший примірник. На рисунку 4.9 представлено структуру моделі бронювання.

```python
from django.db import models
from users.models import User
from books.models import Book

class Booking(models.Model):
    STATUS_CHOICES = [
        ('Booked', 'Booked'),
        ('Returned', 'Returned'),
        ('Canceled', 'Canceled'),
    ]
    user = models.ForeignKey(User, on_delete=models.CASCADE, related_name='bookings')
    book = models.ForeignKey(Book, on_delete=models.CASCADE, related_name='bookings')
    status = models.CharField(max_length=10, choices=STATUS_CHOICES, default='Booked')
    booking_date = models.DateTimeField(auto_now_add=True)
    return_date = models.DateTimeField(null=True, blank=True)

    def __str__(self):
        return f"{self.user.email} - {self.book.title} ({self.status})"
```

Рисунок 4.9 – Модель бронювання книг [17] (рисунок виконано самостійно)

Для здійснення бронювання реалізовано окремий API-ендпоінт /api/bookings/create/, який обробляється класом CreateBookingView. Цей клас перевіряє авторизацію користувача, приймає POST-запит із необхідними даними, виконує валідацію та створює запис у таблиці бронювань. Відповідь повертається у серіалізованому форматі, що містить інформацію про успішно створене бронювання. Фрагмент реалізації ендпоінту показано на рисунку 4.10.

```python
class CreateBookingView(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def post(self, request):
        serializer = CreateBookingSerializer(data=request.data, context={'request': request})
        serializer.is_valid(raise_exception=True)
        booking = serializer.save()

        response_serializer = BookingSerializer(booking, context={'request': request})
        return Response(response_serializer.data, status=status.HTTP_201_CREATED)
```

Рисунок 4.10 – Обробка запиту на створення бронювання [17] (рисунок виконано самостійно)

Для обробки логіки створення використовується спеціальний серіалізатор CreateBookingSerializer. Він автоматично визначає користувача із контексту запиту, отримує ідентифікатор книги, встановлює статус Booked, а також розраховує дату повернення книги на основі глобального налаштування default_borrow_days з таблиці SystemSettings. Якщо налаштування не задано, використовується значення за замовчуванням – 14 днів. Реалізацію серіалізатору показано на рисунку 4.11.

```python
from rest_framework import serializers
from .models import Booking
from system_settings.models import SystemSettings
from datetime import timedelta
from django.utils import timezone
from books.serializers import BookSerializer
class CreateBookingSerializer(serializers.ModelSerializer):
    class Meta:
        model = Booking
        fields = ['book']

    def create(self, validated_data):
        user = self.context['request'].user
        book = validated_data['book']
        settings = SystemSettings.objects.first()
        borrow_days = settings.default_borrow_days if settings else 14

        return Booking.objects.create(
            user=user,
            book=book,
            status='Booked',
            return_date=timezone.now() + timedelta(days=borrow_days)
        )

```

Рисунок 4.11 – Реалізація серіалізатору [17] (рисунок виконано самостійно)

Також реалізовано перегляд активних бронювань та історії у особистому кабінеті користувача. Для перегляду замовлень користувача реалізовано окремий ендпоінт /api/bookings/my/, який повертає всі його бронювання, а також /api/bookings/active/ – лише активні. Вони обробляються відповідними класами та забезпечують зручне отримання даних у особистому кабінеті. Реалізацію даних ендпоінтів зображено на рисунку 4.12.

```python
class MyBookingsView(generics.ListAPIView):
    serializer_class = BookingSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        return Booking.objects.filter(user=self.request.user).order_by('-booking_date')

class ActiveBookingsView(generics.ListAPIView):
    serializer_class = BookingSerializer
    permission_classes = [permissions.IsAuthenticated]

    def get_queryset(self):
        return Booking.objects.filter(user=self.request.user, status='Booked').order_by('-booking_date')
```

Рисунок 4.12 – Перегляд активних та всіх бронювань користувача [17] (рисунок виконано самостійно)

Також у системі передбачено функціонал для перегляду історії бронювань за ID або електронною адресою користувача, що особливо корисно для бібліотекарів під час роботи з відвідувачами в офлайн-режимі. Реалізацію відповідної view-функції для отримання історії бронювань за ID або email користувача показано на рисунку 4.13.

```python
class BookingHistoryByIdOrEmail(APIView):
    permission_classes = [permissions.IsAuthenticated]

    def get(self, request):
        user_id = request.query_params.get('id')
        email = request.query_params.get('email')

        if not user_id and not email:
            return Response({'detail': 'Потрібно вказати або id, або email'}, status=status.HTTP_400_BAD_REQUEST)

        try:
            if user_id:
                user = User.objects.get(id=user_id)
            else:
                user = User.objects.get(email=email)
        except User.DoesNotExist:
            return Response({'detail': 'Користувача не знайдено'}, status=status.HTTP_404_NOT_FOUND)

        bookings = Booking.objects.filter(user=user).order_by('-booking_date')
        serializer = BookingSerializer(bookings, many=True)
        return Response(serializer.data)
```

Рисунок 4.13 – View для отримання історії бронювань користувача за email або ID [17] (рисунок виконано самостійно)

Для реалізації механізму реєстрації користувачів у системі використано окремий серіалізатор RegisterSerializer, який відповідає за створення нового облікового запису. У процесі обробки POST-запиту приймаються email, пароль та роль користувача, після чого автоматично створюється відповідний запис у базі

даних. Пароль зберігається у захищеному вигляді, а роль призначається через зовнішній ключ до моделі Role.

На рисунку 4.14 зображено реалізацію серіалізатора реєстрації.

```python
class RegisterSerializer(serializers.ModelSerializer):
    password = serializers.CharField(write_only=True)
    role = serializers.PrimaryKeyRelatedField(queryset=Role.objects.all())

    class Meta:
        model = User
        fields = ['email', 'password', 'role']

    def create(self, validated_data):
        role = validated_data.pop('role', None)
        user = User.objects.create_user(
            email=validated_data['email'],
            password=validated_data['password'],
        )
        user.role = role
        user.save()
        return user
```

Рисунок 4.14 – Серіалізатор для створення нового користувача [17] (рисунок виконано самостійно)

Після успішної аутентифікації користувач отримує токен, який зберігається у localStorage клієнта, і використовується в усіх наступних запитах. Реалізація побудована на основі SimpleJWT.

Для управління бібліотечними подіями реалізовано модель Event, яка містить основні поля: назву, опис, дату проведення, фотографію (за потреби) та автора створення події. На рисунку 4.15 зображено структуру моделі подій.

```python
from django.db import models
from users.models import User
from photos.models import Photo

class Event(models.Model):
    name = models.CharField(max_length=255)
    description = models.TextField()
    photo = models.ForeignKey(Photo, on_delete=models.SET_NULL, null=True, related_name='events')
    event_date = models.DateTimeField()
    created_by = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, related_name='created_events')

    def __str__(self):
        return self.name
```

Рисунок 4.15 – Модель події [17] (рисунок виконано самостійно)

Для отримання повного списку подій у системі було реалізовано окреме представлення EventListView, яке повертає всі події з бази даних та не потребує

авторизаці. Реалізацію представлення EventListView можна побачити на рисунку 4.16.

```
50  class EventListView(generics.ListAPIView):
51      queryset = Event.objects.all()
52      serializer_class = EventSerializer
53      permission_classes = [AllowAny]
```

Рисунок 4.16 – Обробка запиту на перегляд подій [17] (рисунок виконано самостійно)

Для роботи з книжками реалізовано кілька представлень. BookListView відповідає за виведення списку всіх книжок і доступний без авторизації. BookSearchView дозволяє здійснювати пошук за назвою, автором, жанром і мовою. Для перегляду конкретної книги використовується метод retrieve у BookViewSet, який повертає детальну інформацію або помилку, якщо книга не знайдена. На рисунку 4.17 показано реалізацію представлень для перегляду та пошуку книжок.

backend > books > views.py > BookViewSet > retrieve

```
1   from rest_framework import generics, permissions, filters, viewsets, status
2   from rest_framework.response import Response
3   from .models import Book
4   from .serializers import BookSerializer
5   from rest_framework.permissions import AllowAny
6
7   class BookSearchView(generics.ListAPIView):
8       queryset = Book.objects.all()
9       serializer_class = BookSerializer
10      permission_classes = [AllowAny]
11      filter_backends = [filters.SearchFilter]
12      search_fields = ['title', 'author', 'genre', 'language']
13
14  class BookListView(generics.ListAPIView):
15      queryset = Book.objects.all()
16      serializer_class = BookSerializer
17      permission_classes = [AllowAny]
18
19  class BookViewSet(viewsets.ModelViewSet):
20      queryset = Book.objects.all()
21      serializer_class = BookSerializer
22      permission_classes = [AllowAny]
23      def retrieve(self, request, pk=None):
24          try:
25              book = Book.objects.get(pk=pk)
26              serializer = BookSerializer(book, context={'request': request})
27              return Response(serializer.data)
28          except Book.DoesNotExist:
29              return Response({'error': 'Книга не знайдена'}, status=status.HTTP_404_NOT_FOUND)
```

Рисунок 4.17 – Представлення для роботи з книжками [17] (рисунок виконано самостійно)

У Django маршрутизація запитів здійснюється за допомогою системи URL-конфігурацій (URLconf), що визначає, які частини коду повинні обробляти запити за певними адресами. Головний файл маршрутизації, як правило, розташований у кореневому модулі backend/urls.py, і саме в ньому підключаються URL-маршрути всіх додатків проєкту через функцію include(). На рисунку 4.18 представлено головний файл маршрутизації, в якому реєструються всі API-ендпоінти системи, зокрема для книг, подій, бронювань, користувачів тощо.

backend > backend > urls.py > ...

```python
from django.contrib import admin
from django.urls import path, include
from django.views.generic import TemplateView
from django.conf import settings
from django.conf.urls.static import static
from rest_framework_simplejwt.views import (
    TokenObtainPairView,
    TokenRefreshView,
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', TemplateView.as_view(template_name='index.html')),
    path('api/books/', include('books.urls')),
    path('api/events/', include('events.urls')),
    path('api/users/', include('users.urls')),
    path('api/user-profiles/', include('user_profiles.urls')),
    path('api/bookings/', include('bookings.urls')),
    path('api/system/', include('system_settings.urls')),
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/photos/', include('photos.urls')),
]
urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

```

Рисунок 4.18 – Основний файл маршрутизації Django [17] (рисунок виконано самостійно)

Кожен додаток має власний файл urls.py, де описуються маршрути, пов’язані лише з цим модулем. Наприклад, у модулі users визначено окремі URL-шляхи для реєстрації, авторизації, зміни ролі та видалення користувача. На рисунку 4.19 зображено приклад локальної маршрутизації для модуля користувачів.

```
backend > users > urls.py > ...
1   from django.urls import path
2   from .views import RegisterView, ChangeUserRoleView, DeleteUserView
3   from rest_framework_simplejwt.views import (
4       TokenObtainPairView,
5       TokenRefreshView,
6   )
7
8   urlpatterns = [
9       path('register/', RegisterView.as_view(), name='register'),
10      path('login/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
11      path('token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
12      path('<int:pk>/change-role/', ChangeUserRoleView.as_view(), name='change-user-role'),
13      path('<int:pk>/delete/', DeleteUserView.as_view(), name='delete-user'),
14  ]
```

Рисунок 4.19 – Маршрутизація для модуля користувачів (рисунок виконано самостійно)

4.5 Інтерфейс користувача та доступність функціоналу

Фронтенд частину системи реалізовано з використанням бібліотеки React, що дозволяє створювати зручний, швидкий та адаптивний інтерфейс користувача. Комунікація з бекендом побудована на основі HTTP-запитів через бібліотеку Axios, що забезпечує обмін даними з REST API.

Однією з ключових сторінок є форма входу до системи, яка дає змогу пройти аутентифікацію за допомогою email та пароля. Після успішного введення облікових даних користувач отримує JWT токен, що зберігається у localStorage та використовується для авторизованих запитів у майбутньому. На рисунку 4.20 зображено інтерфейс авторизації користувача у вебзастосунку.

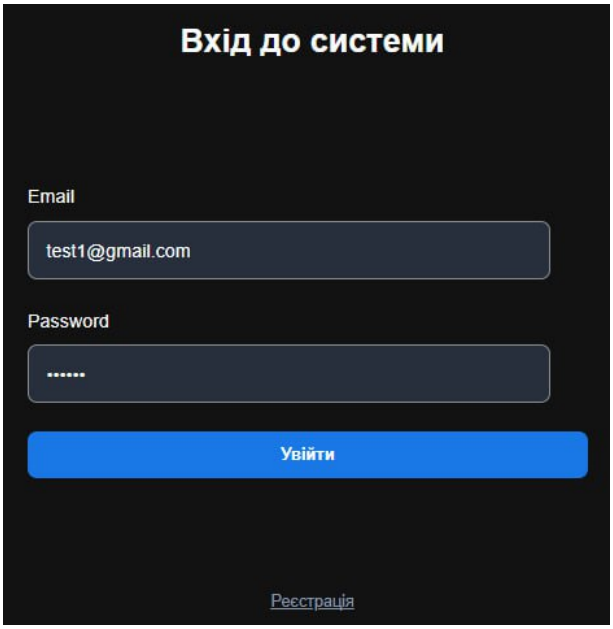


Рисунок 4.20 – Сторінка входу до системи (рисунок виконано самостійно)

Головна сторінка системи містить загальнодоступну інформацію: рекомендовані книги, майбутні події, поле пошуку, а також заголовки та навігацію. Вона є стартовою точкою для всіх категорій користувачів. Візуально її наповнення однакове, однак набір доступних функціональних елементів у верхньому меню залежить від ролі, з якою авторизований користувач.

Неавторизований користувач має можливість лише переглядати каталог книг та список подій, не маючи доступу до персонального кабінету чи функцій бронювання.

Користувач з роллю "Читач" після входу бачить кнопки перегляду своїх активних бронювань та історії позичань. У нього немає доступу до редагування контенту чи взаємодії з іншими користувачами.

Користувач з роллю "Бібліотекар" додатково отримує можливість додавати нові книги й події, переглядати список усіх користувачів та здійснювати офлайн-бронювання.

Адміністратор має повний набір функцій: усі можливості бібліотекаря плюс доступ до зміни параметрів системи, редагування ролей користувачів і керування налаштуваннями застосунку. На рисунку 4.21 зображено приклад головної сторінки для користувача з роллю бібліотекаря.

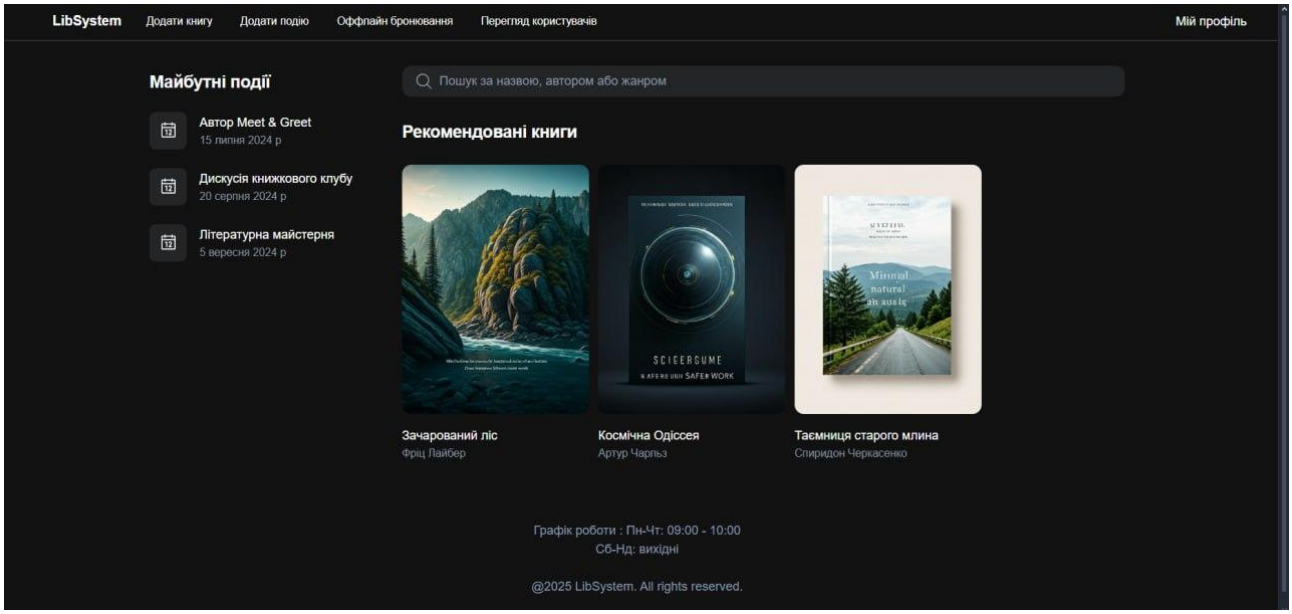


Рисунок 4.21 – Головна сторінка з розширеним меню для бібліотекаря (рисунок виконано самостійно)

Для додавання нової книги до бібліотечного фонду реалізовано окрему форму, доступну лише для авторизованих користувачів з роллю бібліотекаря або адміністратора. У формі необхідно заповнити основні поля: назву, автора, жанр, мову, видавництво, кількість сторінок, рік публікації, короткий опис та можливо прикріпити обкладинку.

Після заповнення даних користувач може натиснути кнопку «Додати книгу», після чого інформація передається на сервер і створюється новий запис у базі даних. Додавання обкладинки відбувається через окремий API-запит завантаження зображення. На рисунку 4.22 представлено інтерфейс форми створення нової книги.

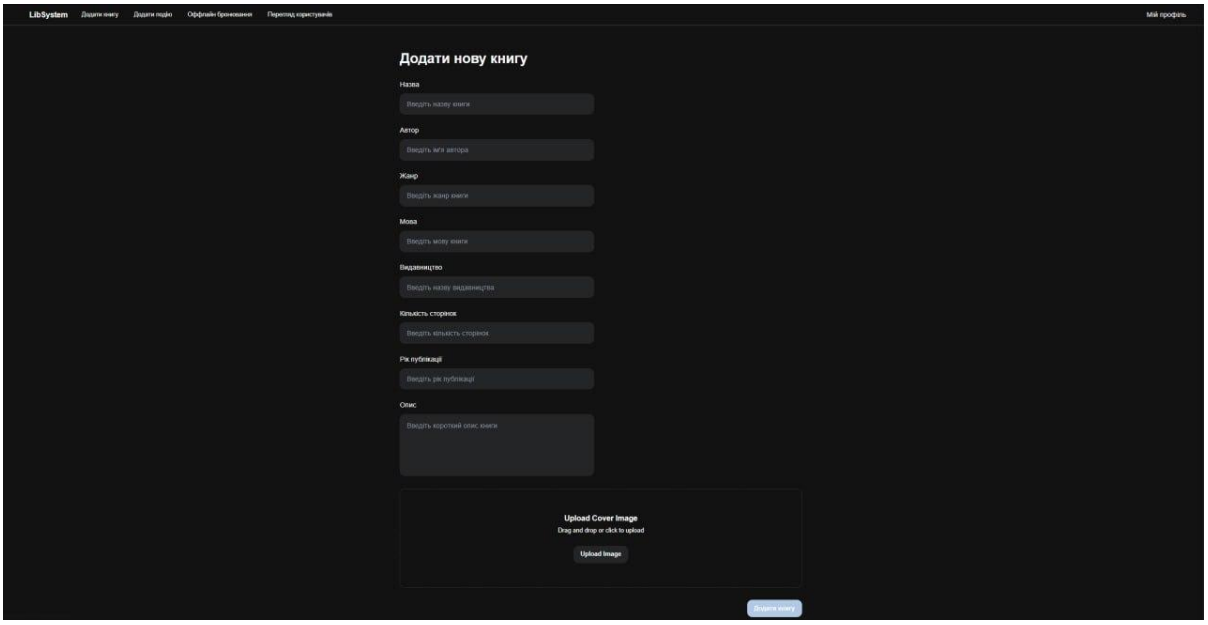


Рисунок 4.22 – Інтерфейс додавання книги у систему. (рисунок виконано самостійно)

4.6 Перевірка роботи ендпоінтів

Було протестовано процес реєстрації нового користувача за допомогою інструменту Postman [15]. Для цього було надіслано POST-запит на ендпоінт /api/users/register/ з тілом запиту у форматі JSON, яке містило електронну пошту, пароль та ID ролі (1 – читач). У відповідь сервер повернув статус 201 Created, що підтверджує успішне створення облікового запису. Результат успішної реєстрації відображено на рисунку 4.23.

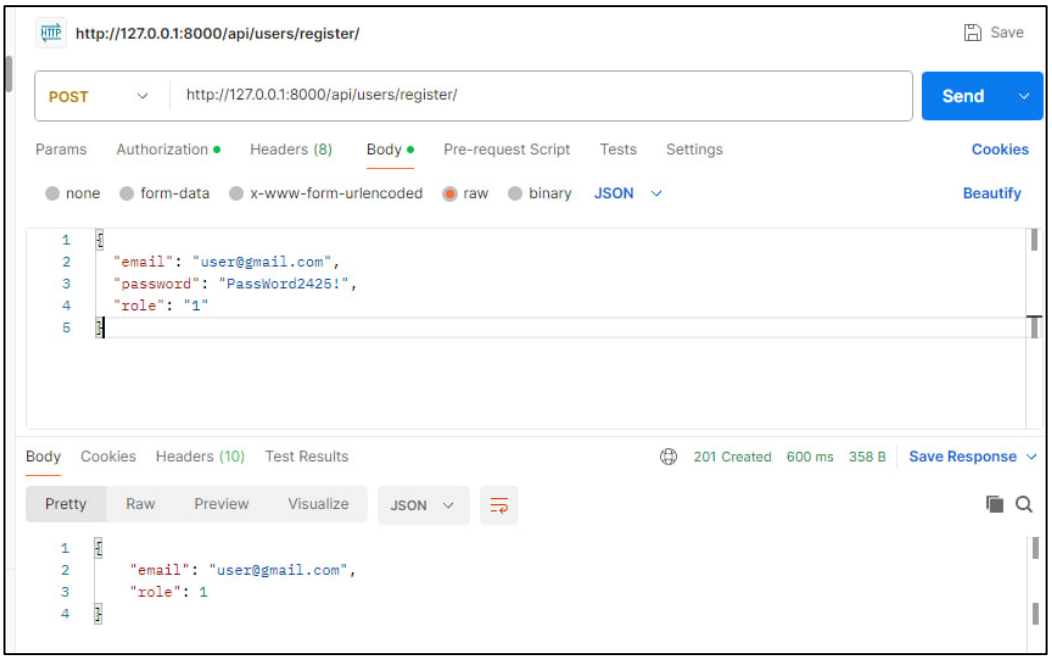


Рисунок 4.23 – Тестування реєстрації користувача через Postman (рисунок виконано самостійно)

Також було перевірено роботу механізму авторизації через JWT. Для цього у Postman було надіслано запит на ендпоінт /api/token/ із валідними обліковими даними. У відповідь сервер повернув access та refresh токени, що підтверджує успішну аутентифікацію. Результат успішної авторизації зображено на рисунку 4.24.

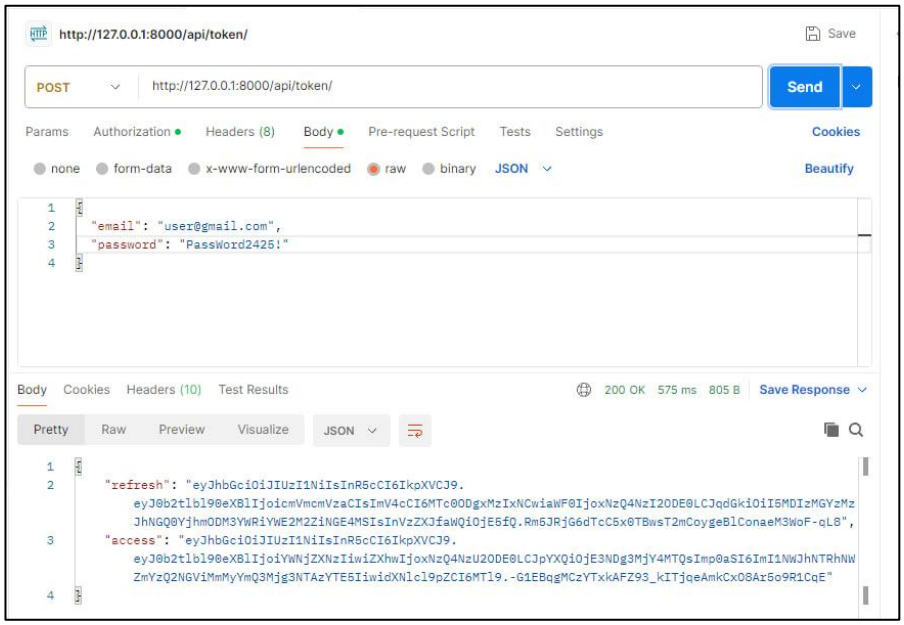


Рисунок 4.24 – Отримання JWT-токенів через Postman (рисунок виконано самостійно)

Після отримання JWT-токена було виконано запит на захищений маршрут /api/user-profiles/me/ з метою оновлення персональних даних користувача. Для авторизації токен було передано в заголовку. Результат оновлення персональних даних користувача зображено на рисунку 4.25.

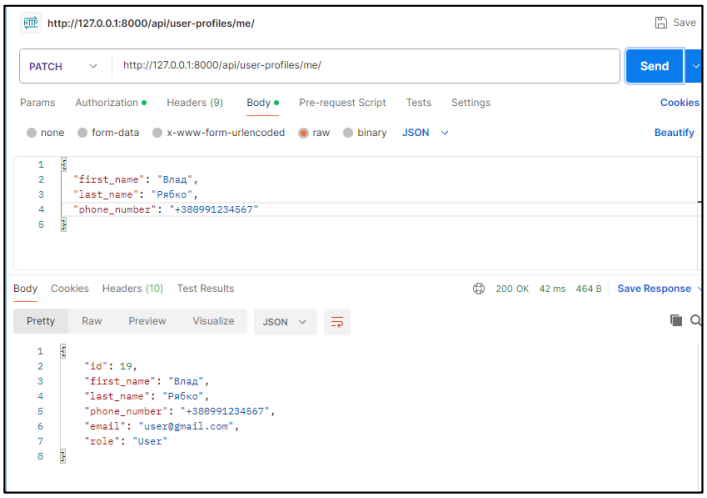


Рисунок 4.25 – Оновлення профілю авторизованого користувача через Postman (рисунок виконано самостійно)

Після цього було виконано запит із токеном користувача, що має роль бібліотекаря. Через endpoint /api/books/manage/ було надіслано POST-запит з інформацією про нову книгу. Як видно з відповіді, запис успішно створено, що підтверджується статусом та відповідю від сервера з данними створенної книги. Результат зображено на рисунку 4.26.

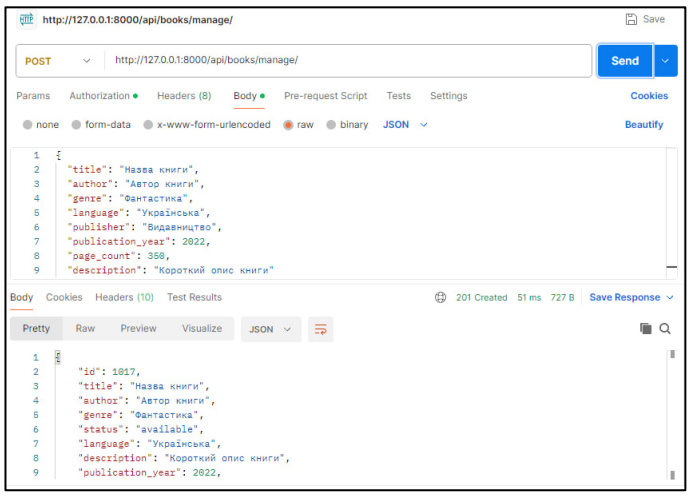


Рисунок 4.26 – Додавання нової книги через Postman (рисунок виконано самостійно)

Також було виконано GET-запит до ендпоінту /api/books/list/, який відповідає за отримання списку всіх доступних книг. Запит не вимагає авторизації (доступ відкритий для всіх користувачів), що дозволяє будь-кому переглядати каталог. Результат запиту зображено на рисунку 4.27.

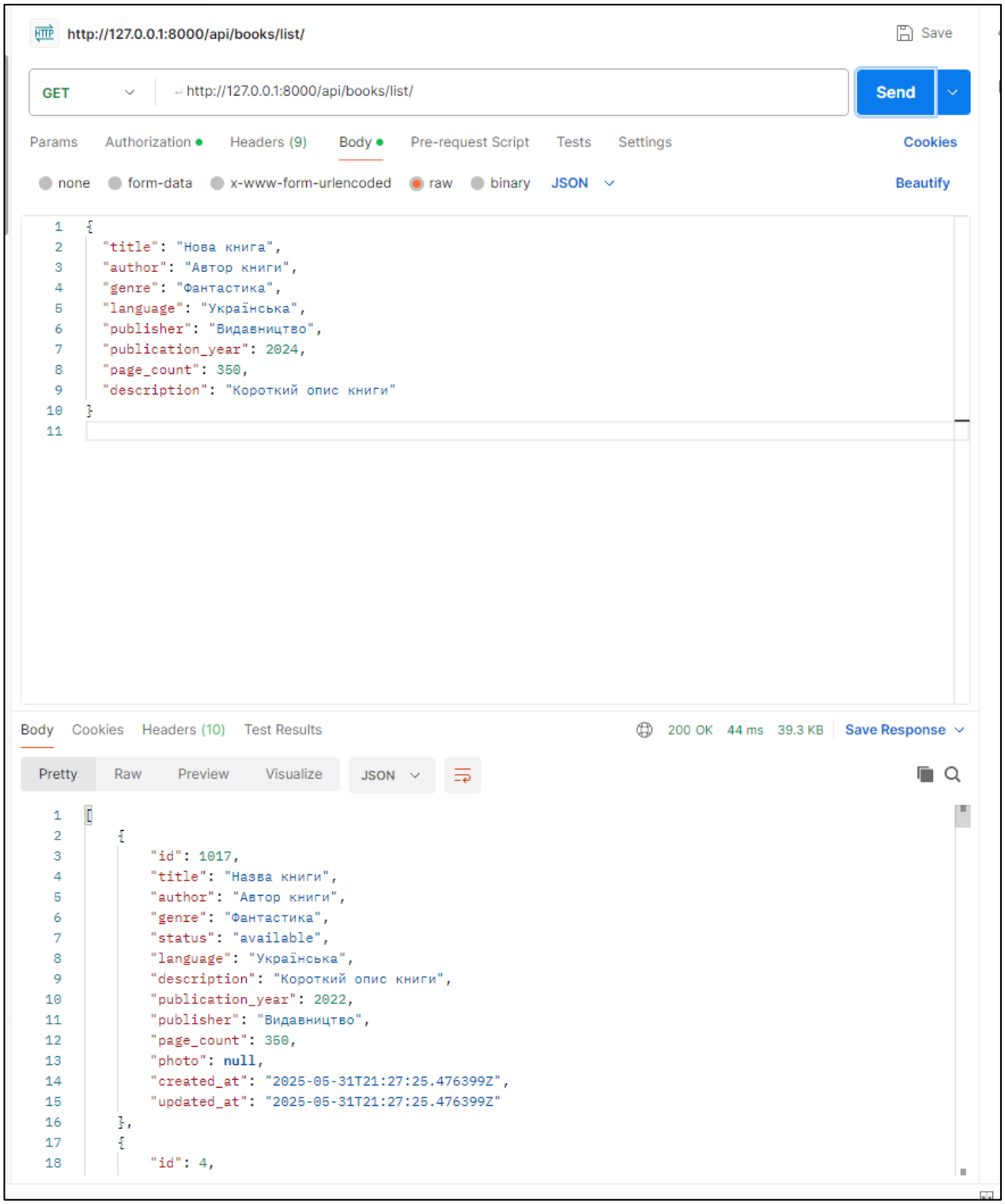


Рисунок 4.27 – Отримання переліку книг через API-запит (рисунок виконано самостійно)

За користувача з роллю читача було протестовано створення бронювання через ендпоінт /api/bookings/create/. У запиті передається ідентифікатор книги, яку потрібно забронювати. Сервер автоматично фіксує користувача, статус, дату бронювання та дату повернення. У відповідь надходить детальна інформація про створене бронювання.

Результат виконаного запиту зображено на рисунку 4.28.

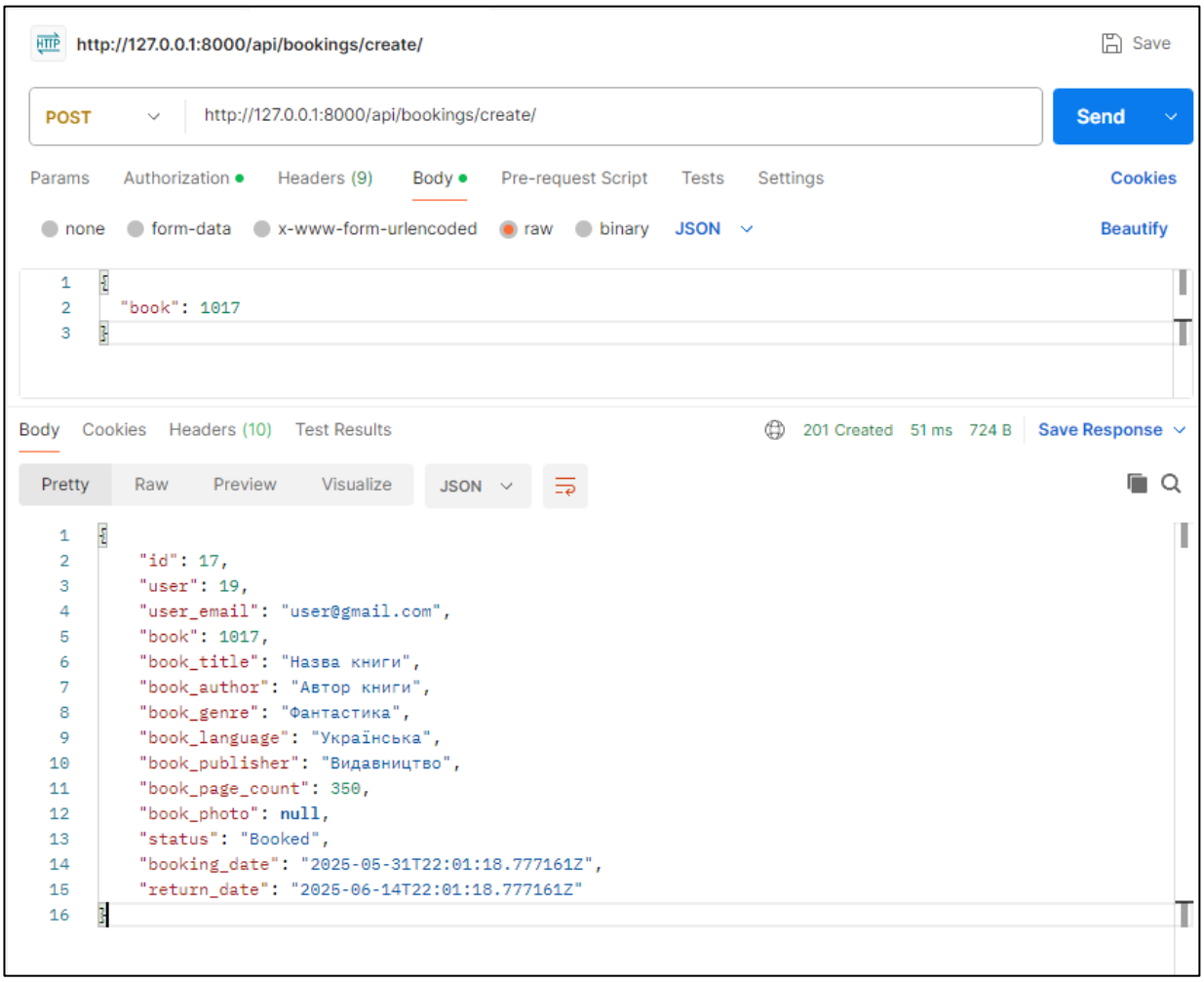


Рисунок 4.28 – Створення бронювання читачем через API (рисунок виконано самостійно)

За користувача з роллю адміністратора було протестовано перегляд і зміну системних параметрів через ендпоінт /api/system/settings/. Результат запиту зображено на рисунку 4.29.

Рисунок 4.29 – Перегляд системних налаштувань через API(рисунок виконано самостійно)

У відповіді повертаються ключові налаштування бібліотеки – кількість днів стандартного бронювання, час початку роботи та тривалість у робочі та вихідні дні.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Навантажувальне тестування API системи

Щоб переконатись, що розроблена система для міської бібліотеки працює стабільно та справляється з навантаженням, було проведено низку тестів із використанням інструменту Apache JMeter [16]. Основна увага приділялась перевірці найважливіших частин бекенду – авторизації, перегляду заходів та книжок.

Для моделювання реального навантаження, яке може виникати під час активного користування (наприклад, під час івентів або вихідних), були налаштовані тести з великою кількістю паралельних користувачів. JMeter дозволяє задавати кількість потоків, інтервали запуску, типи запитів і виводити статистику: середній час відповіді, кількість помилок та кількість запитів у секунду. Це дозволяє ефективно оцінити продуктивність API-серверу до впровадження у реальне середовище.

На першому етапі було виставлено базове налаштування, яке симулює 100 одночасних користувачів з ramp-up періодом в 1 секунду та виконанням лише одного циклу запитів. Таке навантаження вважається типовим для систем, що обслуговують міські сервіси. Налаштування можна побачити на рисунку 5.1.

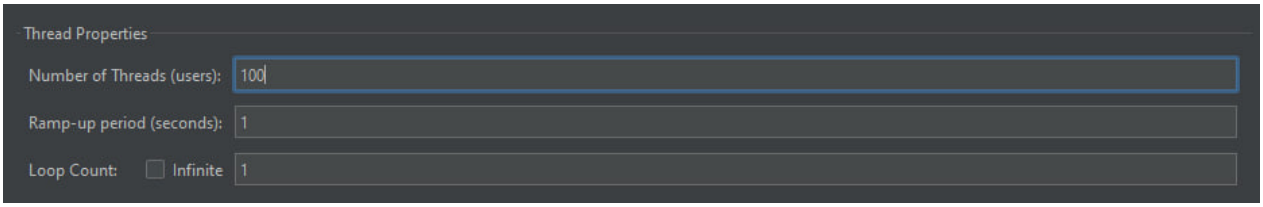


Рисунок 5.1 – Налаштування кількості користувачів у JMeter (рисунок виконано самостійно)

Далі було перевірено ендпоїнт авторизації. Було налаштовано HTTP-запит до ендпоінту /api/token/, через який відбувається авторизація користувачів. Тип запиту – POST, а в тілі було передано email та пароль. Налаштування наведено на рисунку 5.2.

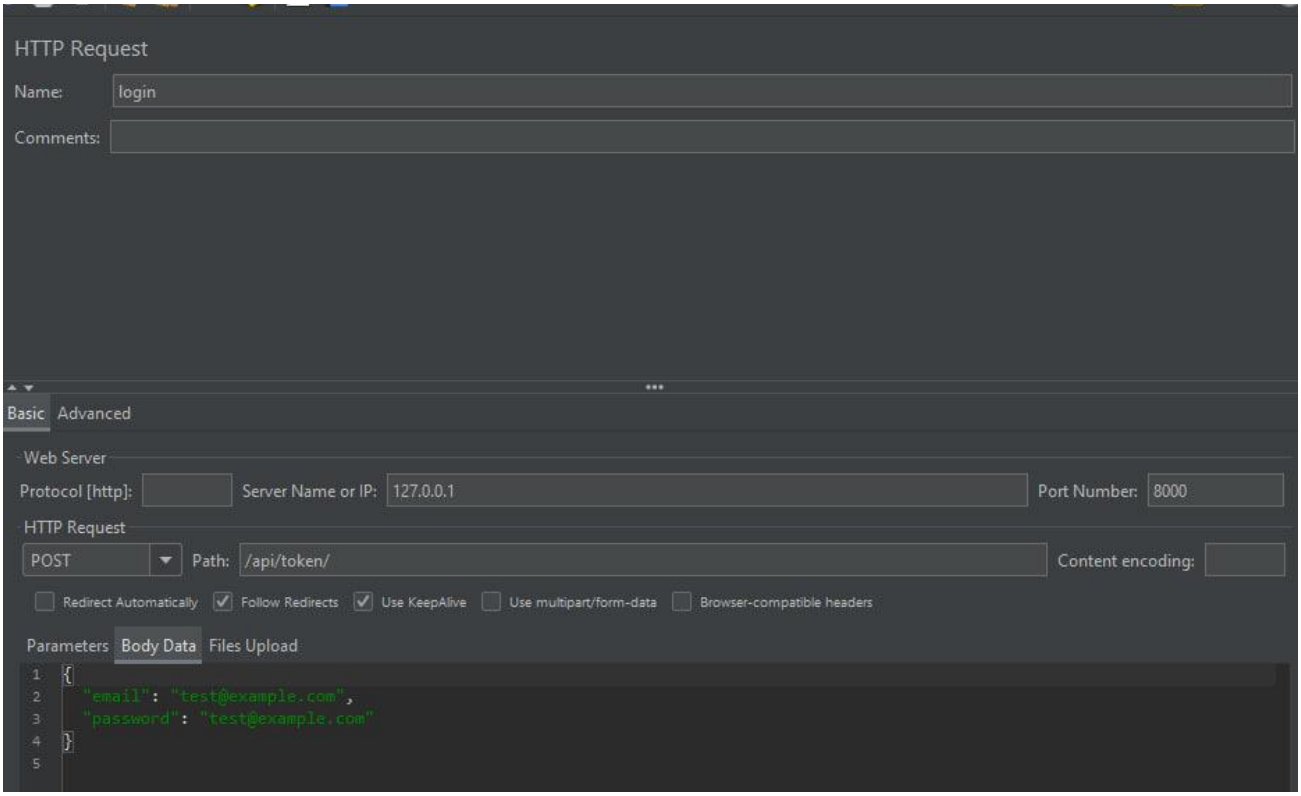
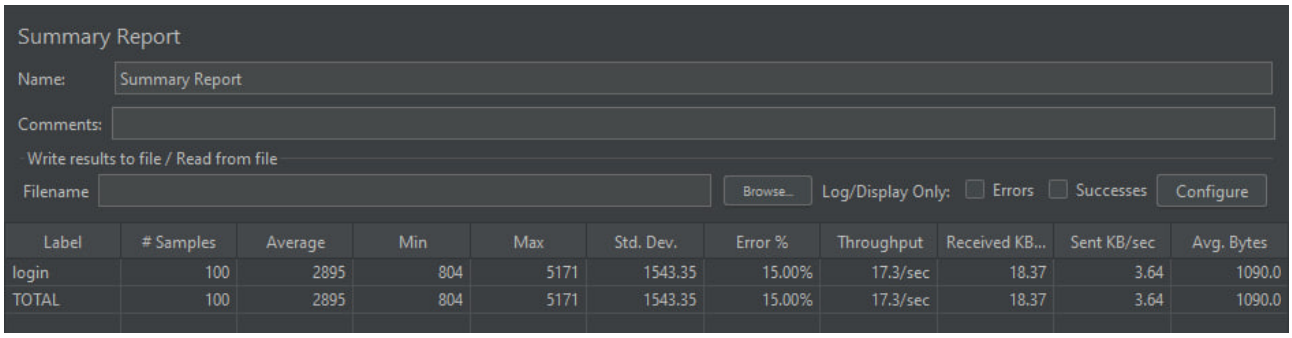


Рисунок 5.2 – Конфігурація HTTP-запиту для перевірки авторизації (рисунок виконано самостійно)

Після запуску тесту з 100 користувачами було отримано наступні результати, подані у таблиці Summary Report. Результати наведено на рисунку 5.3.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB...	Sent KB/sec	Avg. Bytes
login	100	2895	804	5171	1543.35	15.00%	17.3/sec	18.37	3.64	1090.0
TOTAL	100	2895	804	5171	1543.35	15.00%	17.3/sec	18.37	3.64	1090.0

Рисунок 5. 3 – Результати тестування авторизації при 100 користувачах (рисунок виконано самостійно)

Середній час відповіді складає 2895 мілісекунд, 15 відсотків помилок та пропускна здатність в 17.3 запитів за секунду. Ці результати вказують на навантаження системи під час авторизації. Після 80-го користувача почали з’являтися перші помилки у виконанні запитів. Однак така кількість одночасних

авторизацій є малоймовірною в умовах звичайної міської бібліотеки. У пікові періоди система, ймовірно, обслуговуватиме не більше 15–20 користувачів одночасно. Тому отримані результати можна вважати задовільними, а роботу системи – стабільною в межах реального навантаження.

Наступне тестування проводилось з 150 потоками. Для цього у Thread Group було встановлено 150 користувачів із тими ж параметрами ramp-up та loop count.

Налаштування відображено на рисунку 5.4.

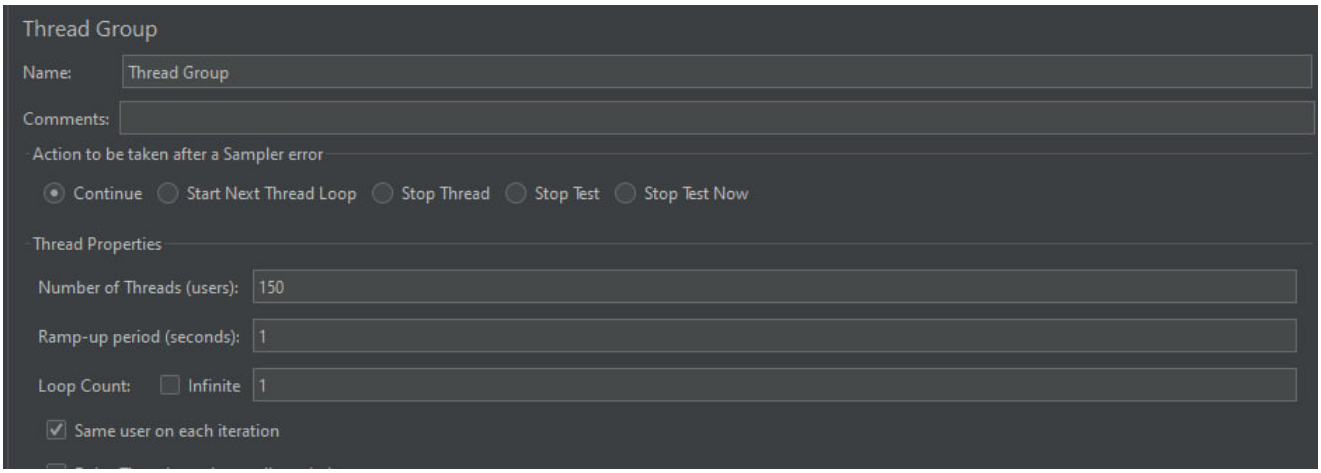


Рисунок 5.4 – Налаштування навантаження на 150 користувачів у JMeter (рисунок виконано самостійно)

Далі було проведено тестування ендпоінту для отримання усіх заходів. Оскільки події відображаються на головній сторінці одразу після входу в систему, саме цей запит виконується щоразу при відкритті головної сторінки. Тому перевірка його продуктивності є особливо важливою.

Для тестування було створено 100 подій із випадковими даними, щоб змоделювати реальне навантаження. Було налаштовано запит до ендпоінту /api/events/list/, який дозволяє користувачу отримувати перелік доступних заходів. Налаштування HTTP-запиту представлено на рисунку 5.5.

Після налаштування було запущено тестування, результати якого наведено на рисунку 5.6.

Середній час відповіді складає 352 мілісекунди, 0 відсотків помилок та пропускна здатність – 107.4 запитів за секунду. Ці результати свідчать про стабільну роботу системи навіть при значному навантаженні. Усі події були

успішно отримані та коректно відображались на головній сторінці, що підтверджує готовність цього ендпоінту до використання в реальному середовищі.

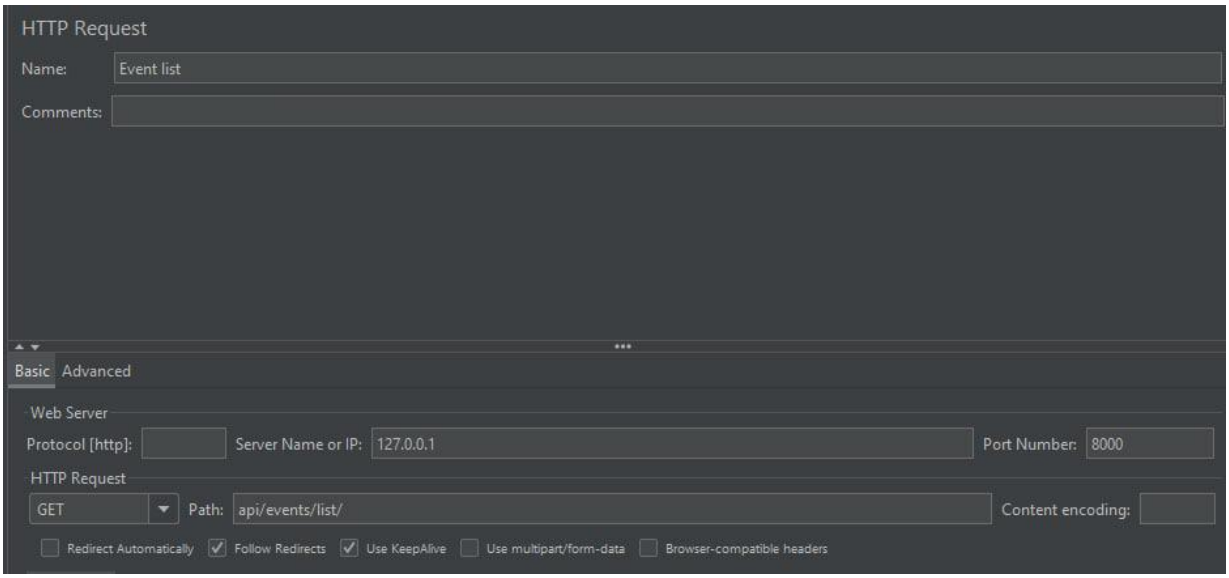
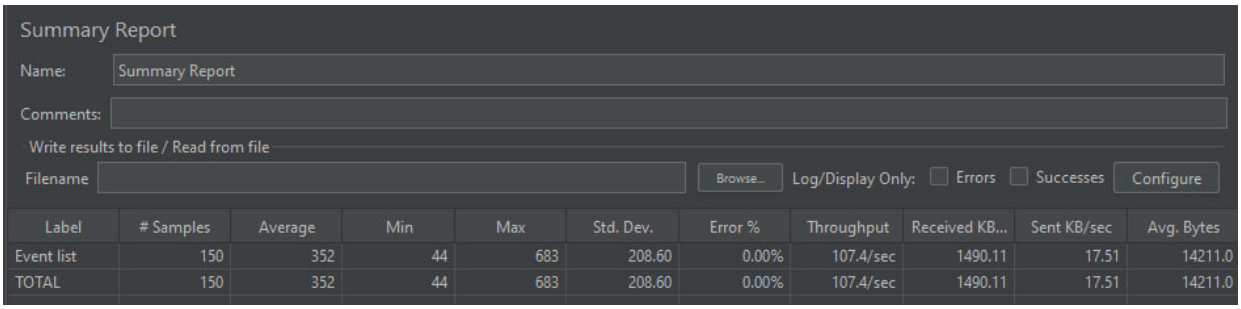


Рисунок 5.5 – HTTP-запит на отримання списку заходів (рисунок виконано самостійно)



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB...	Sent KB/sec	Avg. Bytes
Event list	150	352	44	683	208.60	0.00%	107.4/sec	1490.11	17.51	14211.0
TOTAL	150	352	44	683	208.60	0.00%	107.4/sec	1490.11	17.51	14211.0

Рисунок 5.6 – Результати тестування перегляду подій при 150 користувачах (рисунок виконано самостійно)

Далі був протестований ендпоінт для отримання списку книжок. Цей функціонал є одним із ключових у рамках роботи системи, адже саме з ним найчастіше взаємодіють користувачі, так як цей запит також виконується при відкритті головної сторінки. Для моделювання реального навантаження було згенеровано 100 книжок разом з усією необхідною інформацією про них. Тестування проводилось із 150 одночасними потоками, що було попередньо налаштовано у Thread Group. Налаштування GET-запиту до ендпоінту /api/books/list/ наведено на рисунку 5.7.

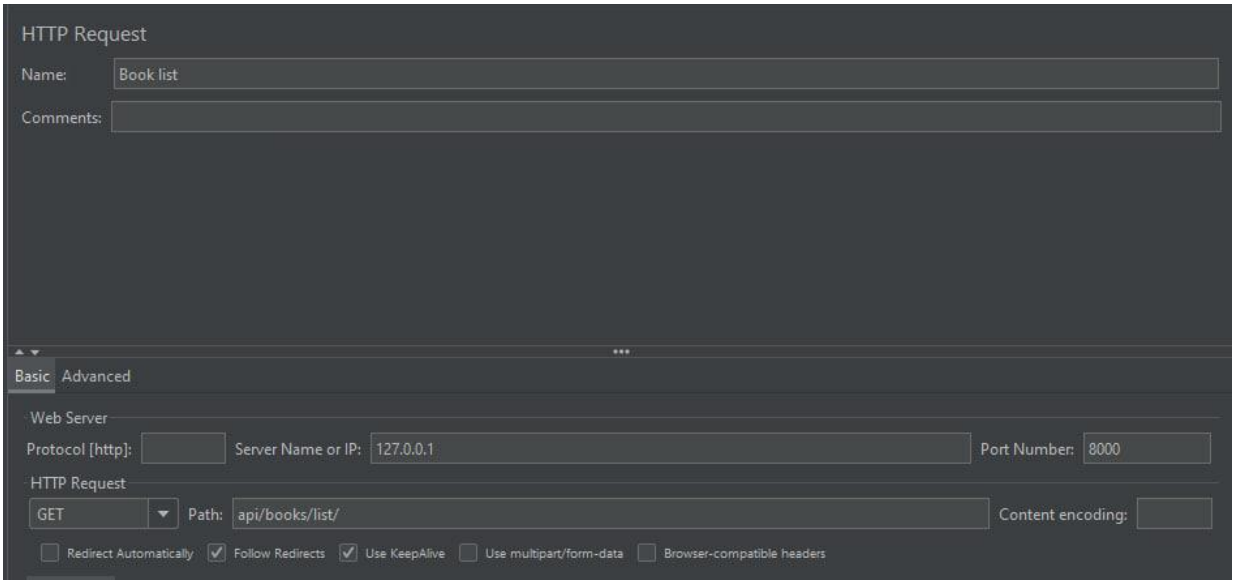
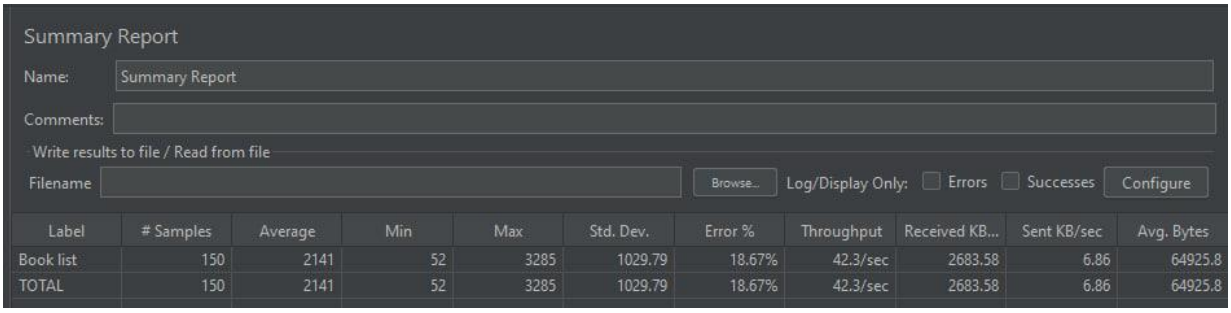


Рисунок 5.7 – Налаштування запиту на отримання списку книжок (рисунок виконано самостійно)

Після налаштування було запущено тестування, результати якого наведено на рисунку 5.8.

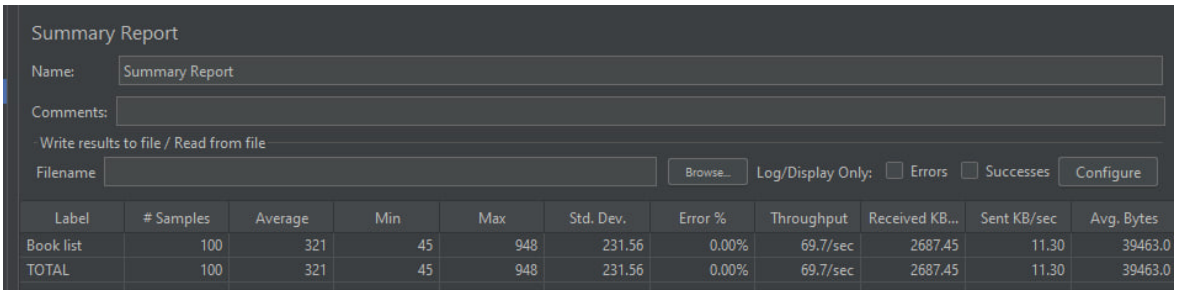


Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB...	Sent KB/sec	Avg. Bytes
Book list	150	2141	52	3285	1029.79	18.67%	42.3/sec	2683.58	6.86	64925.8
TOTAL	150	2141	52	3285	1029.79	18.67%	42.3/sec	2683.58	6.86	64925.8

Рисунок 5.8 – Результати перегляду книжок при 150 користувачах (рисунок виконано самостійно)

Середній час відповіді складає 2141 мілісекунду, 18 відсотків помилок та пропускна здатність – 42.3 запити за секунду. Надмірна кількість одночасних запитів призвела до помітного зниження продуктивності системи та збільшення кількості помилок. Ймовірно, це пов’язано з тим, що під час обробки запиту завантажується повний список книжок, який містить значно більше інформації, ніж список заходів. Зокрема, для кожної книжки зберігаються додаткові атрибути – рік видання, автор, мова, жанр тощо, що створює додаткове навантаження на сервер.

Для перевірки оптимального навантаження тест було повторено з 100 користувачами. Результати значно покращились. Результати показано на рисунку 5.9.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB...	Sent KB/sec	Avg. Bytes
Book list	100	321	45	948	231.56	0.00%	69.7/sec	2687.45	11.30	39463.0
TOTAL	100	321	45	948	231.56	0.00%	69.7/sec	2687.45	11.30	39463.0

Рисунок 5.9 – Результати перегляду книжок при 100 користувачах (рисунок виконано самостійно)

Середній час відповіді складає 321 мілісекунду, 0 відсотків помилок та пропускна здатність – 69.7 запитів за секунду. Система показала стабільну роботу при меншому навантаженні, усі запити були оброблені коректно. Такий рівень продуктивності повністю відповідає очікуваному навантаженню для типової міської бібліотеки та гарантує комфортну взаємодію користувачів із системою.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було спроєктовано та реалізовано повнофункціональну веб-систему для управління міською бібліотекою. Розробка охоплювала як клієнтську частину (frontend), так і серверну (backend), що дозволило створити завершений продукт, готовий до використання.

На початковому етапі було проведено глибокий аналіз предметної області та існуючих рішень, таких як Koha, Evergreen та Alma. Головним недоліком більшості з них виявився складний і перевантажений інтерфейс, а також відсутність частини функцій, які є важливими для звичайних користувачів. Це дало підставу сформулювати вимоги до нової системи, яка б задовольняла як працівників бібліотеки, так і звичайних читачів.

У процесі реалізації було побудовано REST API за допомогою Django REST Framework з використанням JWT-автентифікації (SimpleJWT) для розмежування доступу між ролями: читач, бібліотекар, адміністратор. Створено чітку архітектуру маршрутизації, організовано окремі модулі для книг, користувачів, подій, бронювань і налаштувань системи.

Клієнтська частина розроблена з використанням бібліотеки React та стилізована за допомогою TailwindCSS і Material UI. Axios використано для асинхронної взаємодії з сервером. Інтерфейс відповідає сучасним стандартам UI/UX та адаптований для різних ролей.

Для перевірки коректності роботи API було проведено функціональне тестування за допомогою Postman, що дозволило перевірити правильність відповіді на основні HTTP-запити. Додатково, з метою оцінки навантажувальної витривалості, було застосовано Apache JMeter. За його допомогою змодельовано запити від великої кількості одночасних користувачів до ключових ендпоінтів – авторизації, перегляду книг та перегляду заходів. Результати підтвердили стабільну роботу бекенду під навантаженням, що свідчить про готовність системи до практичного використання.

Таким чином, розроблений веб-застосунок є стабільним, масштабованим і орієнтованим на реальні потреби сучасної міської бібліотеки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Official Website of Koha Library Software. Official Website of Koha Library Software. URL: https://koha-community.org/ (дата звернення: 11.04.2025).

2. Evergreen ILS. Evergreen ILS. URL: https://evergreen-ils.org/ (date of access: 12.05.2025).

3. Evergreen - library software - LinuxLinks. LinuxLinks. URL: https://www.linuxlinks.com/evergreen/ (дата звернення: 11.04.2025).

4. Using the Public Access Catalog :: Evergreen Documentation. Redirect Notice. URL: https://docs.evergreen-ils.org/docs/latest/opac/using_the_public_access_catalog.html (дата звернення: 11.04.2025).

5. Library Management System : Alma - Ex Libris. Ex Libris. URL: https://exlibrisgroup.com/products/alma-library-services-platform/ (дата звернення: 11.04.2025).

6. Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability. Pearson Education, 2014. 216 p.

7. Faceted search: 9 best practices to improve UX and conversions. FactFinder blog. URL: https://www.fact-finder.com/blog/faceted-search/ (дата звернення: 15.04.2025).

8. Home - Django REST framework. Home - Django REST framework. URL: https://www.django-rest-framework.org/ (дата звернення: 15.04.2025).

9. Simple JWT – Simple JWT 5.2.2.post30+gfaf92e8 documentation. Simple JWT – Simple JWT 5.2.2.post30+gfaf92e8 documentation. URL: https://django-rest-framework-simplejwt.readthedocs.io/ (дата звернення: 15.04.2025).

10. Use case diagrams are UML diagrams describing units of useful functionality (use cases) performed by a system in collaboration with external users (actors). Unified Modeling Language (UML) description, UML diagram examples, tutorials and reference for all types of UML diagrams - use case diagrams, class, package, component, composite structure diagrams, deployments, activities, interactions, profiles, etc. URL: https://www.uml-diagrams.org/use-case-diagrams.html (дата звернення: 19.04.2025).

11. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. 560 p.

12. Architectural Styles and the Design of Network-based Software Architectures. Home - UC Irvine Donald Bren School of Information & Computer Sciences. URL: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата звернення: 20.04.2025).

13. Understanding ACID Compliance in PostgreSQL | Timescale. PostgreSQL ++ for time series and events | Timescale. URL: https://www.timescale.com/learn/understanding-acid-compliance (дата звернення: 25.04.2025).

14. Models | Django documentation. Django Project. URL: https://docs.djangoproject.com/en/stable/topics/db/models/ (дата звернення: 20.05.2025).

15. Postman: The World's Leading API Platform | Sign Up for Free. Postman API Platform. URL: https://www.postman.com/ (дата звернення: 23.05.2025).

16. Apache JMeter - Apache JMeter™. Apache JMeter - Apache JMeter™. URL: https://jmeter.apache.org/ (дата звернення: 28.05.2025).

17. Посилання на код програми на GitHub. GitHub. URL: https://github.com/NureRiabkoVladyslav/2025_B_PI_PZPI-21-9_Riabko_V_A (дата звернення: 02.06.2025).