НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

Комп'ютерних наук

_____Голуб Б.Л.

" " 2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему

«Програмне забезпечення інтернет магазину електронних книг»

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми

к.т.н., доцент _____ / Вайганг Г.О. /
(Науковий ступень та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи

к.т.н., доцент _____ Бородкіна І.Л.
(Науковий ступень та вчене звання) (підпис) (ПІБ)

Виконав _____ Рудківський І. А.
(підпис) (ПІБ)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерних наук
_____Голуб Б.Л.
_" "_____20 р._

З А В Д А Н Н Я

на виконання бакалаврської кваліфікаційної роботи студенту

Рудківському Іллі Андрійовичу

Спеціальність 121 – «Інженерія програмного забезпечення»

Тема бакалаврської кваліфікаційної роботи: «Програмне забезпечення інтернет магазину електронних книг»

Затверджена наказом ректора НУБіП України від 16.12.2024 № 2249 «С».

Термін подання завершеної роботи на кафедру _____

Вихідні дані до бакалаврської кваліфікаційної роботи: документація бібліотек, нормативні та технічні документи

Перелік питань , які потрібно розробити: проаналізувати предметну область, аналоги, розробити вимоги до системи, спроектувати та розробити систему

Дата видачі завдання "_____"_____20___ р.

Керівник бакалаврської кваліфікаційної роботи

к.т.н., доцент_____ _____ ___Бородкіна І.Л. _

(науковий ступінь та вчене звання) (підпис) (ПІБ)

Завдання прийняв до виконання _____ Рудківський І. А.

(підпис) (ПІБ студента)

ЗМІСТ

ВСТУП

Історично склалося так, що основним джерелом передачі знань для людства тривалий час залишалися друковані книги. Через сторінки паперових томів від покоління до покоління передавалася не лише інформація, а й культура, досвід, наукові досягнення. Проте реалії XXI століття кардинально змінили ситуацію: більшість людей тепер звертаються до інтернету, коли шукають відповіді на свої питання чи хочуть щось прочитати. Сучасні технології зробили доступ до знань майже миттєвим — достатньо лише смартфона та підключення до мережі.

Усе це призвело до того, що паперові книги поступово поступаються місцем електронним аналогам. Електронні книги поєднують у собі зручність традиційного читання та переваги цифрових рішень. До переваг електронних книг відноситься те, що:

1. Вони не займають фізичного простору
2. Вони завжди під рукою
3. Вони часто дешевші за фізичні
4. Вони екологічні
5. Вони доступні завдяки онлайн-платформам

Популярність електронних книг зростає з кожним роком, так само як і потреба в сервісах для їх збереження, організації та розповсюдження. З цієї причини виникає потреба у створенні програмного забезпечення, яке б відповідало цим новим викликам і забезпечувало ефективну роботу з цифровими книжковими ресурсами. Такі системи мають бути не лише зручними для користувачів, а й дозволяти підприємствам автоматизувати та оптимізувати свої внутрішні процеси, пов'язані з роботою з електронними виданнями.

Мета дослідження полягає в розробці програмного продукту, який би відповідав тематиці електронних книг, технічним вимогам і стандартам, що застосовуються у сфері сучасного програмування.

Об’єктом дослідження виступає інформаційна система, орієнтована на роботу з електронними книгами.

Предмет дослідження – методи та засоби, які використовуються для створення такого програмного забезпечення.

Основні завдання:

1. Виявлення основних проблем у предметній області.
2. Аргументація вибору технологій та інструментів для реалізації системи.
3. Формулювання технічного завдання.
4. Аналіз функціональних вимог і структури майбутньої системи.
5. Проєктування архітектури програмного забезпечення.
6. Безпосередня розробка та впровадження сервісу.

Розроблений програмний додаток має прикладну цінність завдяки своїм численним перевагам. Він забезпечує:

1. Зручність використання;
2. Доступ до книг у будь-який момент;
3. Великий вибір літератури;
4. Простий інтерфейс керування;
5. Можливість читати на різних пристроях;
6. Підтримку численних форматів електронних видань.

В якості середовища для розробки використовується Visual Studio та мову програмування C#, це IDE від компанії Microsoft, яка надає весь необхідний функціонал для створення програм різного рівня.

За сервер бази даних відповідає Postgres, це зручна та надійна база даних, яка надає різноманітні функції, яких немає у аналогів. Також вона має відкритий код і безкоштовну версію, яка повністю задовольняє більшість потреб.

Для зв'язку з базою даних використовується Entity Framework Core. Entity Framework Core - це фреймворк для розробки програмного забезпечення, який забезпечує зв'язок між даними в додатку та базою даних.

Він є частиною .NET Core і дозволяє легко і зручно працювати з реляційними базами даних. Це потужний інструмент для забезпечення зв'язку з базою даних, який дозволяє легко та швидко розробляти додатки.

Для розробки серверної частини застосовується фреймворк Asp.Net core. Створений Microsoft для розробки сучасних застосунків які працюють з інтернетом. До переваг даного фреймворку відносять кросплатформенність та високу продуктивність.

Клієнтська частина використовує такі технології як HTML, CSS, JavaScript. Вони працюють разом і дозволяють створювати красиві, зручні та динамічні веб сторінки. HTML відповідає за розмітку сторінки, CSS – за зовнішній вигляд та стилі, JavaScript – за поведінку та динаміку.

У межах цієї кваліфікаційної роботи буде здійснено повний цикл створення інформаційної системи — від аналізу вимог до реалізації функціонального сервісу. Розроблене рішення повинно відповідати технічному завданню, бути сучасним, зручним та відповідати актуальним стандартам у галузі програмної інженерії.

РОЗДІЛ 1

1.1. Опис предметної області

На сьогоднішній день, електронні джерела поступово витісняють аналогові способи отримання інформації, які домінували в минулі епохи. Зокрема, електронні книги набули значного поширення, що пов’язано з активним використанням сучасних пристроїв: смартфонів, комп’ютерів, планшетів тощо. У зв’язку з цим виникає потреба у спеціалізованих системах зберігання таких книг, які здатні вирішувати численні задачі, пов’язані з їх обробкою та розповсюдженням.

Однією з головних проблем, яка потребує вирішення, є зростаюча кількість контенту: щороку видається все більше літератури, з’являються нові автори й видавництва, що ускладнює управління інформаційними масивами. Ефективна система, яка дозволяє централізовано керувати контентом, стає необхідністю, особливо для розповсюджувачів літератури.

Водночас важливим аспектом є доступність. Існує велика кількість користувачів, які не мають фізичної можливості користуватись друкованими виданнями, тому електронний формат для них є не лише зручним, а й єдиним способом ознайомлення з літературою. Зручний інтерфейс системи, адаптація форматів до різних екранів та пристроїв створює комфорт для читача, подібний до того, що забезпечують паперові книги.

Не менш актуальною є проблема захисту авторського права. В умовах цифрової доби електронне піратство набуває загрозливих масштабів, через що багато держав на законодавчому рівні запроваджують санкції та обмеження. Тому програмне рішення повинно забезпечувати належний рівень захисту інтелектуальної власності — від контролю за розповсюдженням контенту до збереження інформації про власників прав.

Таким чином, створення інформаційної системи для електронних книг дозволить не лише забезпечити комфортне користування літературним контентом, а й вирішити питання зберігання, доступу, захисту авторських прав

та автоматизації управління цифровими матеріалами. Саме така система відповідає вимогам часу та викликам цифрової епохи.

1.2. Огляд інформаційних джерел та існуючих рішень

У процесі розробки інформаційної системи продажу електронних книг важливо провести аналіз існуючих рішень, що вже представлені на ринку. Це дозволяє виявити сильні та слабкі сторони аналогічних систем, визначити ключові функціональні вимоги, а також уникнути дублювання наявних функцій та зосередитися на удосконаленні або інноваціях.

Першим кроком було розглянуто інтернет магазин «Amazon Kindle Store». Це інтернет магазин електронних книг від компанії Amazin. На рисунку 1.1. представлено скріншот цього сервісу.

Рис. 1.1. Скріншот роботи сервісу «Amazon Kindle Store»

Це один з найбільших у світі онлайн-магазин електронних книг, інтегрований з екосистемою Kindle. До його функціональних можливостей відноситься купівля, завантаження, зберігання та читання книг через пристрої Kindle або мобільні застосунки. Також присутня підписка Kindle Unlimited.

Що стосується переваг цього сервісу, до них можна віднести високу популярність та високий рівень довіри користувачів, потужну систему рекомендацій, а також підтримку DRM-захисту. До недоліків відноситься закрита екосистема, що означає підтримку лише формату Kindle, а також високу конкуренцію серед авторів.

Наступним кроком було розглянуто платформу Google Play Books. За допомогою цього сервісу можна завантажувати електронні книги й читати їх на кількох пристроях. Також можна додавати файли й друкувати книги. Скріншот роботи сервісу наведено на рисунку 1.2.

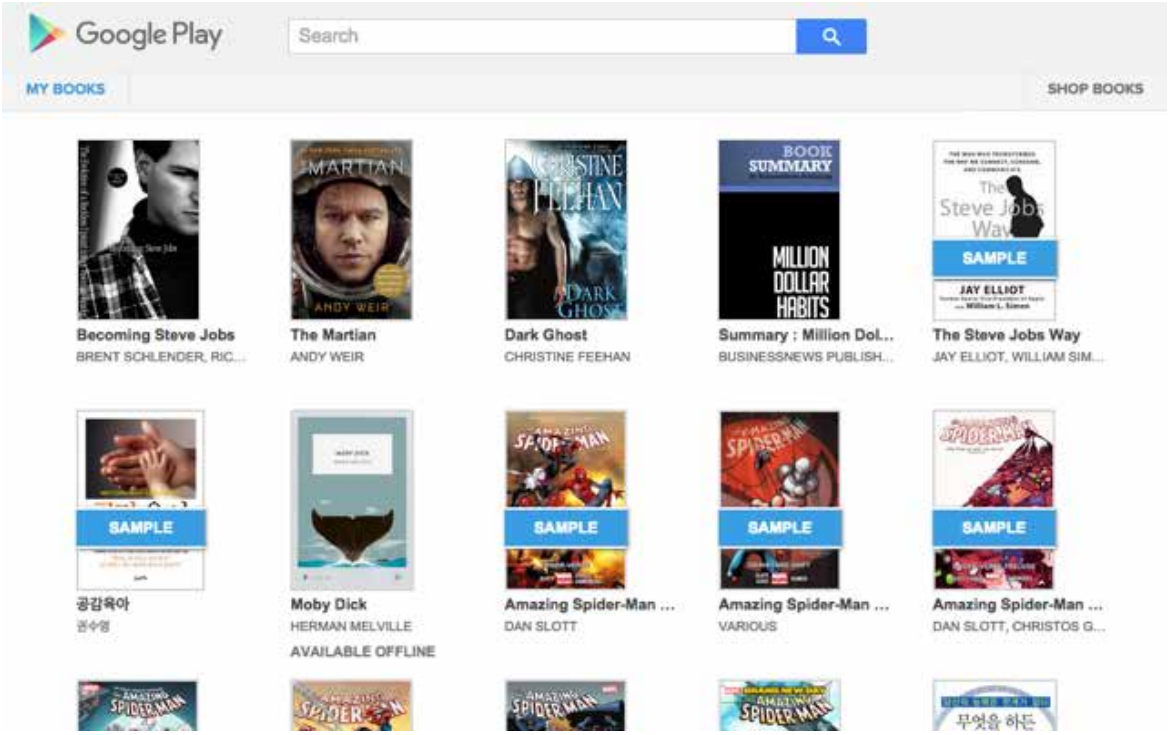

Рис. 1.2. Скріншот роботи сервісу «Google Play Books»

В цьому сервісу можна купувати електронні книги напряму або проводити читання завантажених PDF/EPUB-файлів. Також присутня можливість хмарного зберігання власних книг та їх завантаження. До переваг цієї системи відноситься широка інтеграція з обліковим записом Google і, відповідно, з екосистемою Google, а також синхронізація між пристроями. Цільова аудиторія це користувачі Android та Google-екосистеми.

Третім сервісом під час аналізу став «Kobo Books». Kobo Books — це популярна онлайн-платформа для придбання цифрових книг і аудіоконтенту,

яка пропонує великий вибір літератури на будь-який смак. У каталозі сервісу представлені твори художньої літератури, наукові видання, біографії, книги для дітей та багато інших жанрів. На рисунку 1.3. представлено скріншот роботи даного сервісу.

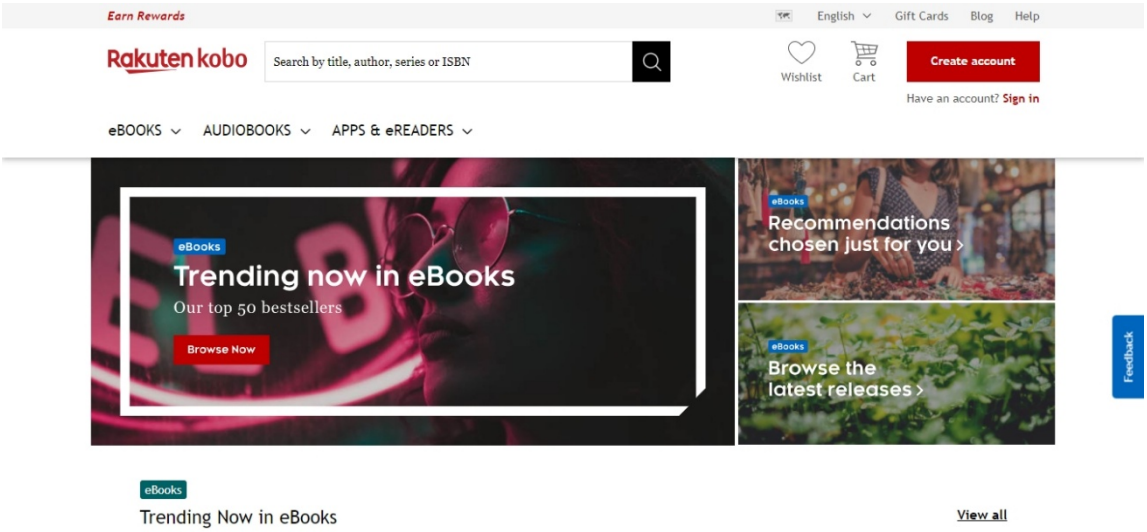

Рис. 1.3. Скріншот роботи сервісу «Kobo Books»

Однією з головних переваг Kobo є висока якість доступного контенту та багатомовна підтримка, включаючи наявність книжок українською мовою. Завдяки партнерству з провідними авторами та видавництвами з різних країн світу, користувачі мають змогу першими отримувати доступ до сучасних літературних новинок.

Після аналізу програм аналогів для даної предметної області було зроблено порівняння та наведено у таблиці 1.1.

Таблиця 1.1. Порівняльна таблиця платформ продажу електронних книг

Критерій	**Amazon Kindle**	**Google Play Books**	**Kobo Books**	**Smashwords**
Підтримувані формати	AZW, MOBI (Kindle only)	EPUB, PDF	EPUB, MOBI, PDF	EPUB, MOBI, PDF
Платформа для авторів	Так (Kindle Direct Publishing)	Ні (через сторонні сервіси)	Так (Kobo Writing Life)	Так (безкоштовна публікація)
DRM-захист	Так	Так	Частково	Ні (опціонально)

Мовна підтримка (вкл. укр.)	Обмежена	Так	Частково (є українські книги)	Залежить від автора
Офлайн-доступ до книг	Так	Так	Так	Так
Синхронізація між пристроями	Так (через Kindle-акаунт)	Так (через Google-акаунт)	Так	Частково
Поширення через інші магазини	Ні	Ні	Обмежено	Так (Apple Books, B&N, ін.)
Доступність застосунків	Android, iOS, ПК, Kindle	Android, iOS, браузер	Android, iOS, eReader	Браузер, EPUB-читачі
Інтеграція API	Обмежено	Частково (через Google API)	Немає	Немає
Можливість самостійного завантаження книг	Ні	Так	Так	Так

1.3. Аналіз вимог до програмної системи

Після аналізу програм аналогів та створення порівняльної таблиці, наступним кроком є аналіз та оформлення списку вимог до проектованої системи.

«Вимоги до ПЗ – це специфікація того що повинно бути реалізовано. Це описи поведінки системи, властивості системи або її атрибути. Вони можуть бути обмеженні процесом розробки системи» [1].

В загальному випадку можна виділити 3 рівні вимог: бізнес вимоги, функціональні та нефункціональні вимоги.

Бізнес-вимоги — це стратегічні наміри або ключові очікування організації чи замовника, які відображають загальну мету впровадження програмного забезпечення на високому рівні абстракції.

Вимоги користувача призначені для визначення того, яку функціональність має забезпечувати програмне забезпечення з точки зору

кінцевого користувача. У той час як нефункціональні вимоги описують обмеження, характеристики та умови, за яких ця функціональність має бути реалізована — наприклад, швидкодія, масштабованість, зручність у використанні або рівень безпеки системи.

1.3.1. Функціональні вимоги

- Програма має можливість створення користувачів на основі ролей.
- Програма має можливість розмежування прав доступу до даних та операцій з даними на основі ролей адміністратора та користувача.
- Адміністратор системи має можливість вносити, редагувати та видаляти дані про книги.
- Адміністратор системи має можливість переглядати дані інших користувачів.
- Адміністратор системи має можливість переглядати дані замовлень, детальної інформації про замовлення, такої як статус, користувач, дата, тощо.
- Адміністратор системи має можливість переглядати звітні дані в вигляді таблиць, графіків, тощо.
- Користувач системи має можливість переглядати наявні продукти.
- Користувач системи має можливість сортування та фільтрації продуктів за необхідними йому змінними, такими як мова книги, автор, видання, тощо.
- Користувач системи має пошуку необхідної книги за приблизною назвою.
- Користувач системи має можливість реєстрації в системі з введенням своїх даних.
- Користувач системи має можливість аутентифікації в системі на основі даних які він використовував під час реєстрації.
- Користувач має можливість додавання, видалення даних з своєї корзини продуктів.
- Користувач має можливість додавання та видалення продуктів зі списку бажаного.

- Користувач має можливість створювати рецензії на книги, редагувати їх, видаляти за необхідності, також переглядати рецензії інших користувачів.
- Користувач має можливість оцінювати книгу по шкалі від 1 до 5, що буде мати вплив на загальний рейтинг книги.
- Користувач має можливість створити замовлення на основі своєї корзини покупок, також редагувати, додавати, видаляти елементи.
- Користувач має можливість отримати доступ до безкоштовної версії книги, яка має бути обмежена по кількості сторінок.
- Користувач має можливість отримати доступ до повної версії книги після виконання замовлення.
- Користувач має можливість переглядати свою історію рецензій, замовлень, списку бажаного.

1.3.2. Нефункціональні вимоги

- Система повинна забезпечувати завантаження сторінки з переліком книг за не більше ніж 2 секунди при середньому навантаженні.
- Час відповіді на запит пошуку книги не повинен перевищувати 1 секунди при середньому навантаженні.
- Архітектура має дозволяти горизонтальне масштабування для забезпечення більшого навантаження в майбутньому.
- Система повинна бути доступною не менше ніж 99.5% часу впродовж календарного місяця.
- У випадку збою система повинна автоматично відновлювати доступ до критичних функцій протягом 5 хвилин.
- Доступ до системи повинен здійснюватися через безпечний протокол HTTPS.
- Дані користувача, включаючи паролі, мають зберігатися в зашифрованому вигляді (наприклад, за допомогою bcrypt).
- Система повинна забезпечувати захист від типових атак: SQL-ін’єкцій, XSS, CSRF тощо.

- Має бути реалізований механізм автентифікації і авторизації з обмеженням доступу до функцій за ролями.
- Інтерфейс користувача повинен бути інтуїтивно зрозумілим
- Система повинна коректно працювати в основних браузерах: Chrome, Firefox, Edge, Safari (останні 2 версії).
- Дані користувача не повинні передаватися третім сторонам без згоди.
- Має бути реалізовано механізм видалення облікового запису та всіх пов’язаних із ним даних на вимогу користувача.

1.4. Моделювання предметної області

Моделювання предметної області є важливою частиною розробки програмної системи. В даному випадку, моделювання буде проводитися з використанням певної кількості різних діаграм, серед яких значна частина є UML діаграмами.

«UML (Unified Modeling Language) – уніфікована мова моделювання – це система позначень, що застосовується для об'єктно-орієнтованого аналізу та проектування, це мова діаграм або позначень для специфікації, візуалізації та документації моделі об'єктно-орієнтованих програмних систем. UML не є методом розробки, тобто він не визначає послідовність дій при розробці ПЗ. Він допомагає описати свою ідею і взаємодіяти з іншими розробниками системи. UML управляється Object Management Group (OMG) і є промисловим стандартом, що описує моделі ПЗ» [3].

Моделювання предметної області доцільно почати з виділення основних абстракцій присутній в системі. На рисунку 1.4. представлено основні абстракції системи. Абстракція «Книга» зберігає всі властивості книги, такі як назва, автори, жанр, опис, формат, ціна, тощо. Ця абстракція відповідає на отримання даних певної книги та перевірки її наявності. Наступним кроком, було виділено абстракцію «Користувач». Користувач має всі необхідні властивості для користувача системи, такі як ім’я, прізвище, пароль, роль,

тощо. Користувач може реєструватися, авторизуватися, переглядати каталог книг та купувати книги.

Далі було виділено такі абстракції як каталог, замовлення, платіж та відгук. Кожна з цих абстракцій має свою зону відповідальності. Абстракція «Замовлення» відповідає за характеристики та обов'язки замовлень в системі, в той час, як платіж, відповідає за відповідну сутність в системі. На основі цих абстракцій буде будуватися майбутня система.

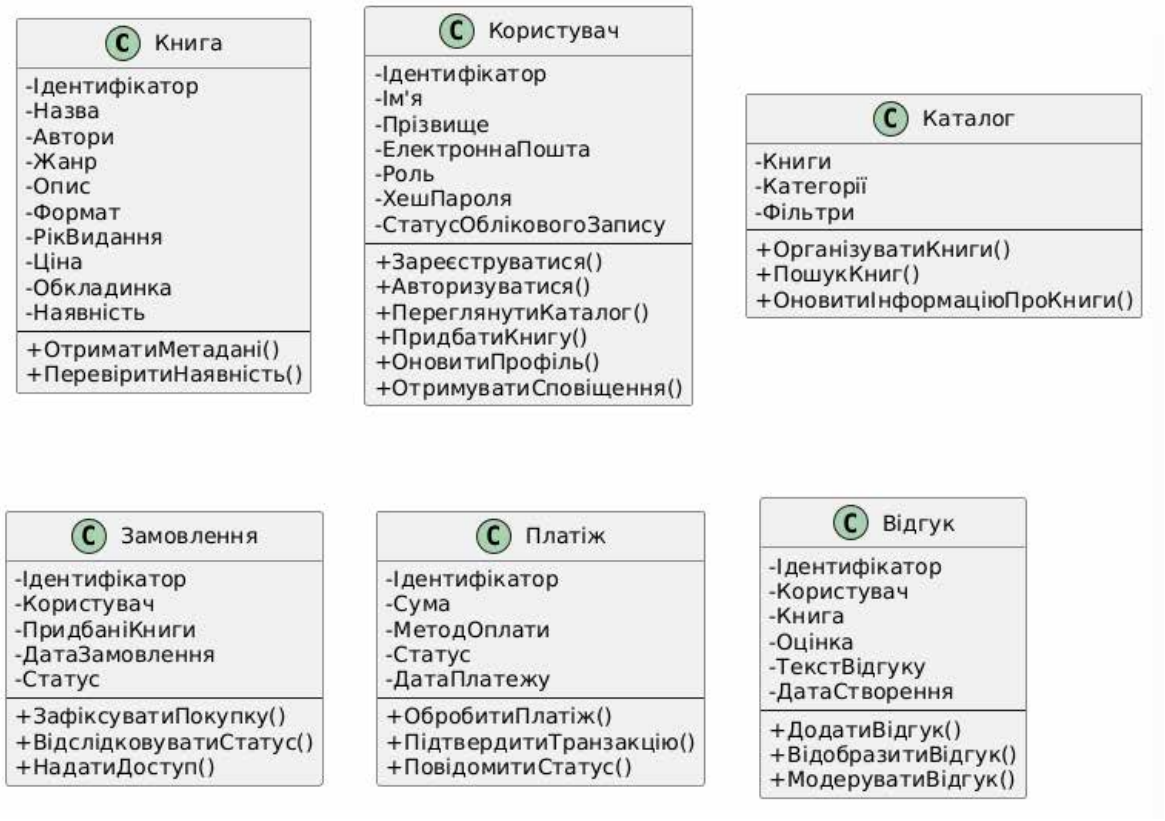

Рис. 1.4. Основні абстракції системи

Одним із найбільш поширених та дієвих підходів до покращення зрозумілості та чіткості вимог є їх представлення у формі сценаріїв використання (use case), що було запропоновано Іваром Якобсоном.

«Діаграма прецедентів (use case diagram) – це графічне представлення всіх або частини акторів, прецедентів та їх взаємодій в системі. У кожній системі зазвичай є головна діаграма прецедентів, яка відображає межі систем (акторів) та основну функціональну поведінку системи (прецедентів). Інші діаграми прецедентів можуть створюватися при необхідності» [3].

На рисунку 1.5. знаходиться діаграма прецедентів для актора «Клієнт». Як можна помітити, клієнт може виконувати такі операції, як: керування списком бажаного, замовлення то оцінка товару, перегляд та скачування книг, керування кошиком та особистим кабінетом. Деякі прецеденти мають включення, наприклад прецедент «Керування списком бажаного» має в собі додавання, видалення та перегляд елементів. Схожим чином працюють і інші прецеденти.

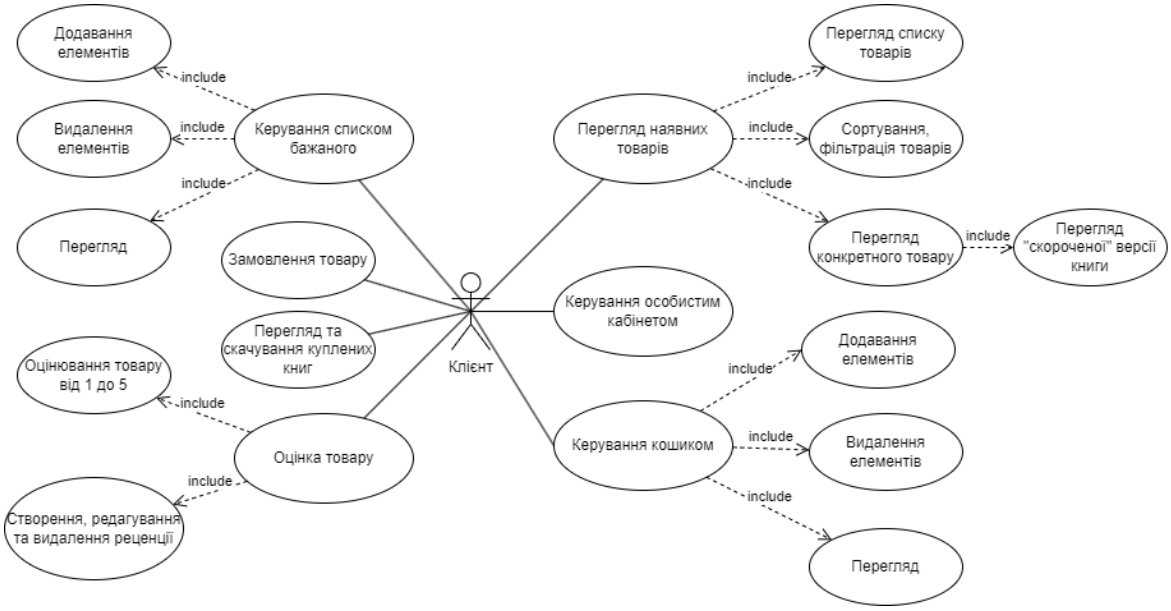

Рис. 1.5. Діаграма прецедентів для актора «Клієнт»

На рисунку 1.6. зображено діаграму прецедентів для «Адміністратора». Адміністратор може виконувати такі операції над товарами: додавання, редагування, видалення та перегляд. Також адміністратор може переглядати списки клієнтів, переглядати історії замовлень і формувати звітну документацію.

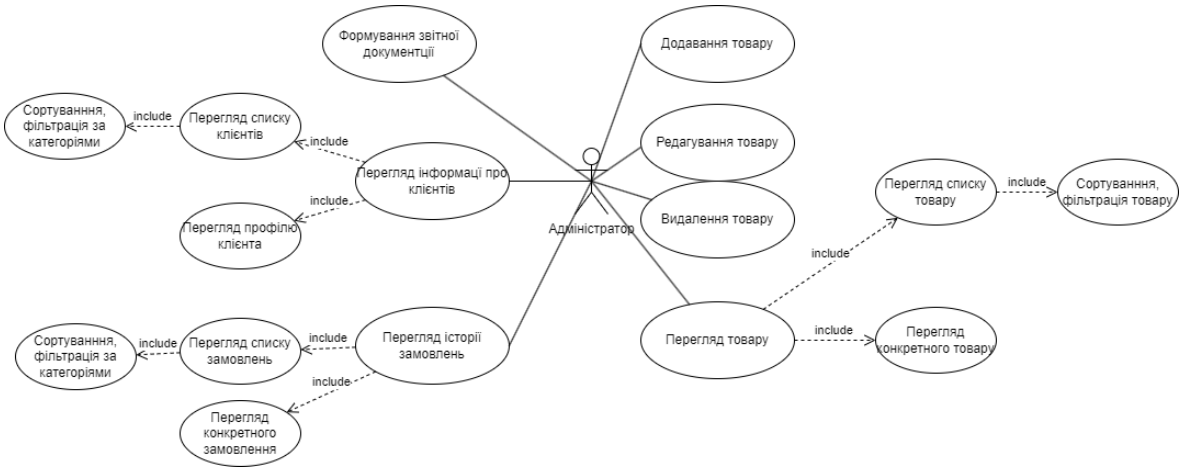

Рис. 1.6. Діаграма прецедентів для актора «Адміністратор»

Наступним кроком, після аналізу акторів та прецедентів необхідно перейти до аналізу послідовності системи. В цьому допоможе діаграма послідовності.

«На діаграмі послідовності зображуються ті об'єкти, що безпосередньо беруть участь у взаємодії і не показуються можливі статичні асоціації з іншими об'єктами. Діаграма має два виміри. Один у напрямку з лівої сторони праворуч у вигляді вертикальних ліній, кожна з яких зображує лінію життя окремого об'єкта, що приймає участь у взаємодії. Крайнім ліворуч на діаграмі зображується об'єкт, що є ініціатором взаємодії» [2].

На рис. 1.7. зображено діаграму послідовності, це діаграма на якій показані взаємодії об'єктів, упорядковані за часом їхнього прояву. На даній діаграмі можна побачити взаємодії користувача з системою, то що відбувається «за лаштунками».

Кожен компонент системи надсилає запит і повертає відповідь, яка в кінці надходить до користувача і дозволяє далі користуватися системою. Саме в даному випадку продемонстрована послідовності вибору книги, оформлення замовлення та оплату, та всі необхідні компоненти системи які використовуються в цій послідовності.

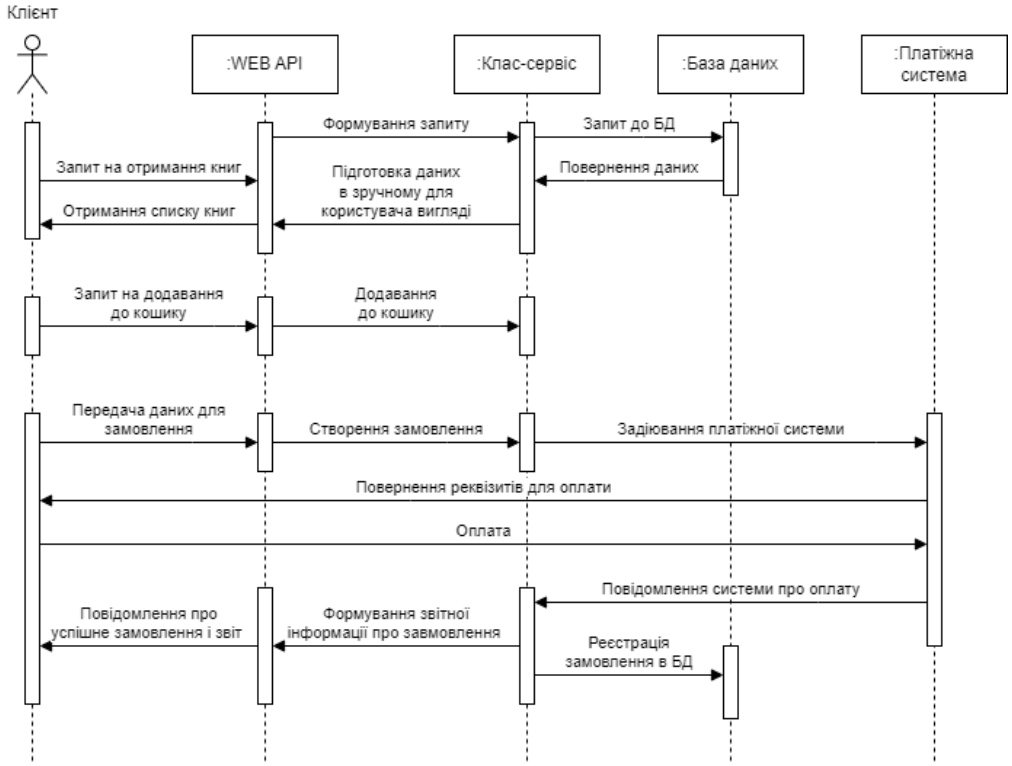

Рис. 1.7. Діаграма послідовності до проектованої системи

1.5. Висновки до розділу 1

Під час роботи над першим розділом було досліджено та описано предметну область. Зроблено висновок, що однією з головних проблем, яка потребує вирішення, є зростаюча кількість контенту, отже ефективна система, яка дозволяє централізовано керувати контентом, стає необхідністю, особливо для розповсюджувачів літератури.

Було розглянуто та описано існуючі програми аналоги та на основі їх аналізу та аналізу потреб користувачів розроблено функціональні та нефункціональні вимоги до розроблюваної системи.

Під час моделювання предметної області було продемонстровано діаграми прецедентів для акторів "Клієнт" та "Адміністратор". Ці діаграми відображають основних акторів та прецедентів існуючих в системі. Останнім кроком було продемонстровано діаграму послідовності, на якій показані взаємодії об'єктів, упорядковані за часом їхнього прояву. Таким чином, на основі цих напрацювань буде відбуватися подальша робота над системою.

РОЗДІЛ 2

6.1. Логічна модель даних у вигляді ER діаграми

Логічна модель даних — це абстрактне подання структури даних, яке відображає організацію інформації без прив’язки до конкретної фізичної реалізації. Одним із найбільш поширених та ефективних методів моделювання логічної структури бази даних є створення ER-діаграми (Entity-Relationship Diagram).

ER-діаграма дозволяє наочно й системно відобразити основні елементи предметної області: сутності, атрибути та зв’язки між ними. Вона сприяє глибшому розумінню структури даних, що є особливо важливим на етапі проектування інформаційних систем. Завдяки графічному представленню, ER-діаграма полегшує комунікацію між розробниками, аналітиками та замовниками, сприяючи точнішому визначенню вимог і уникненню непорозумінь. На рисунку 2.1. зображено логічну модель даних для проектованої системи у вигляді ER діаграми.

Кожна модель відображає всі необхідні характеристики для сутностей в системі, так, наприклад, користувач має ім’я, прізвище, пароль, роль, тощо, в свою чергу книга має назву, авторів, жанр, рік, опис, тощо. Детальніше слід розглянути типи зв’язків між сутностями. Зв'язок «Користувач-Замовлення» неідентифікуючий, адже замовлення може існувати лише у контексті конкретного користувача (замовник), проте ідентифікатор замовлення не залежить від ідентифікатора користувача (має власний первинний ключ).

Зв’язок «Замовлення-Книга» має ідентифікуючий тип зв’язку, тому, що одне замовлення може містити кілька книг, а одна книга може бути придбана у багатьох замовленнях. Тут необхідно зауважити, що це зв’язок «багато-до-багатьох», а отже при проектуванні фізичної моделі бази даних необхідно буде створити проміжну таблицю.

Зв’язок «Книга-Відгук» є неідентифікуючим, адже відгуки пишуться до конкретної книги, проте не визначають ідентичність книги. Книга може існувати незалежно від відгуків.

Зв’язок «Користувач-Відгук» має неідентифікуючий тип зв’язку. Це зумовлено тим, що відгук завжди має автора, але користувач не залежить від існування відгуку.

«Замовлення-Платіж» є ідентифікуючим, причина цьому, в тому, що, якщо кожне замовлення має один платіж, то це один-до-одного. Якщо дозволяються часткові платежі — один-до-багатьох. Платіж логічно не може існувати без замовлення, тому це ідентифікуючий зв’язок.

Таким чином, кожна таблиця має свій первинний ключ, всі поля в кожній таблиці залежать від первинного ключу і не існує транзитивних залежностей. На основі цього можна зробити висновок, що логічна модель даних до проектованої системи відповідає третій нормальній формі.

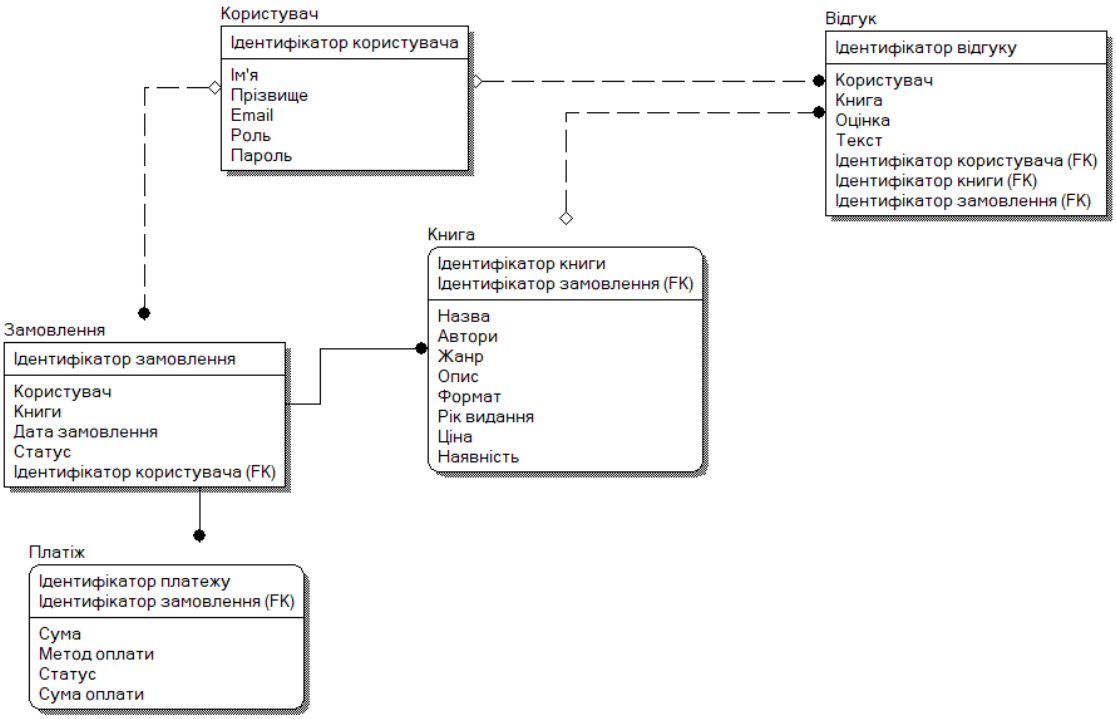

Рис. 2.1. Логічна модель даних у вигляді ER діаграми

6.2. Діаграма класів

Важливим етапом моделювання предметної області є створення діаграми класів.

Це діаграма, на якій показані класи, інтерфейси та відносини між ними. Головний елемент діаграми класів – клас. При проектуванн

об'єктноорієнтованих систем діаграми класів обов'язкові. «Класи використовуються в процесі аналізу предметної області для складання словника предметної області системи, що розробляється. Це можуть бути як абстрактні поняття предметної області, так і класи, на які спирається розробка та які описують програмні або апаратні сутності» [3].

Першим кроком під час моделювання класів було створено діаграму класів з використанням лише асоціацій. Такий підхід дозволить виділити всі основні сутності системи та відразу побачити взаємозв’язки між ними. На рисунку 2.2. продемонстровано діаграму класів з використанням асоціацій.

Користувач може купувати книги, це демонструється в зв’язку між користувачем та книгами. В свою чергу, розглядаючи зв’язки між сутностями «User», «Book» та «Review» можна помітити, що всі вони між собою з’єднані. Користувач може написати відгук на куплену книгу, відповідно книга може мати відгук. Це виражається в асоціативних зв’язках між цими сутностями.

Далі з діаграми можна помітити, що існує асоціативний зв’язок між користувачем та замовленням, а останній в свою чергу має зв’язок з платежем. Таким чином забезпечується функціональність, того, що користувач можу створювати та оплачувати замовлення.

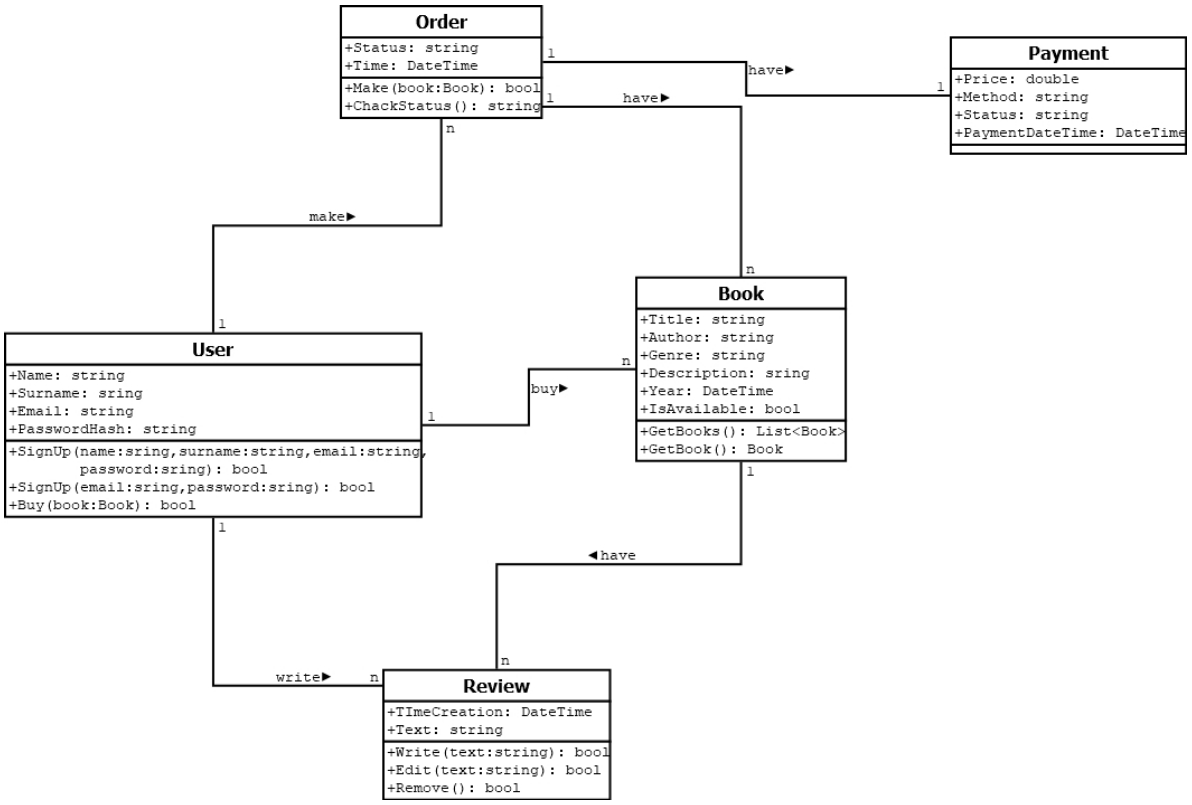

Рис. 2.2. Діаграма класів з використанням асоціацій

На основі побудованих раніше асоціацій в майбутньому буде будуватися діаграма класів, але варто зауважити, що вони можуть відрізнятися, адже попередня фокусується саме на моделях та їх взаємозв’язках в системі, в той час, як наступна діаграма уже буде відображати структуру класів та сервісів в системі.

На рисунку 2.3. знаходиться діаграма класів, яка відображає елементи, такі як: класи, типи даних, їх зміст та відношення. На діаграмі можна помітити, що всі класи-сервіси наслідують клас AbstractService. Це клас який представляє батьківський клас кожного класу-сервісу системи. Він містить в собі екземпляр поля бази даних, а також наслідує інтерфейс IRepository. Це означає, що він має визначити в собі всі методи, зазначені в інтерфейсі, а саме методи роботи з базою даних, такі операції як додати запис, видалити запис, редагувати запис. В свою чергу клас контролера використовують класи сервіси в якості своїх полів. Класи контролери це саме ті класи, які надають функціонал API.

Тобто, якщо ми будемо по API звертатися, наприклад, по адресу Books/Get. Це означає, що буде використовуватися Get метод класу контролера.

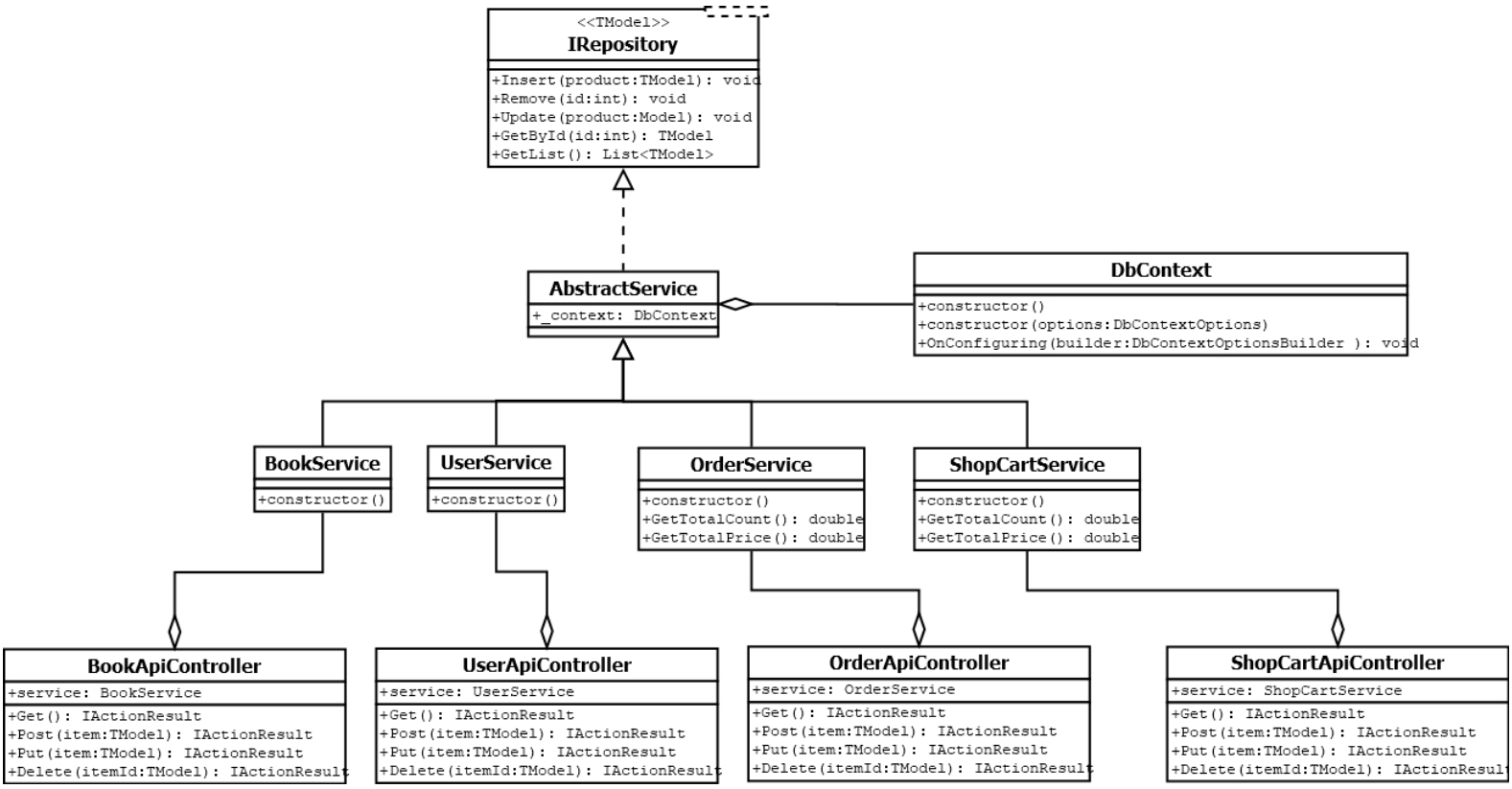

Рис. 2.3. Діаграма класів до проектованої системи

6.3. Діаграма станів

На рис. 2.4. зображено діаграму станів системи. Діаграма зображує стани системи в різний період часу. Спочатку система перебуває в стані аутентифікації, в якому користувач вводить дані, а система перевіряє їх і надає доступ до системи. Наступний стан пошук книги, після якого відбувається замовлення та оплата замовлення.

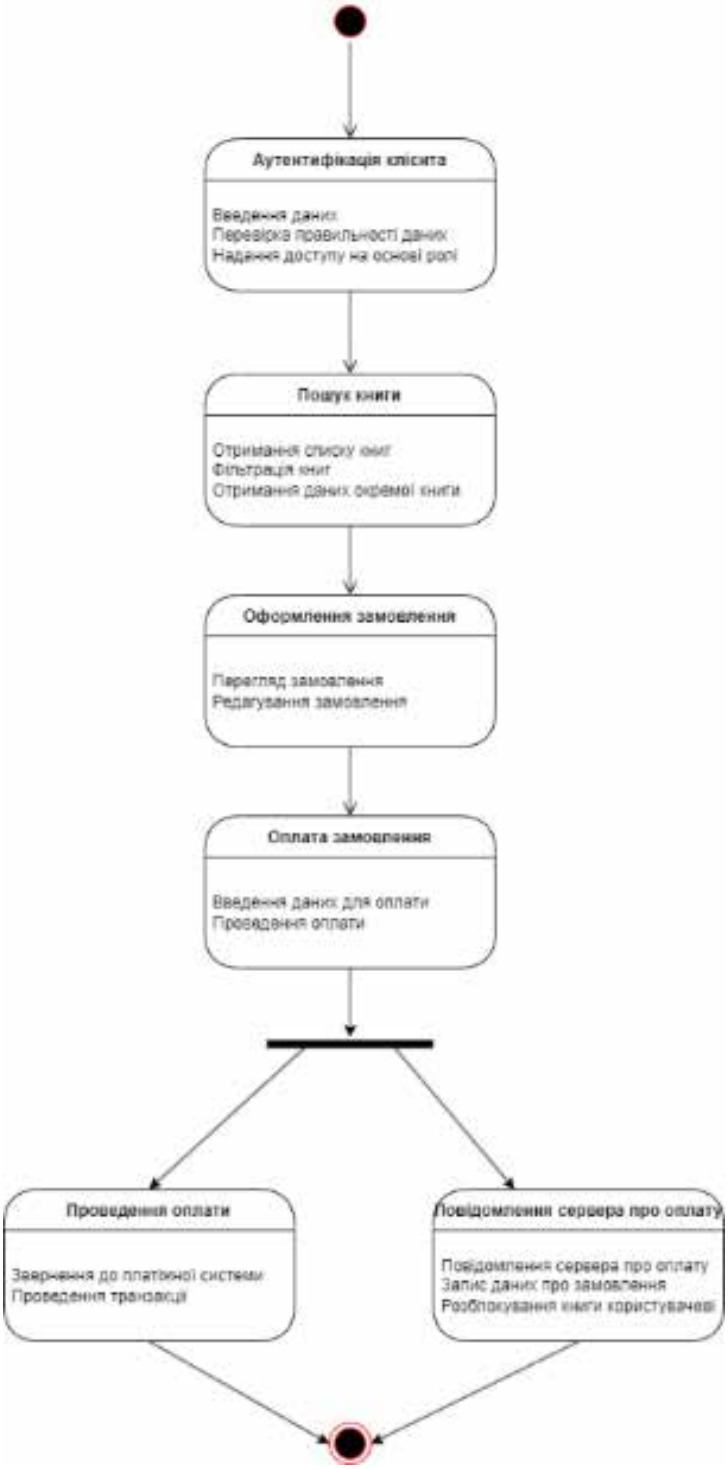

Рис. 2.4. Діаграма станів системи

6.4. Діаграма компонентів

«Діаграма компонентів служить частиною фізичного представлення моделі системи та грає важливу роль в процесі об'єктно-орієнтованого аналізу і проектування, є необхідною для генерації програмного коду. Вона показує різні компоненти системи і залежності між ними» [3].

На рисунку 2.5. представлено діаграму компонентів системи електронного магазину книг.

Кожен відображення залежить від відповідного сервісу, а саме:

1. BookView залежить від BookService, разом вони надають логіку та представлення роботи з сутністю книг.
2. OrderView залежить від OrderService, разом вони надають логіку та представлення роботи з сутністю замовлень.
3. WishListView залежить від WithlistService, разом вони надають логіку та представлення роботи з сутністю списку бажаних книг.
4. AuthView залежить від AuthService, разом вони надають логіку та представлення роботи з функціоналом авторизації.
5. ShopCartView залежить від ShopcartService, разом вони надають логіку та представлення роботи з сутністю кошику покупок.
6. PaymantView залежить від PaymantService, разом вони надають логіку та представлення роботи з функціоналом оплати замовлень.
7. DashboardView залежить від DashboardService, разом вони надають логіку та представлення роботи з функціоналом аналізу продаж.
8. ReviewView залежить від ReviewService, разом вони надають логіку та представлення роботи з сутністю коментарів.
9. UserView залежить від UserService, разом вони надають логіку та представлення роботи з сутністю користувачів.

Кожен сервіс в свою чергу залежить від класу взаємодії зі сховищем даних, а саме DatabaseClass, який в свою чергу залежить напряму від сховища даних.

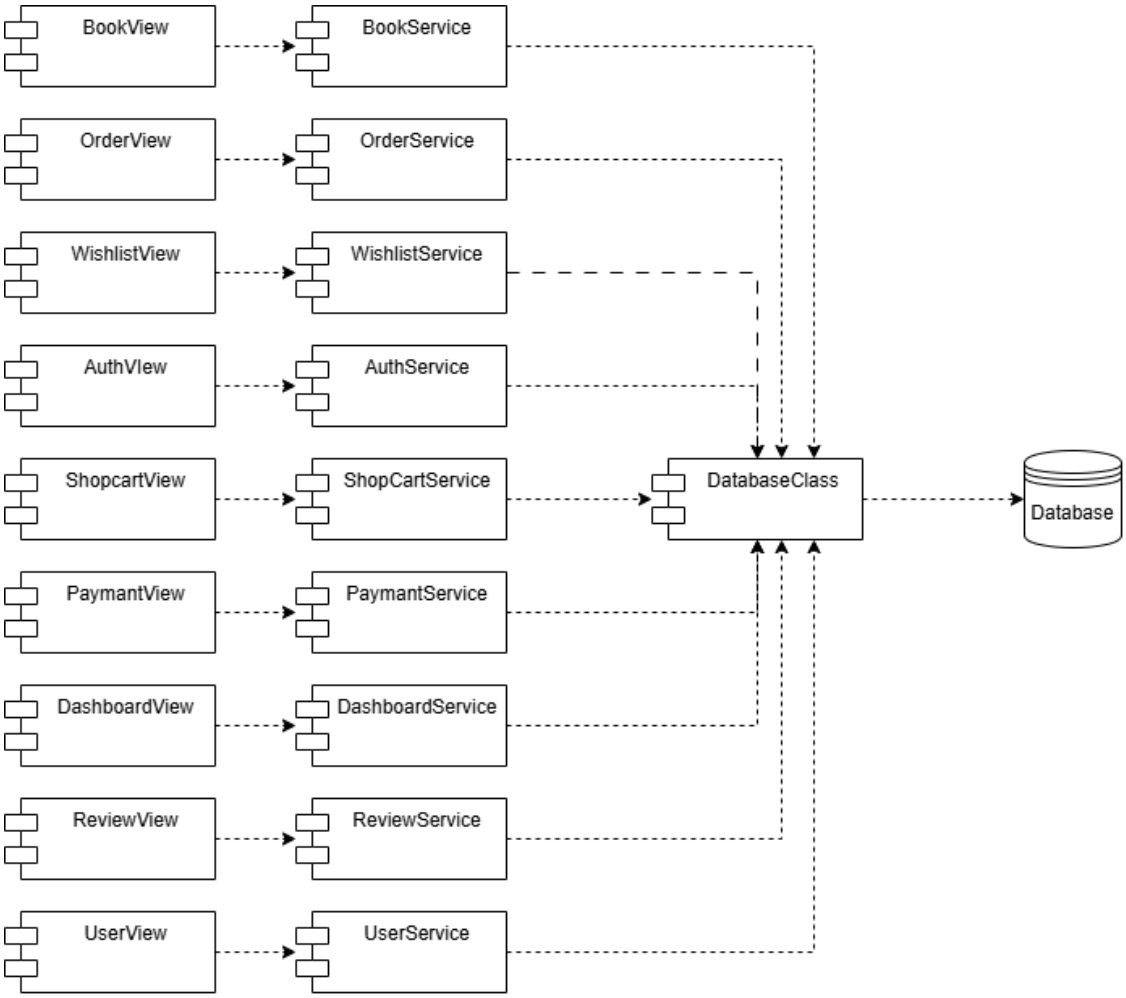

Рис. 2.5. Діаграма компонентів до проектованої системи

6.5. Діаграма пакетів до проектованої системи

На рисунку 2.6. наведено діаграму пакетів до проектованої системи електронного магазину книг. Пакет views представляє собою пакет файлів відображення, який використовує configs – пакет конфігурацій, та view logic – пакетом логіки відображень. В свою чергу view також залежить від services, пакету, який реалізовує та надає всю логіку для роботи системи, цей пакет реалізовує відповідні інтерфейси, що знаходяться в Interfaces. Сам вищеописаний пакет залежить від пакету доступу до бази даних, який в свою чергу залежить від самої бази даних.

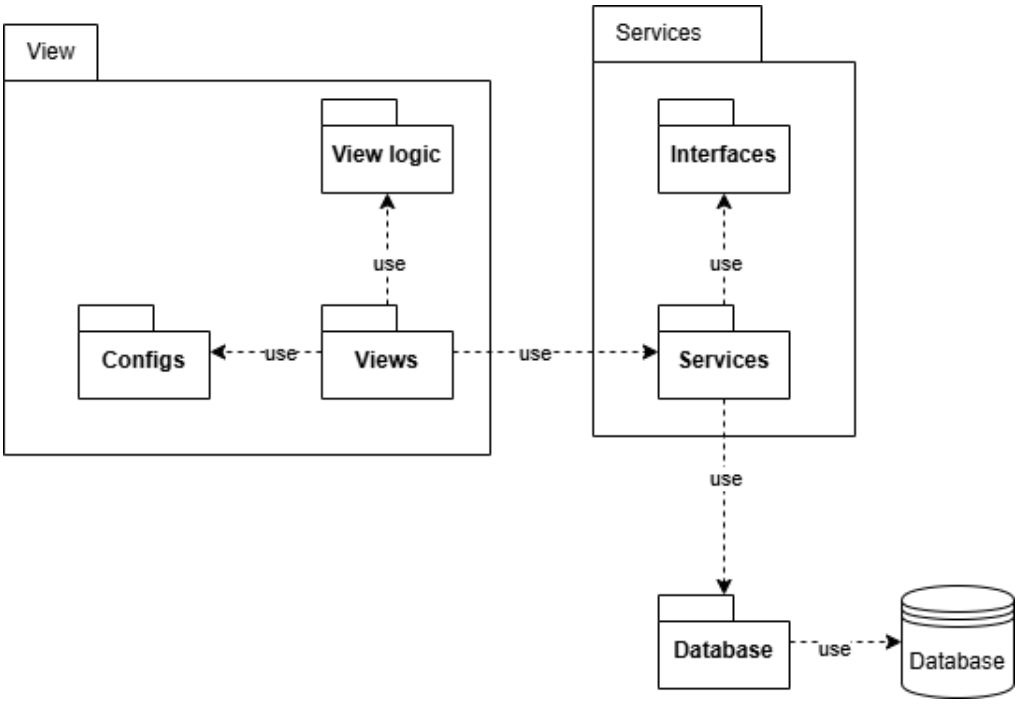

Рис. 2.6. Діаграма пакетів до проектованої системи

6.6. Висновки до розділу 2

Другий розділ зосереджено на процесі проєктування програмної системи, що охоплює два ключові аспекти: інформаційну та функціональну складові.

На початковому етапі було сформовано концептуальну модель даних у формі ER-діаграми, яка супроводжувалася ґрунтовним описом її структури та зв’язків. Аналіз показав, що розроблена модель відповідає вимогам реляційних баз даних і реалізована з урахуванням принципів третьої нормальної форми.

Далі здійснювалося моделювання програмної архітектури за допомогою візуальних засобів UML. Зокрема, було створено діаграми класів, кооперацій, пакетів, станів і компонентів. Побудова діаграми класів включала два етапи: спочатку визначалися основні зв’язки між сутностями, а згодом формувався повний варіант зі всіма атрибутами та методами. Діаграма пакетів дозволила логічно згрупувати елементи системи, надаючи загальне уявлення про її модульну структуру. Діаграма компонентів, у свою чергу, відображала фізичну реалізацію модулів та взаємозв’язки між ними на рівні розгортання.

РОЗДІЛ 3

7.1. Система управління інформаційною базою

У процесі розробки програмного забезпечення важливо обрати ефективну та надійну систему управління базами даних (СУБД), яка забезпечить збереження, обробку та захист інформації.

«Системи управління базами даних (СУБД) − це програмні комплекси, призначені для роботи зі спеціально організованими файлами, які називаються базами даних» [4].

У межах даного проєкту як основну СУБД було обрано PostgreSQL, яка є однією з найпотужніших систем з відкритим вихідним кодом.

PostgreSQL — це об’єктно-реляційна система управління базами даних з відкритим вихідним кодом, яка підтримує більшість стандартів SQL та надає розширені можливості для роботи з даними. Розробка цієї СУБД ведеться з 1986 року на базі Каліфорнійського університету в Берклі, що робить її однією з найстаріших систем у своєму класі. PostgreSQL активно розвивається міжнародною спільнотою, що гарантує її актуальність, безпеку та відповідність сучасним вимогам програмної інженерії.

PostgreSQL підтримує зберігання та обробку структурованих даних з використанням як реляційного підходу (таблиці, зв’язки), так і об’єктно-орієнтованих розширень, таких як типи, функції, оператори та тригери. Система дозволяє створювати складні логічні структури даних та забезпечує високий рівень цілісності інформації.

PostgreSQL володіє широким спектром функціональних можливостей, серед яких:

- **Підтримка транзакцій** з дотриманням властивостей ACID (атомарність, узгодженість, ізольованість, довговічність);
- **Потужний механізм тригерів, представлень, процедур і функцій**, що дозволяє реалізовувати складну бізнес-логіку на стороні бази даних;
- **Підтримка розширюваності** — можливість створення власних типів даних, індексів, мов запитів;

- **Індексація** — B-дерева, хеш, GiST, GIN та BRIN-індекси для оптимізації пошуку;
- **Масштабованість** — можливість обробки великих обсягів даних, підтримка паралельного виконання запитів;
- **Реплікація** — синхронна та асинхронна реплікація для створення резервних копій і розвантаження навантаження;
- **Безпека** — гнучка система прав доступу, автентифікація за різними протоколами, включно з Kerberos, LDAP, SSL тощо.

У порівнянні з іншими популярними системами, такими як MySQL, SQLite або Oracle, PostgreSQL вирізняється низкою переваг, що стали вирішальними при її виборі:

1. **Повна підтримка стандарту SQL** — на відміну від MySQL, PostgreSQL забезпечує сувору відповідність стандарту SQL, що полегшує перенесення додатків та інтеграцію з іншими системами.
2. **Розширюваність та відкритість** — система має відкритий код, активну спільноту та дозволяє розширювати функціональність за рахунок модулів і розширень (наприклад, PostGIS для роботи з геоданими).
3. **Висока надійність і відмовостійкість** — PostgreSQL широко використовується у фінансовому секторі, телекомунікаціях та інших критичних сферах.
4. **Гнучка система прав доступу** — порівняно з MySQL, PostgreSQL має більш глибокий і контрольований механізм управління доступом.
5. **Масштабованість і продуктивність** — забезпечується обробка великих обсягів даних і складних запитів без значних втрат у швидкодії, завдяки паралельним запитам, оптимізованому плануванню та сучасним механізмам кешування.

Важливо відзначити, що PostgreSQL активно підтримується у хмарних рішеннях, таких як Amazon RDS, Google Cloud SQL, Heroku тощо, що спрощує розгортання додатків у хмарному середовищі.

7.2. Вибір інструментарію для створення прикладного програмного забезпечення

В якості середовища для розробки використовується Visual Studio та мову програмування C#, це IDE від компанії Microsoft, яка надає весь необхідний функціонал для створення програм різного рівня.

За сервер бази даних відповідає Postgres, це зручна та надійна база даних, яка надає різноманітні функції, яких немає у аналогів. Також вона має відкритий код і безкоштовну версію, яка повністю задовольняє більшість потреб.

Для зв'язку з базою даних використовується Entity Framework Core. Entity Framework Core - це фреймворк для розробки програмного забезпечення, який забезпечує зв'язок між даними в додатку та базою даних. Він є частиною .NET Core і дозволяє легко і зручно працювати з реляційними базами даних. Це потужний інструмент для забезпечення зв'язку з базою даних, який дозволяє легко та швидко розробляти додатки.

Для розробки серверної частини застосовується фреймворк Asp.Net core. Створений Microsoft для розробки сучасних застосунків які працюють з інтернетом. До переваг даного фреймворку відносять кросплатформенність та високу продуктивність.

Клієнтська частина використовує такі технології як HTML, CSS, JavaScript. Вони працюють разом і дозволяють створювати красиві, зручні та динамічні веб сторінки. HTML відповідає за розмітку сторінки, CSS – за зовнішній вигляд та стилі, JavaScript – за поведінку та динаміку.

Як підмножину мови JavaScript клієнтська сторона буде використовувати бібліотеку React. React – це відкрита бібліотека, створена компанією Facebook, яка швидко набрала популярність завдяки своїй зручності. Головними концепціями React JS є компоненти та стани. JS React базується на компонентній архітектурі, що дозволяє розбивати інтерфейс на незалежні компоненти. Це спрощує розробку, та підтримку коду, оскільки компоненти можуть бути повторно використані та легко замінюються.

Загалом використання цієї технології спрощує створення сайтів, полегшує роботу з компонентами, та забезпечує зручний та надійний досвід програмування.

7.3. Розробка інформаційної бази

Як було зазначено раніше, для роботи з базою даних використовується EF. Це ORM, яка дозволяє взаємодіяти з об’єктами C#, як з таблицями в базі даних, не занурюючи програміста в деталі реалізації SQL коду.

Сама бібліотека підтримує декілька підходів до створення баз даних:

1. Code First.
2. Database-Firs

Перший підхід має на увазі те, що спочатку створюються звичайні C# класи, які і будуть відображати відповідні моделі в базі даних. При другому підході спочатку створюється база даних, а потім відбувається генерація C# класів на їх основі. Що стосується даного проекту – буде використовуватися підхід Code First.

Першим кроком для роботи щ бібліотекою її необхідно завантажити в менеджері пакетів nuget. Наступним кроком створюються самі моделі. Як уже було зазначено раніше – моделі представляють собою звичайні класи. Наприклад, на рисунку 3.1. зображено код класу для моделі «Видавник».

```
public class Publisher
{
    public Guid Id { get; set; }
    3 usages
    public string Title { get; set; } = null!;

    public List<Book>? Books { get; set; }
}
```

Рис. 3.1. Код для моделі «Видавник»

Варто зауважити, що кожна модель, яка буде відображатися в базі даних, повинна мати первинний ключ. Це досягається двома шляхами,: встановлення назви властивості, як Id, або за допомогою атрибуту «Key».

Варто зауважити, що якщо модель має зв’язок має з іншою моделлю, це також повинно відображатися в проектованих класах. Наприклад, якщо модель «Книга» має зв’язок з «Автором» «один-до-одного», то в моделі книги повинно бути два додаткових поля: вторинний ключ з назвою, яка закінчується на Id та полем навігації, того типу, з яким є зв’язок. На рисунку 3.2. представлено фрагмент коду де можна помітити ці поля, для зв’язку з таблицю авторів.

```
public class Book
{
    6 usages
    public Guid Id { get; set; }
    5 usages
    public string Title { get; set; } = null!;

    //author foreign key
    1 usage
    public Guid AuthorId { get; set; }
    10 usages
    public Author Author { get; set; } = null!;
```

Рис. 3.2. Фрагмент коду з використанням зв’язків між моделями

Таким чином створюються всі інші моделі для бази даних. Після створення необхідно їх налаштувати, встановити обмеження, тощо. Для цього необхідно створити спеціальний клас, в якому буде відбуватися налаштування за допомогою класу EntityTypeBuilder, який надається бібліотекою EF. На рисунку 3.3. наведено фрагмент коду налаштування сутності «Книга». В даному коді відбувається обмеження довжини для назви, опису та поля під назвою ISBNю Також нижче відбувається налаштування ключів для проміжної таблиці.

```
public class BookConfiguration: IEntityTypeConfiguration<Book>
{
    Tenshi AL * More...
    public void Configure(EntityTypeBuilder<Book> builder)
    {
        builder.Property(p :Book => p.Title)
            .HasMaxLength(50);
        builder.Property(p :Book =>p.ISBN)
            .HasMaxLength(50);
        builder.Property(p :Book => p.Description)
            .HasColumnType("text");


        builder.HasMany(p :Book => p.Jenres) // CollectionNavigationBuilder<Book,Genre>
            .WithMany(p :Genre => p.Books) // CollectionCollectionBuilder<Genre,Book>
            .UsingEntity<BookGenre>();
    }
}
```

Рис. 3.3. Фрагмент коду з налаштуванням сутності «Книга»

Наступним кроком при розробці бази даних для проектованої системи необхідно створити клас, який буде взаємодіяти з базою даних. Клас, що наслідується від DbContext, містить DbSet<T> для кожної сутності. Саме через ці сутності і будуть відбуватися всі операції з даними, такі як: читання, додавання, редагування та видалення даних. На рисунку 3.4. наведено фрагмент коду класу-контексту бази даних, який містить в якості полів, всі типи створені для бази даних. Варто зауважити, всі перечисленні поля в цьому класі будуть створені в базі даних в якості таблиць з конфігураціями.

```
public DbSet<Book> Books { get; set; }= null!;
5 usages
public DbSet<Genre> Genres { get; set; } = null!;
1 usage
public DbSet<Language> Languages { get; set; }= null!;
1 usage
public DbSet<Format> Formats { get; set; }= null!;
3 usages
public DbSet<Publisher> Publishers { get; set; }= null!;
3 usages
public DbSet<Author> Authors { get; set; }= null!;
public DbSet<BookGenre> BookJenres { get; set; }= null!;
5 usages
public DbSet<WishList> WishLists { get; set; } = null!;
7 usages
public DbSet<Review> Reviews { get; set; } = null!;
6 usages
public DbSet<Order> Orders { get; set; }
7 usages
public DbSet<ShopCardItems> ShopCartItems { get; set; } = null;
3 usages
public DbSet<UserLibraries> UserLibraries { get; set; }
```

Рис. 3.4. Фрагмент коду класу-контексту бази даних

Ще однією задачею цього класу є фіксація налаштувань моделей. Класи, за допомогою яких відбувалася конфігурація моделей, викликаються в спеціальному методі «OnModelCreating», таким чином, фіксуючи налаштування. На рисунку 3.5. представлено код цього методу.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new BookConfiguration());
    modelBuilder.ApplyConfiguration(new FormatConfiguration());
    modelBuilder.ApplyConfiguration(new LanguageConfiguration());
    modelBuilder.ApplyConfiguration(new JenreConfiguration());
    modelBuilder.ApplyConfiguration(new IdentityRoleConfiguration());
    modelBuilder.ApplyConfiguration(new WishListConfiguration());

    base.OnModelCreating(modelBuilder);
}
```

Рис. 3.5. Код методу для налаштування конфігурацій моделей

Згідно з правилами фреймворку, клас-контексту бази даних має мати два конструктори: без параметрів та з параметрами. Це необхідно для внутрішніх налаштувань бібліотеки. Також в цих конструкторах викликається метод, який створює базу даних при першому запуску програми. Код конструкторів приведено на рисунку 3.6.

```
public EbookContext()
{
    Database.EnsureCreated();
}
Tenshi AL
public EbookContext(DbContextOptions options) : base(options)
{
    Database.EnsureCreated();
}
```

Рис. 3.6. Код конструкторів класу-контексту бази даних

Таким чином, база даних буде створена бід час першого виклику програми, але необхідні ще деякі налаштування, а саме налаштування підключення до БД. Рядок підключення до бази даних зберігається в спеціальному файлі налаштувань під назвою «appsettings.json». Код цього файлу з рядком підключення наведено на рисунку 3.7.

```
"ConnectionStrings": {
  "DefaultConnection": "Host=pg-aba5a75-ebook-d624.a.aivencloud.com;Port=26594;Database=defaultdb;Username=avnadmin;Password=AVNS_6FSm7hn-BbjLjUByx-R"
},
```

Рис. 3.7. Рядок підключення до бази даних

Рядок підключення вказується в спеціальному методі, який і створює екземпляр класу-контексту бази даних в системі. Це простий метод, код якого приведено на рисунку 3.8.

```
builder.Services.AddDbContext<EbookContext>(options =>
    options.UseNpgsql(builder.Configuration.GetConnectionString( name: "DefaultConnection")));
```

Рис. 3.8. Код створення класу-контексту бази даних

Таким чином, створення бази даних в системі завершено. Як уже було сказано раніше, при першому запуску буде створено базу даних з моделями та їх конфігураціями.

7.4. Алгоритмізація та програмування програмних модулів

Під час алгоритмізації та програмування програмних модулів буде відбуватися розбір та кодування алгоритмів бізнес-логіки системи. Спочатку доречно розглянути алгоритми авторизація та аутентифікації в системі.

Аутентифікація — це процес встановлення істинності особи, що звертається до інформаційної системи, шляхом перевірки її облікових даних або інших факторів ідентифікації. Вона забезпечує підтвердження того, що суб’єкт доступу є саме тим, за кого себе видає, з використанням паролів, біометричних характеристик, токенів чи багатофакторних механізмів перевірки.

Авторизація — це процедура надання користувачу певного обсягу прав доступу до ресурсів або функціональних можливостей інформаційної системи після успішної аутентифікації. Вона визначає рівень дозволених дій на основі ролей, політик безпеки або інших параметрів, що регулюють доступ до системних об’єктів.

На рисунку 3.9. зображено блок-схему алгоритму реєстрації нового користувача в системі. Як можна побачити з блок-схеми, першим кроком відбувається введення вхідних даних, а саме: ім’я, прізвищ, по-батькові, пароль та email. В разі, якщо такий користувач ще не зареєстрований та передані дані валідні відбувається реєстрація нового користувача в системі. Якщо ж дані було пошкоджено, чи користувач вже зареєстрований, реєстрація не відбудеться, згенерується відповідне повідомлення про помилку і алгоритм закінчиться.

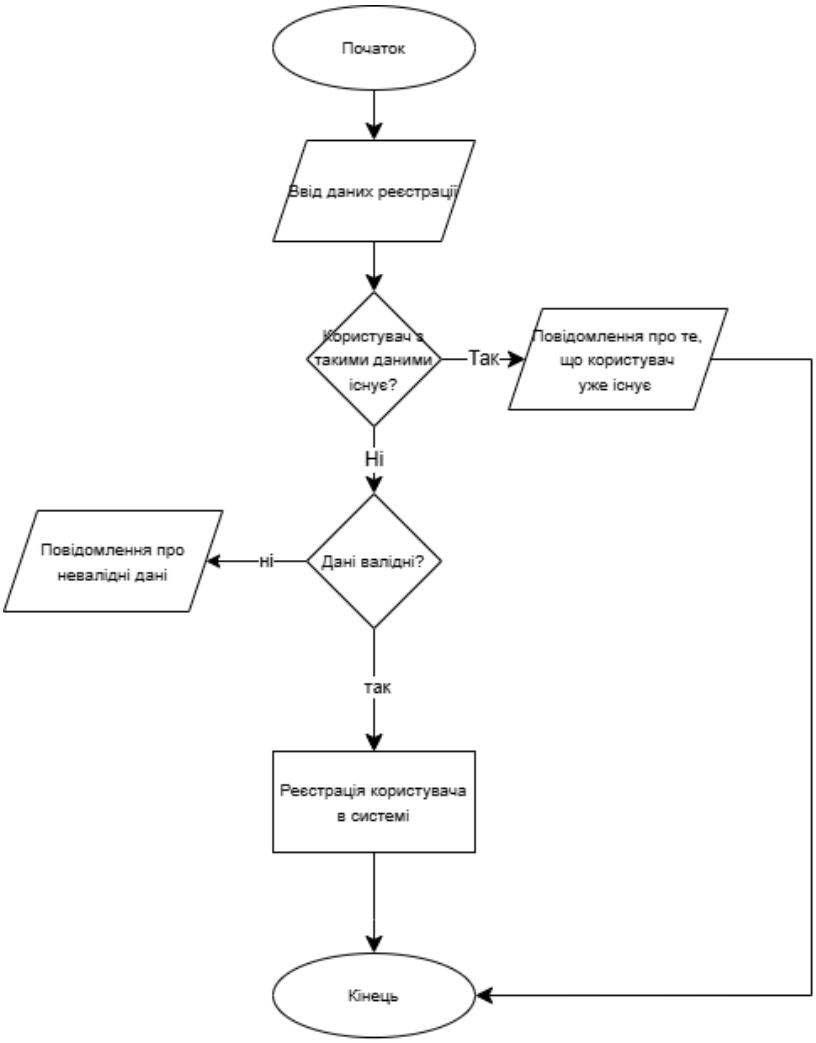

Рис. 3.9. Блок-схема алгоритму реєстрації користувача в системі

Для створення коду бізнес логіки використовується патерн «Команда». Патерн команда – це поведінковий патерн, задача якого перетворювати запити на об’єкти і передавати їх, як аргументи. Також цей шаблон забезпечує модульність системи, адже кожна команда має свою зону відповідальності. На рисунку 3.10 наведено код класу команди реєстрації нового користувача в системі. Можна помітити, що код класу відразу містить всі необхідні параметри для виконання команда. Сама команда створює нового користувача, та повертає результат створення.

```
public class RegisterUserCommand: IRequest<IdentityResult>
{
    public string Email { get; set; } = null!;
    public string UserName { get; set; } = null!;
    public string Name { get; set; } = null!;
    public string Surname { get; set; } = null!;
    public string Password { get; set; } = null!;

    public class RegisterUserCommandHandler: IRequestHandler<RegisterUserCommand,IdentityResult>
    {
        public RegisterUserCommandHandler(UserManager<User> userManager, RoleManager<IdentityRole<Guid>> roleManager)
        {
            _userManager = userManager;
            _roleManager = roleManager;
        }
        private readonly UserManager<User> _userManager;
        private readonly RoleManager<IdentityRole<Guid>> _roleManager;

        public async Task<IdentityResult> Handle(RegisterUserCommand request, CancellationToken cancellationToken)
        {
            User newUser = new User()
            {
                Email = request.Email,
                UserName = request.UserName,
                Name = request.Name,
                Surname = request.Surname,
                FullName = $"{request.Name} {request.Surname}"
            };

            var result = await _userManager.CreateAsync(newUser, request.Password);

            await _userManager.AddToRoleAsync(newUser, role: "User");

            return result;
        }
    }
}
```

Рис. 3.10. Код команди реєстрації нового користувача в системі

Після реєстрації доречно розглянути алгоритм авторизації та його реалізацію. Блок-схема алгоритму авторизації наведена на рисунку 3.11. Після введення вхідних даних відбувається перевірка їх валідності. В разі успішної перевірки валідності відбувається перевірка наявності користувача з такими даними в системі. Якщо остання перевірка пройшла успішно відбувається авторизація користувача в системі і генерується відповідне повідомлення. В разі виникнення помилки на будь-якому етапі виконання алгоритму генерується повідомлення про помилку і алгоритм закінчується.

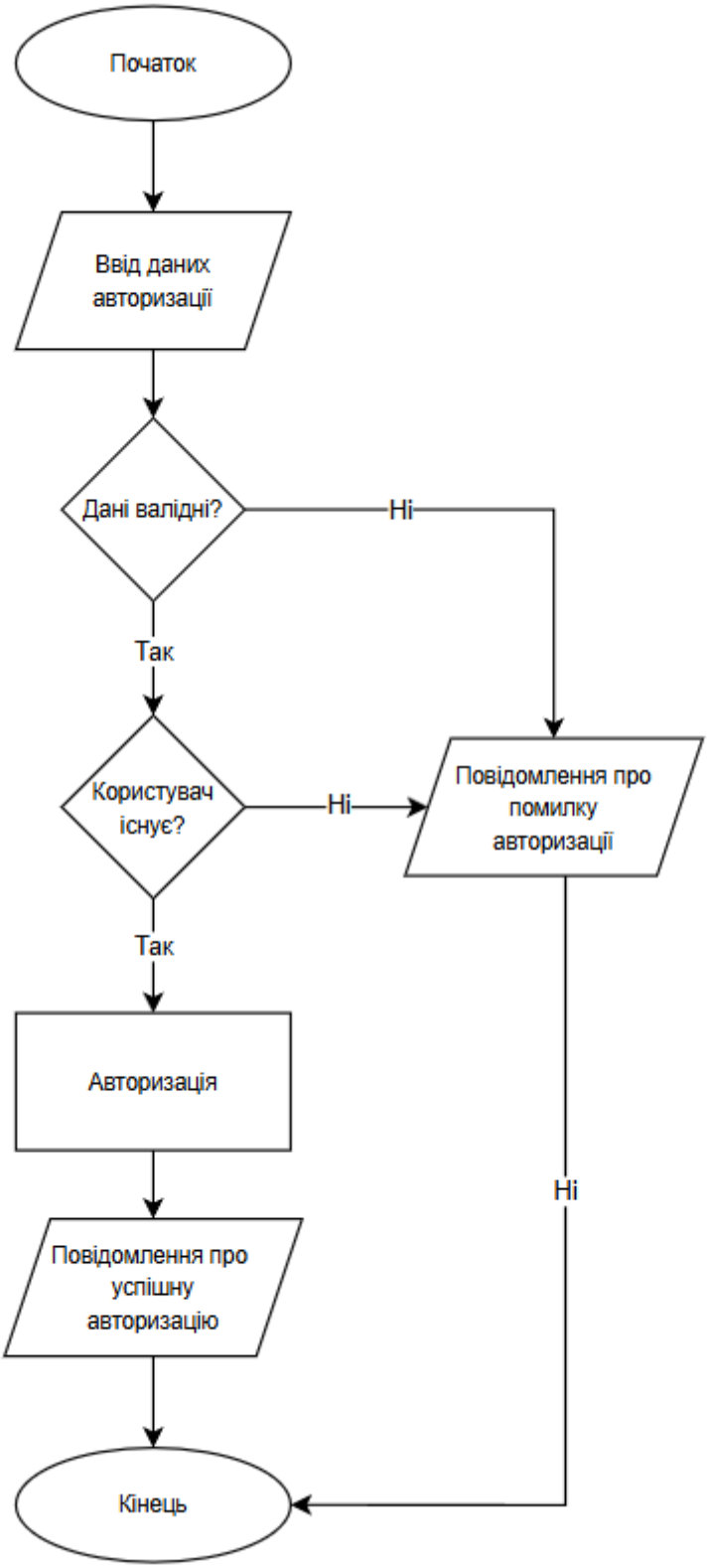

Рис. 3.11. Блок-схема алгоритму авторизації

Реалізація авторизації, як і в попередньому випадку, виконується з використанням шаблону проектування «Команда». Реалізацію класу-команди, який забезпечує авторизацію надано на рисунку 3.12.

```
public class LoginUserCommand: IRequest<Result<Request>>
{
    public string Email { get; set; } = null!;
    public string Password { get; set; } = null!;

    public class LoginUserCommandHandler : IRequestHandler<LoginUserCommand, Result<Request>>
    {
        public LoginUserCommandHandler(UserManager<User> userManager, IConfiguration configuration)
        {
            _userManager = userManager;
            _configuration = configuration;
        }
        private readonly UserManager<User> _userManager;
        private readonly IConfiguration _configuration;

        public async Task<Result<Request>> Handle(LoginUserCommand request, CancellationToken cancellationToken)
        {
            //check user is exists
            var user = await _userManager.FindByEmailAsync(request.Email);

            if (user is not null && await _userManager.CheckPasswordAsync(user, request.Password))
            {
                //get user role
                var roles IList<string> = await _userManager.GetRolesAsync(user);

                var claims = new List<Claim>()
                {
                    new Claim( type: ClaimTypes.Email, request.Email),
                    new Claim( type: JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
                };

                foreach (var role string in roles)
                {
                    claims.Add(new Claim( type: ClaimTypes.Role, role));
                }

                //get token
                var signingKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["JWT:Secret"]));
                var token = new JwtSecurityToken(
                    issuer: _configuration["JWT:ValidIssuer"],
                    audience: _configuration["JWT:ValidAudience"],
                    expires: DateTime.Now.AddHours(int.Parse(_configuration["JWT:ValidTime"])),
                    claims: claims,
                    signingCredentials: new SigningCredentials(signingKey, algorithm: SecurityAlgorithms.HmacSha256)
                );

                return Result.Ok(new Request()
                {
                    Token = token,
                    Roles = roles,
                    User = user,
                });
            }

            return Result.Fail("Login error");
        }
    }
```

Рис. 3.12. Реалізація класу для забезпечення авторизації

На рисунку 3.13. зображено блок-схему алгоритму для написання коментаря до книги. Користувач пише відгук, далі система перевіряє чи він має право залишати відгук та валідність тексту. В разі успіху система фіксує відгук за книгою.

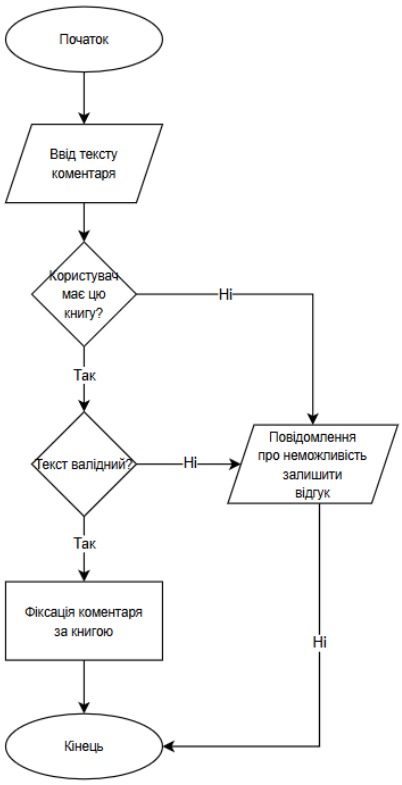

Рис. 3.13. Блок-схема алгоритму написання коментаря

На рисунку 3.14. зображено код реалізації цього алгоритму. Як і в попередніх випадках використовується шаблон «Команда».

```
public class InsertReviewCommand: IRequest
{
    public Guid BookId { get;set; }
    public Guid UserId { get; set; }
    public string? Text { get; set; }
    public int Score { get;set; }
    public DateTime Time { get; set; }

    public class InsertReviewCommandHandler : IRequestHandler<InsertReviewCommand>
    {
        public InsertReviewCommandHandler(EbookContext db)
        {
            _db=db;
        }
        private readonly EbookContext _db;
        public async Task Handle(InsertReviewCommand request, CancellationToken cancellationToken)
        {
            var review = new Review()
            {
                BookId = request.BookId,
                UserId = request.UserId,
                Text = request.Text,
                Score = request.Score,
                Time = request.Time
            };

            _db.Reviews.Add(review);
            await _db.SaveChangesAsync();
        }
    }
}
```

Рис. 3.14. Код реалізація додавання відгуку

Останнім кроком буде розглянуто алгоритм та його реалізацію для покупки нової книги. На рисунку 3.15. зображено блок-схему для цього алгоритму. Як можна помітити, першим кроком, користувач додає до корзини книгу. Якщо у користувача уже є така в наявності, кількість товару збільшується на одиницю. Далі користувач може продовжити шукати книги, або оформити замовлення. При оформленні замовлення система очікує на оплату та додає книгу до бібліотеки користувача, в разі, якщо вона пройшла.

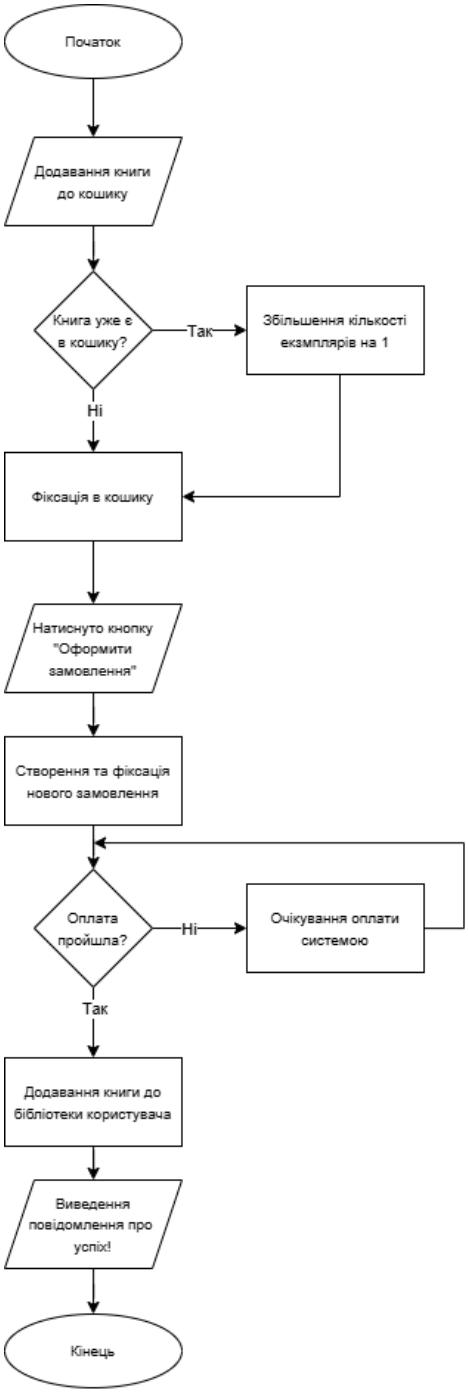

Рис. 3.15. Блок-схемам алгоритму придбання книги

Реалізація цього алгоритму наведена на рисунку 3.16. Важливим моментом є те, що тут є транзакція.

«Транзакція – це послідовність операцій над БД, розглянутих СУБД як єдине ціле. Або транзакція успішно виконується, і СУБД фіксує зміни БД, вироблені цією транзакцією, у зовнішній пам'яті, або жодна з цих змін ніяк не позначається на стані БД. Поняття транзакції необхідне для підтримки логічної цілісності БД» [4].

Першим кроком в транзакції виконується створення нового замовлення, далі, після оплати книга додається до бібліотеки користувача і корзина для покупок очищається. Якщо під час однієї бідь-якої операції виникла помилка, транзакція відміняється і дані в системі залишаються непошкоджені.

```
using var transaction = _db.Database.BeginTransaction();
try
{
    //make new order
    var books :List<Book> = await _db.Books.Where(p :Book => request.Books.Contains(p.Id)).ToListAsync(cancellationToken);
    var order = new Order()
    {
        UserId = request.UserId,
        TotalPrice = request.TotalPrice,
        Books = books,
        CreatedTime = DateTime.UtcNow
    };

    _db.Orders.Add(order);

    //add book to user lobrary
    foreach (var book in books)
    {
        _db.UserLibraries.Add(new UserLibraries()
        {
            UserId = request.UserId,
            BookId = book.Id
        });
    }

    //clear shop cart
    var shopCartItems :List<ShopCardItems> = await _db.ShopCartItems.Where(p :ShopCardItems => p.CartId == request.UserEmail) // IQueryable<ShopCardItems>
        .ToListAsync(cancellationToken); // Task<List<...>>

    foreach (var item :ShopCardItems in shopCartItems)
    {
        _db.ShopCartItems.Remove(item);
    }

    await _db.SaveChangesAsync(cancellationToken);
    transaction.Commit();
```

Рис. 3.16. Код транзакції в алгоритмі придбання книги

7.5. Висновки до розділу 3

Під час роботи над третім розділом було обрано СУБД для проектованої системи. Було зроблено висновок, що у процесі розробки програмного забезпечення важливо обрати ефективну та надійну систему управління базами даних (СУБД), яка забезпечить збереження, обробку та захист

інформації. У межах даного проекту було обрано PostgreSQL - потужну систему з відкритим вихідним кодом.

При виборі інструментарію для створення прикладного програмного забезпечення було обрано Visual Studio та мову програмування C#, це IDE від компанії Microsoft, яка надає весь необхідний функціонал для створення програм різного рівня. Для зв'язку з базою даних використовується Entity Framework Core. Entity Framework Core - це фреймворк для розробки програмного забезпечення, який забезпечує зв'язок між даними в додатку та базою даних. Для розробки серверної частини застосовується фреймворк Asp.Net core. Обрані технології відповідають сучасним стандартам до розробки програмнизх додатків,а також зможуть забезпечити реалізацію функціональних вимог до проектованої системи.

Після обрання СУБД та інструментарію для розробки ПЗ, наступним кроком було описано розробку інформаційної бази. Оскільки при розробці даної системи використовується ORM, розробка БД полягає в створення моделей та відповідних класів в мові програмування C#. Розділ 2.9. повністю описує весь процес, від створення та налаштування моделей, до реалізації класу-контексту бази даних, за допомогою якого і буде відбуватися взаємодія з БД в програмній системі.

Наступним кроком було описано та реалізовано деякі алгоритми бізнес-процесів в системі, а саме: реєстрація, авторизація, написання відгуку до книги та оформлення замовлення нової книги. Кожен алгоритм супроводжувався блок-схемою та кодом з реалізацією. Також варто зауважити, що в цьому підрозділі було реалізовано описано та реалізовано транзакцію, яка допомагає захистити дані від пошкоджень та зробити систему більш надійнішою в цілому.

РОЗДІЛ 4

8.1. Тестування системи

«Тестування програмного забезпечення (англ. Software Testing) – техніка контролю якості, що перевіряє відповідність між реальною і очікуваною поведінкою програми завдяки кінцевому набору тестів, які обираються певним чином. Техніка тестування також включає як процес пошуку помилок або інших дефектів, так і випробування програмних складових з метою оцінки» [5].

До основних типів тестування відносяться: модульне, інтеграційне, системне, навантажувальне. В межах даного проекту було обрано модульне тестування.

«Модульне (Компонентне) тестування (Unit testing – тестування елемента, блока) – це метод тестування програмного забезпечення, який полягає в окремому (ізольованому) тестуванні кожного мінімально можливого для тестування компоненту програми. Модулем називають найменшу частину програми, яка може бути протестованою. У процедурному програмуванні модулем вважають окрему функцію або процедуру, модулі програм. В об’єктно-орієнтованому програмуванні – інтерфейс, об’єкт, клас. Модульні тести, або unit-тести, розробляються в процесі розробки програмістами та, іноді, тестувальниками білої скриньки» [5].

Для успішного тестування було розроблено тестові випадки, які містили умови та очікуваний результат до роботи функції.

Функція додавання нових книг до інвентарю. Функція приймає на вхід дані книги, такі як назва, автор, кількість, ціна, ISBN, створює екземпляр та додає в базу даних, при некоректних даних видає помилку типу ModelValidationExceptions. В таблиці 4.1. наведено тест-кейси даної функції.

Таблиця 4.1. Тест-кейси функції додавання нових книг до інвентарю

Умови	**Очікуваний результат**
Всі дані коректні	Книгу успішно додано
ISBN продублювало двічі	Книгу не додано, помилка **ModelValidationExceptions**
Кількість менше нуля	Книгу не додано, помилка **ModelValidationExceptions**
Ціна менше нуля	Книгу не додано, помилка **ModelValidationExceptions**

У відповідності до тестових випадків було розроблено код тестування. Код наведено на рисунку 4.1.

```
[Theory(DisplayName = "Success add book with valid data")]
[InlineData("The Great Gatsby", "F. Scott Fitzgerald", "9780141182636", 10, 19.99)]
[InlineData("1984", "George Orwell", "9780451524935", 15, 12.99)]
[InlineData("To Kill a Mockingbird", "Harper Lee", "9780061120084", 8, 14.99)]
[InlineData("Pride and Prejudice", "Jane Austen", "9780141040349", 5, 9.99)]
[InlineData("The Hobbit", "J.R.R. Tolkien", "9780547928227", 12, 18.49)]
public async Task SuccessAddBookWithValidData(string name, string author, string ISBN, int
count, double price)
{
    //act
    int beforeCount = BookService.Count();
    await BookService.AddBook(name, author, ISBN, count, price);
    int afterCount = BookService.Count();

    //assert
    afterCount.ShouldBe(beforeCount+1);
}

[Theory(DisplayName = "Fail add book with invalid data")]
[InlineData("The Great Gatsby", "F. Scott Fitzgerald", "9780141182636", -5, -19.99)]
[InlineData("1984", "George Orwell", "9780451524935", -10, -12.99)]
[InlineData("To Kill a Mockingbird", "Harper Lee", "9780061120084", -1, -14.99)]
[InlineData("Pride and Prejudice", "Jane Austen", "9780141182636", -3, -9.99)]
[InlineData("The Hobbit", "J.R.R. Tolkien", "9780451524935", -8, -18.49)]
public async Task FailAddBookWithInvalidData(string name, string author, string ISBN, int
count, double price)
{
    //act
    int beforeCount = BookService.Count();
    await Should.ThrowAsync<ModelValidationExceptions>(BookService.AddBook(name, author, ISBN,
count, price));
    int afterCount = BookService.Count();

    //assert
    afterCount.ShouldBe(beforeCount);
}
```

Рис. 4.1. Код тестування функції додавання нових книг до інвентарю

Після виконання тестів цієї функції було отримано результати, які зображено на рисунку 4.2.

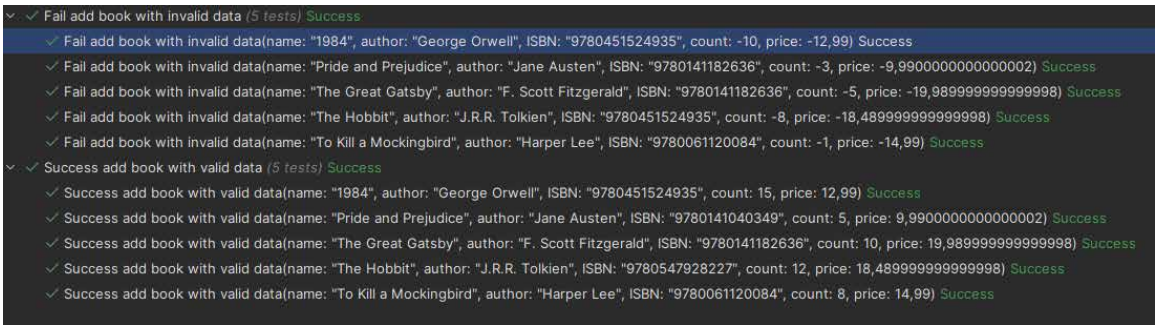

Рис. 4.2. Результат тестування функції додавання нових книг до інвентарю

Функція обробки покупок книг. Функція приймає ідентифікатори користувача та книги і кількість книг. Фіксує замовлення в базі даних та змінює кількість книг в замовленого екземпляра книги. В таблиці 4.2. наведено тест-кейси функції обробки покупки книг.

Таблиця 4.2. Тест кейси функції обробки покупки книг

Всі дані коректні	Замовлення зафіксовано в базі даних, а кількість екземплярів книг змінено
Некоректний ідентифікатор користувача	Замовлення не зафіксовано, кількість книг не змінено і відображено помилку типу **OrderValidationException**
Некоректний ідентифікатор книги	Замовлення не зафіксовано, кількість книг не змінено і відображено помилку типу **OrderValidationException**
Некоректна кількість книг для замовлення	Замовлення не зафіксовано, кількість книг не змінено і відображено помилку типу **OrderValidationException**

На рисунку 4.3. наведено код для тестів цієї функції.

```
[Fact(DisplayName = "Success make order with valid data")]
public async Task SuccessMakeOrderWithValidData()
{
    //arrange
    var user = DataBase.Users.FirstOrDefault(p => p.Email == "john.doe@example.com");
    var book = DataBase.Books.FirstOrDefault(p => p.Name == "The Great Gatsby");

    var validBookId = book.Id;
    var validUserId = user.Id;
    var validCount = 5;

    //act
    var orderCountBefore = OrderService.Count();
    var bookCountBeforeOrder = book.Count;
    await OrderService.MakeOrder(validUserId, validBookId, validCount);
    var orderCountAfter = OrderService.Count();
    var bookCountAfterOrder = DataBase.Books.FirstOrDefault(p => p.Name == "The Great
Gatsby").Count;

    //assert
    orderCountAfter.ShouldBe(orderCountBefore+1);
    bookCountAfterOrder.ShouldBe(bookCountBeforeOrder-validCount);
    bookCountAfterOrder.ShouldBeGreaterThan(0);
}

[Theory(DisplayName = "Fail make order with invalid data")]
[InlineData("00000000-0000-0000-0000-000000000000", "00000000-0000-0000-0000-000000000000",
-5)]
[InlineData("00000000-0000-0000-0000-000000000000", "f81d4fae-7dec-11d0-a765-00a0c91e6bf6",
-10)]
[InlineData("f81d4fae-7dec-11d0-a765-00a0c91e6bf6", "00000000-0000-0000-0000-000000000000",
-1)]
[InlineData("f81d4fae-7dec-11d0-a765-00a0c91e6bf6", "f81d4fae-7dec-11d0-a765-00a0c91e6bf6",
0)]
[InlineData("00000000-0000-0000-0000-000000000000", "00000000-0000-0000-0000-000000000000",
0)]
public async Task FailMakeOrderWithInvalidData(Guid userId, Guid bookId, int count)
{
    //act
    var orderCountBefore = OrderService.Count();
    Should.Throw<OrderValidationException>(OrderService.MakeOrder(userId, bookId, count));
    var orderCountAfter = OrderService.Count();

    //assert
    orderCountAfter.ShouldBeEquivalentTo(orderCountBefore);
}
```

Рис. 4.3. Код тестів для функції обробки покупки книг

Результати тестування функції обробки покупки книг наведено на рисунку 4.4.

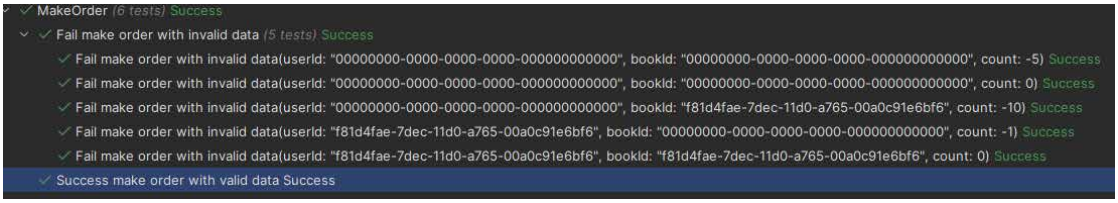

Рис. 4.4. Результати тестування функції обробки покупки книг

Функція відстеження замовлення клієнтів. Функція приймає на вхід ідентифікатор замовлення та повертає всю інформацію про замовлення. В таблиці 4.3. наведено тест-кейси для цієї функції.

Таблиця 4.3. Тест кейси функції відстеження замовлення

Умови	**Очікуваний результат**
Коректний ідентифікатор замовлення	Повернення замовлення
Некоректний ідентифікатор замовлення	Повернення **null**

Код для тестування функції відстеження замовлення наведено на рисунку 4.5.

```
[Fact(DisplayName = "Success get order with valid id")]
public async Task SuccessGetOrderWithValidID()
{
    //arrange
    var user = DataBase.Users.FirstOrDefault(p => p.Email == "jane.smith@example.com");
    var book = DataBase.Books.FirstOrDefault(p => p.Name == "1984");
    var orderId = await OrderService.MakeOrder(user!.Id, book!.Id, 5);

    //act
    var order = await OrderService.Get(orderId);

    //assert
    order.ShouldNotBeNull();
    order.UserId.ShouldBeEquivalentTo(user.Id);
    order.BookId.ShouldBeEquivalentTo(book.Id);
    order.Count.ShouldBe(5);
    order.Status.ShouldBe("В обробці");
}

[Fact(DisplayName = "Fail get order with invalid id")]
public async Task FailGetOrderWithInvalidId()
{
    //arrange
    var id = Guid.Empty;

    //act
    var order = await OrderService.Get(id);

    //assert
    order.ShouldBeNull();
}
```

Рис. 4.5. Код для тестування функції відстеження замовлення

Результати тестування функції відстеження замовлення клієнтів наведено на рисунку 4.6.

Рис. 4.6. Результати тестування функції відстеження замовлення клієнтів наведено на рисунку

Функція рейтингу книг. Користувач може оцінювати книги за шкалою від одного до 5. Функція повертає середню оцінку книги. В таблиці 4.4. наведено тест кейси для функції оцінювання книг.

Таблиця 4.4. Тест кейси функції оцінювання книг

Умови	**Очікуваний результат**
Всі дані коректні	Оцінка фіксується в базі даних, функція повертає середню оцінку.
Оцінка менше 1 чи більше 5	Функція повертає помилку типу **ReviewValidationException**
Некоректні ідентифікатори користувача чи книги	Функція повертає помилку типу **ReviewValidationException**

На рисунку 4.7. наведено код для тестування функції рейтингу книг, відповідно на рисунку 4.8. наведено результати тестування функції.

```
[Fact(DisplayName = "Success rating with valid data")]
public async Task SuccessRatingWithValidData()
{
    //arrange
    var userId1 = await UserService.AddUser("John", "Doe", "john.doe@example.com");
    var userId2 = await UserService.AddUser("Jane", "Smith", "jane.smith@example.com");
    var userId3 = await UserService.AddUser("Robert", "Johnson",
"robert.johnson@example.com");
    var bookId = await BookService.AddBook("The Great Gatsby", "F. Scott Fitzgerald",
"9780141182636", 10, 19.99);

    //act
    await BookService.MakeReview(userId1, bookId, 1);
    await BookService.MakeReview(userId2, bookId, 2);
    var averageRating = await BookService.MakeReview(userId3, bookId, 3);

    //assert
    averageRating.ShouldBe(2);
}

[Theory(DisplayName = "Fail rating with invalid score")]
[InlineData(0)]
[InlineData(-1)]
[InlineData(-3)]
[InlineData(6)]
[InlineData(999)]
public async Task FailRatingWithInvalidScore(int score)
{
    //arrange
    var ISBN = Guid.NewGuid();
    var userId = await UserService.AddUser("Emily", "Davis", "emily.davis@example.com");
    var bookId = await BookService.AddBook("1984", "George Orwell",ISBN.ToString() , 15,
12.99);

    //act
    await Should.ThrowAsync<ReviewValidationException>(BookService.MakeReview(userId, bookId,
score));
}

[Fact(DisplayName = "Fail rating with invalid book id")]
public async Task FailRatingWithInvalidBookId()
{
    //arrange
    var userId = await UserService.AddUser("Michael", "Brown", "michael.brown@example.com");
    var bookId = Guid.NewGuid();

    //assert
    await Should.ThrowAsync<ReviewValidationException>(BookService.MakeReview(userId, bookId,
5));
}

[Fact(DisplayName = "Fail rating with invalid user id")]
public async Task FailRatingWithInvalidUserId()
{
    //arrange
    var userId = Guid.NewGuid();
    var bookId = await BookService.AddBook("To Kill a Mockingbird", "Harper Lee",
"9780061120084", 8, 14.99);

    //assert
    await Should.ThrowAsync<ReviewValidationException>(BookService.MakeReview(userId, bookId,
5));
}
```

Рис. 4.7. Код для тестування функції рейтингу книг

Рис. 4.8. Результати тестування функції рейтингу книг.

Підсумовуючи, можна сказати, що у результаті проведеного функціонального тестування програмного забезпечення було перевірено основні сценарії використання системи відповідно до вимог, сформованих на етапі аналізу. Усі тестові кейси виконано успішно: помилок під час тестування не виявлено, система продемонструвала стабільну та передбачувану поведінку.

Проведені тести підтвердили відповідність реалізованого функціоналу заданим вимогам, а також дозволили переконатися в правильності обробки вхідних даних, коректності відображення результатів і належній реакції на помилки користувача. Таким чином, можна зробити висновок про працездатність системи та її готовність до подальшого використання.

8.2. Вимоги до апаратного та програмного забезпечення

Першим кроком під час аналізу вимог до апаратного та програмного забезпечення необхідно проаналізувати фізичні вузли системи. В цьому допоможе діаграма розгортання.

«Діаграма розгортання показує, яким чином ПЗ розгортається на обчислювальні елементи» [3].

На рисунку 4.9. продемонстровано діаграму розгортання до проектованої системи.

При розгортанні системи виділяється три основні компоненти: сервер клієнтської частини, сервер API і сервер бази даних (БД). Сервер клієнтської частини відповідає за обробку запитів від клієнтської сторони додатку. Він

надає веб-сервер для статичних ресурсів, таких як HTML, CSS та JavaScript, які використовуються клієнтами. Цей сервер має бути налаштованим для взаємодії з сервером API через HTTP-запити. Сервер API виконує роль проміжного програмного забезпечення, яке обробляє запити від клієнтів і надає їм доступ до ресурсів або функціоналу системи. У даній системі він взаємодіє з сервером бази даних для отримання або збереження даних, та з сервером користувача для повернення користувачеві даних (або результату операції). Сервер бази даних відповідає за управління та зберігання даних, які використовуються в системі. Сервер БД має бути надійним та відмовостійким. Також має бути функціонал резервного копіювання даних, що б дані користувачів у разі аварії не зникли.

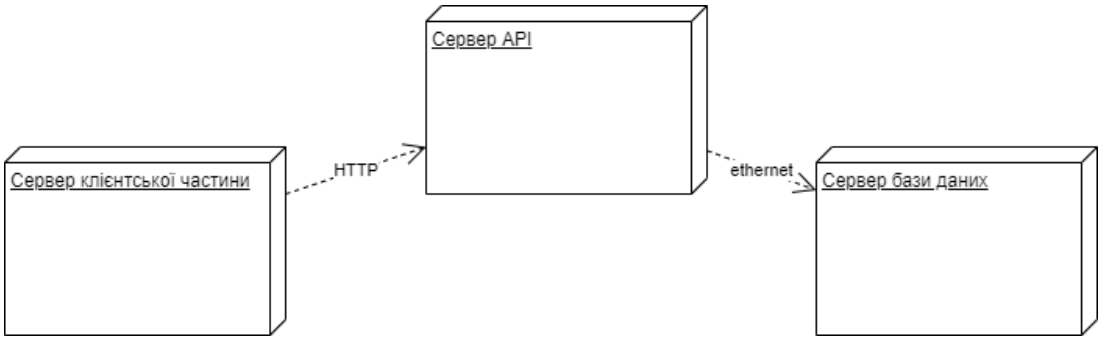

Рис. 4.9. Діаграма розгортання до проектованої системи

На основі аналізу розгортання можна описати вимоги до апаратного та програмного забезпечення для кожного вузла системи. Основні фізичні вузли включають сервер клієнтської частини, сервер API та сервер бази даних. Кожен з них має окремий набір характеристик, що забезпечують надійність, продуктивність та масштабованість програмного комплексу.

Сервер клієнтської частини має наступні вимоги:

Апаратні вимоги:

- Процесор: не менше ніж 2 ядра з частотою 2.0 ГГц або вище
- Оперативна пам’ять (RAM): 4 ГБ або більше
- Жорсткий диск: SSD-диск об'ємом не менше 40 ГБ
- Мережевий інтерфейс: стабільне з’єднання з пропускною здатністю не менше 100 Мбіт/с

Програмні вимоги:

- Операційна система: Ubuntu Server 20.04 LTS або Windows Server 2019
- Веб-сервер: Nginx 1.18+ або Apache 2.4+
- Node.js (для обслуговування клієнтських скриптів): версія 16.0 або вище
- Платформа розгортання: Docker (опціонально, для контейнеризації)

Що стосується вимог, до сервера API, варто зауважити, що сервер повинен забезпечувати швидке опрацювання запитів, масштабованість і стабільну роботу під навантаженням.

Апаратні вимоги:

- Процесор: не менше 4 ядер з частотою від 2.5 ГГц
- Оперативна пам’ять: 8 ГБ або більше
- Жорсткий диск: SSD-диск від 60 ГБ
- Мережевий інтерфейс: високошвидкісне з’єднання з мінімальною затримкою

Програмні вимоги:

- Операційна система: Ubuntu Server 22.04 LTS або Windows Server 2022
- Платформа: ASP.NET Core Runtime 7.0 або вище
- Середовище виконання: .NET SDK 7.0 або новіше
- Система управління процесами: systemd або IIS (залежно від ОС)
- Засоби логування: Serilog або аналогічні

Сервер бази даних зберігає всю інформацію, пов’язану з користувачами, проєктами, завданнями, ролями та іншими даними. Він має забезпечувати надійне зберігання, високу швидкодію запитів і захист даних від втрати.

Апаратні вимоги:

- Процесор: мінімум 4 ядра, оптимально 6 ядер і більше
- Оперативна пам’ять: 16 ГБ (для середнього навантаження)
- Жорсткий диск: SSD обсягом 100+ ГБ із підтримкою резервування (RAID-1)
- Додатково: резервне джерело живлення (UPS), система аварійного копіювання

Програмні вимоги:

- Операційна система: Ubuntu Server 22.04 або Windows Server 2022
- СУБД: PostgreSQL 14 або Microsoft SQL Server 2022
- Засоби резервного копіювання: pgBackRest (для PostgreSQL), SQL Server Agent або інші
- Фаєрвол: налаштований доступ лише для авторизованих серверів API

8.3. Склад інсталяційного пакету

Сервер клієнтської частини відповідає за розміщення і доставку статичних файлів фронтенду до користувачів. Його інсталяційний пакет має включати:

Компоненти:

- **Веб-сервер:**
  - Nginx 1.18+ або Apache 2.4+
- **Платформа обслуговування статичних файлів:**
  - Node.js (версія 16 або вище) — для збірки та обслуговування SPA-додатку (Single Page Application)
- **Системні утиліти:**
  - pm2 або аналог — для автоматичного запуску фронтенду
  - ufw або інший брандмауер — для обмеження доступу
- **Сертифікати HTTPS:**
  - SSL-сертифікат (наприклад, Let's Encrypt)
- **Фронтенд-додаток:**
  - Скомпільовані файли HTML, CSS, JavaScript (папка dist/ або build/)
- **Конфігураційні файли:**
  - nginx.conf або .htaccess — для маршрутизації запитів

API-сервер виконує бізнес-логіку, обробляє запити клієнтів і взаємодіє з базою даних. Склад його інсталяційного пакета включає:

Компоненти:

- **Середовище виконання:**

  - .NET SDK та ASP.NET Core Runtime (версія 7.0 або новіше)
- **Бібліотеки:**
  - Newtonsoft.Json, Serilog, AutoMapper, Entity Framework Core, JWT Tokens (NuGet-пакети)
- **Системні служби:**
  - systemd service unit або IIS application pool (в залежності від ОС)
- **Конфігураційні файли:**
  - appsettings.json, appsettings.Production.json — з параметрами бази даних, логуванням, секретами
  - .env або Secrets Manager — для чутливих даних (ключі токенів, підключення до БД)
- **API-додаток:**
  - Зібраний .dll-файл та залежності у вигляді publish-папки
- **Скрипти:**
  - Bash або PowerShell скрипти для розгортання/запуску
  - Health check endpoints для моніторингу

Сервер БД відповідає за зберігання всіх даних системи, тому потребує окремої конфігурації для безпеки, доступності та відновлення.

Компоненти:

- **СУБД:**
  - PostgreSQL 14+ або SQL Server 2022 (в залежності від вибору)
- **Інструменти резервного копіювання:**
  - pgBackRest або pg_dump для PostgreSQL
  - SQL Server Agent для Microsoft SQL
- **Схема бази даних:**
  - SQL-скрипти створення таблиць, індексів, ключів
  - Скрипти ініціалізації довідників (ролі, статуси, тощо)
- **Засоби моніторингу:**
  - pgAdmin або SQL Server Management Studio
- **Файли конфігурації:**

- postgresql.conf, pg_hba.conf — для налаштування доступу
- firewall-налаштування (доступ лише з IP API-сервера)

- **Резервні скрипти:**
  - Автоматичне створення бекапів (cron/Task Scheduler)

8.4. Висновки до розділу 4

Робота над останнім розділом складалася з наступних частин: тестування системи, визначення вимог до апаратного та програмного забезпечення, визначення складу інсталяційного пакету.

Було визначено, що таке тестування, які його типи існують, а також який було обрано для даного проекту. Для успішного тестування було розроблено тестові випадки, які містили умови та очікуваний результат до роботи функції. Для кожної функції було наведено тестові випадки, код тестів та їх результат. В кінці було зроблено висновок, що всі тести пройшли успішно, а отже можна бути впевненим в коректності відображення результатів і належній реакції на помилки користувача.

Після створення тестів відбувся етап визначення вимог до апаратного та програмного забезпечення. Було проаналізовано фізичні вузли системи та відображено їх взаємозв'язки в діаграмі розміщення. На онові цієї діаграми було визначено вимоги до кожного вузла системи: серверу клієнтського додатку, серверу API, та серверу бази даних. Останнім кроком було надано інструкції інсталяційного пакету.

ВИСНОВКИ

В першому розділі відбувся аналіз предметної області та програм аналогів. Було наведено дяку кількість програм-аналогів, та здійснено порівняльний аналіз. На їх основі та на основі аналізу потреб користувачів було визначено вимоги до системи. Бізнес-вимоги включали забезпечення зручного доступу до електронних книг для користувачів, зниження витрат на фізичне зберігання та доставку книг та створення ефективного каналу дистрибуції електронного контенту. В цьому ж розділі було проведено моделювання предметної області. Першим кроком під час моделювання було створено та описано основні абстракції системи. Далі було проаналізовано основних акторів та прецедентів і відображено їх на відповідних діаграмах. Останнім кроком було створено та продемонстровано діаграму послідовності, яка допомогла зрозуміти потік даних в системі.

На другому етапі роботи було збудовано логічну модель у вигляді ER діаграми, що відображала основні сутності інформаційного сервісу електронних книг, їх атрибути та взаємозв’язки між ними. На основі побудованої логічної моделі відбувалося подальше проектування. ER-діаграма дозволила виявити та структуровано представити логічну організацію даних системи, що стало основою для подальшого проєктування бази даних та реалізації моделі у системі управління базами даних (СУБД).

Було створено діаграми класів для розуміння їх взаємодії в межах проектованої системи. Також було створено діаграми пакетів, станів, компонентів, які дозволили глибше проаналізувати архітектуру системи, поведінку об'єктів та організацію її структурних елементів.

Діаграма пакетів надала уявлення про модульність програмного забезпечення, демонструючи розподіл класів по окремих логічних модулях — наприклад, аутентифікації, управління каталогом книг, обробки замовлень, роботи з базою даних, тощо. Це допомогло забезпечити структурованість та повторне використання коду.

Діаграми компонентів були застосовані для візуалізації фізичної структури програмного забезпечення — показано, які саме програмні компоненти реалізують функціональність системи, які інтерфейси вони надають та які залежності існують між ними.

Наступним кроком, перед створенням програмної системи було обрано СУБД та програмні інструменти для реалізації. В якості СУБД було обрано PostgreSQL, а для програмування логіки серверу використовувалася мова C# з використанням бібліотек Entity Framework Core та Asp.Net Core. Для створення інтересної частини системи було задіяно HTML, CSS, JS, React.Js.

Важливим етапом розробки було створення алгоритмів бізнес-логіки. В третьому розділі демонструються деякі алгоритми, такі як реєстрація, авторизація, замовлення книги. Кожен алгоритм супроводжується блок-схемою та реалізацією з поясненнями.

Перед введенням програми в експлуатацію, важливим кроком, було тестування. Для кожної функції було створено таблицю тестових випадків та відповідний код, який займався тестуванням. Було зроблено висновок, що успішне тестування системи допомагає покращити її надійність та відмово стійкість.

В останньому розділі було проаналізовано фізичні вузли для системи та відображено їх на відповідній діаграмі розгортання. На основі аналізу розгортання було визначено апаратні та програмні вимоги до кожного з вузлів та описано в повному об’єму. Останнім кроком було надано інструкцію до інсталяційного пакету.

Підсумовуючи, було створено програмну системи, яка задовольняє всім сучасним вимогам до створення програмного забезпечення, визначеним раніше функціональним та нефункціональним вимогам і має потенціал для розширення. Модульна архітектура проекту, заснована на шаблоні проектування «Команда» дозволить вводити новий функціонал в систему без значних затрат, що в майбутньому дозволить розширити функціонал системи при потребі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Козак О. Л. Опорний конспект лекцій з курсу «Аналіз вимог до програмного забезпечення» для студентів напрямку підготовки «Програмна інженерія» / О. Л. Козак. – Тернопіль, 2011. – 56 с.
2. Яшина О. В., Земляна С. В., Мозгова І. В. Мова об’єктно-орієнтованого проектування UML : навч. посіб. – Дніпро : РВВ ДДТУ, 2003. – 58 с.
3. Гаркуша І. М. Конспект лекцій з дисципліни «Проектування інформаційних систем» для студентів галузі знань 12 «Інформаційні технології» спеціальності 126 «Інформаційні системи та технології» / І. М. Гаркуша. – Дніпро : НТУ «ДП», 2020. – 75 с.
4. Доценко С. І. Організація та системи керування базами даних : навч. посіб. / С. І. Доценко. – Харків : УкрДУЗТ, 2023. – 117 с. : рис. 92, табл. 3.
5. Авраменко А. С., Авраменко В. С., Косенюк Г. В. Тестування програмного забезпечення : навч. посіб. – Черкаси : ЧНУ ім. Б. Хмельницького, 2017. – 284 с.
6. Коноваленко І.В. Платформа .NET та мова програмування C# 8.0: навчальний посібник / Коноваленко І.В., Марущак П.О. – Тернопіль: ФОП Паляниця В. А., 2020 – 320 с
7. Entity Framework Core 7. URL: https://abitap.com/1-1-entity-framework-core-6/
8. Freeman, A. Pro ASP.NET Core MVC 2 / Adam Freeman. – 7th ed. – New York: Apress, 2017. – 1017 p.
9. Duckett, J. HTML and CSS: Design and Build Websites / Jon Duckett. – Indianapolis: Wiley, 2011. – 490 p.
10. Freeman, E. Head First HTML and CSS / Elisabeth Freeman, Eric Freeman. – 2nd ed. – Sebastopol: O'Reilly Media, 2012. – 768 p.
11. Banks, A., & Porcello, E. Learning React: Functional Web Development with React and Redux / Alex Banks, Eve Porcello. – 3rd ed. – Sebastopol: O'Reilly Media, 2023. – 350 p.

Додаток А. Код серверної частини

```
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using Domain.Models;
using EBook.API.Models;
using IdentityServer4.Extensions;
using Infrastructure.CQRS.Account;
using MediatR;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

namespace EBook.API.Controllers;

[ApiController]
[Route("[controller]")]
public class AccountController: ControllerBase
{
   public AccountController(IMediator mediator, IHttpContextAccessor
httpContextAccessor, UserManager<User> userManager)
   {
      _mediator = mediator;
      _userManger = userManager;
      _httpContextAccessor = httpContextAccessor;
   }
   private readonly IMediator _mediator;
   private readonly IHttpContextAccessor _httpContextAccessor;
   private readonly UserManager<User> _userManger;
```

```
[HttpGet("currentUser")]
public async  Task<IActionResult> GetCurrentUser()
{

    if(!_httpContextAccessor.HttpContext.User.IsAuthenticated())
        return Unauthorized();

    var userEmail = _httpContextAccessor.HttpContext.User.Claims.First(c =>
c.Type == ClaimTypes.Email).Value;

    var user = await _userManger.FindByEmailAsync(userEmail);

    var roles = await _userManger.GetRolesAsync(user);

    return Ok(new
    {
        User = user,
        Roles = roles
    });
}

[HttpPost("register")]
public async Task<IActionResult> Register([FromBody]RegisterModel request)
{
    var result = await _mediator.Send(new RegisterUserCommand()
    {
        Email = request.Email,
        Name = request.Name,
        Surname = request.Surname,
        Password = request.Password,
```

```
            UserName = request.UserName,
        });

        return result.Succeeded ? Ok(result) : BadRequest(result);
    }

    [HttpPost("registerAdmin")]
    public async Task<IActionResult> AdminRegister([FromBody]RegisterModel request)
    {
        var result = await _mediator.Send(new AdminRegisterCommand()
        {
            Email = request.Email,
            Name = request.Name,
            Surname = request.Surname,
            Password = request.Password,
            UserName = request.UserName,
        });
        return result.Succeeded ? Ok(result) : BadRequest(result);
    }

    [HttpPost("login")]
    public async Task<IActionResult> Login([FromBody]LoginView request)
    {
        var result = await _mediator.Send(new LoginUserCommand()
        {
            Email = request.Email,
            Password = request.Password
        });
```

```
        return result.IsSuccess ? Ok(new
        {
            token = new JwtSecurityTokenHandler().WriteToken(result.Value.Token),
            roles = result.Value.Roles,
            expiration = result.Value.Token.ValidTo,
            user = result.Value.User
        }) : Unauthorized();
    }

}
[ApiController]
[Route("[controller]")]
[Produces("application/json")]
[DisableRequestSizeLimit]
public class BookController: ControllerBase
{
    public BookController(IMediator mediator, IHttpContextAccessor
httpContextAccessor, UserManager<User> userManager)
    {
        _mediator = mediator;
        _httpContextAccessor = httpContextAccessor;
        _userManager = userManager;
    }
    private readonly IMediator _mediator;
    private readonly IHttpContextAccessor _httpContextAccessor;
    private UserManager<User> _userManager;
    [HttpGet("getCurrentUserLibrary")]
    [Authorize]
    public async Task<IActionResult> GetCurrentUserLibrary()
    {
```

```
        var userEmail = _httpContextAccessor.HttpContext.User.Claims.First(c => c.Type == ClaimTypes.Email).Value;
        var user = await _userManager.FindByEmailAsync(userEmail);

        var library = await _mediator.Send((new GetUserLibraryQuery()
        {
            UserId = user.Id
        }));

        return Ok(library);
    }
    [HttpPut]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status400BadRequest)]
    public async Task<IActionResult> Put(Guid id, [FromForm]BookEditDTO book)
    {
        await _mediator.Send(new EditBookCommand()
        {
            BookId = id,
            BookEditDTO = book
        });
        return Ok();
    }
    [HttpGet("getBooks/page={page}/pageSize={pageSize}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    public async Task<IActionResult> Get(int page, int pageSize, [FromQuery]BookQueryModel bookQueryModel)
    {
        var allBooks = await _mediator.Send(new GetAllBooksQuery());
```

```
        var filteredBooks = await _mediator.Send(new FilterBookQuery()
        {
            Books = allBooks,
            Genres = bookQueryModel.Genres,
            Authors = bookQueryModel.Authors,
            MaxPrice = bookQueryModel.MaxPrice,
            MinPrice = bookQueryModel.MinPrice
        });

        var sortedBooks = await _mediator.Send(new SortBookQuery()
        {
            Books = filteredBooks,
            SortState = bookQueryModel.SortState
        });

        var response = new BookView()
        {
            Books = sortedBooks.Skip(pageSize * (page - 1)).Take(pageSize).ToList(),
//change to my pagination method
            Page = page,
            PageSize = pageSize,
            PageCount = (sortedBooks.Count + pageSize - 1) / pageSize,
            TotalCount = sortedBooks.Count,
            MinPrice = allBooks.Min(p=>p.Price),
            MaxPrice = allBooks.Max(p=>p.Price)
        };

        return Ok(response);
    }
```

```
[HttpGet("get")]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<IActionResult> Get(Guid id)
{
    var book = await _mediator.Send(new FirstOrDefaultBookQuery()
    {
        Query = p =>p.Id == id
    });

    return book is null ? NotFound() : Ok(book);
}

[HttpGet]
[ProducesResponseType(StatusCodes.Status200OK)]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public async Task<IActionResult> Get(string name)
{
    var books = await _mediator.Send(new SelectBookLikeQuery()
    {
        BookName = name
    });

    return books is null ? NotFound() : Ok(books);
}

[HttpPost]
[ProducesResponseType(StatusCodes.Status200OK)]
public async Task<IActionResult> ChangeAvailable(Guid bookId)
{
```

```
    await _mediator.Send(new ChangeAvailableBookCommand()
    {
        BookId = bookId
    });
    return Ok();
}

/// <summary>
/// Create new book
/// </summary>
/// <remarks>
[HttpPost("post")]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task<IActionResult> Post([FromForm]BookWriteDTO book)
{
    var directory = await _mediator.Send(new CreateDirectoryCommand()
    {
        BookName = book.Book.FileName,
        ImageName = book.Image.FileName
    });

    await _mediator.Send(new SaveFileCommand()
    {
        FilePath = directory.bookPath,
        File = book.Book
    });

    await _mediator.Send(new SaveFileCommand()
    {
```

```
            FilePath = directory.imagePath,
            File = book.Image
        });

        await _mediator.Send(new CreatePreviewBookCommand()
        {
            InputPdfPath = directory.bookPath,
            OutputPdfPath =
Path.Combine(Path.GetDirectoryName(directory.bookPath),$"{Path.GetFileName
WithoutExtension(directory.bookPath)}_Preview.pdf")
        });

        var id = await _mediator.Send(new InsertBookCommand()
        {
            BookWriteDto = book
        });

        return CreatedAtAction(nameof(Get), new {id = id},book);
    }
}
using Infrastructure.CQRS.DashboardCQRS;
using MediatR;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace EBook.API.Controllers;

[ApiController]
[Authorize]
[Route("[controller]")]
```

```
public class DashboardController: ControllerBase
{
    public DashboardController(IMediator mediator)
    {
        _mediator = mediator;
    }

    private readonly IMediator _mediator;

    [HttpGet]
    public async Task<IActionResult> Get()
    {
        var dashboard = await _mediator.Send(new GetDashboardQuery());
        return Ok(dashboard);
    }
}
using Microsoft.OpenApi.Models;
using System.Reflection;

namespace EBook.API.ServicesConfigure;

public static class SwaggerServiceExtension
{
    public static IServiceCollection SwaggerConfigure(this IServiceCollection
service)
    {
        service.AddSwaggerGen(options =>
        {
            options.SwaggerDoc("v1", new OpenApiInfo
            {
```

```
            Version = "v1",
            Title = "Ebook API",
            // Description = "Ebook shop",
            // TermsOfService = new Uri("https://example.com/terms"),
            // Contact = new OpenApiContact
            // {
            //     Name = "Example Contact",
            //     Url = new Uri("https://example.com/contact")
            // },
            // License = new OpenApiLicense
            // {
            //     Name = "Example License",
            //     Url = new Uri("https://example.com/license")
            // }
        });
        // using System.Reflection;
        var                                xmlFilename                                =
$"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
        options.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory,
xmlFilename));
    });
    return service;
  }
}
using Domain.Models;
using MediatR;
using Persistence;

namespace Infrastructure.CQRS.AuthorCQRS;
```

```
public class InsertAuthorCommand: IRequest<int>
{
   public string Name { get; set; } = null!;
   public string Surname { get; set; } = null!;

   public class InsertAuthorCommandHandler: IRequestHandler<InsertAuthorCommand,int>
   {
      public InsertAuthorCommandHandler(EbookContext db)
         => _db = db;
      private readonly EbookContext _db;
      public async Task<int> Handle(InsertAuthorCommand request, CancellationToken cancellationToken)
      {
         _db.Authors.Add(new Author()
         {
            Name = request.Name,
            Surname = request.Surname
         });

         return await _db.SaveChangesAsync(cancellationToken);
      }
   }
}
using AutoMapper;
using Infrastructure.DTO;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Persistence;
```

```
namespace Infrastructure.CQRS.AuthorCQRS;

public class SelectAuthorQuery: IRequest<List<AuthorReadDTO>>
{
   public class SelectAuthorQueryHandler: IRequestHandler<SelectAuthorQuery,List<AuthorReadDTO>>
   {
      public SelectAuthorQueryHandler(EbookContext db, IMapper mapper)
      {
         _db = db;
         _mapper = mapper;
      }
      private readonly EbookContext _db;
      private readonly IMapper _mapper;
      public async Task<List<AuthorReadDTO>> Handle(SelectAuthorQuery request, CancellationToken cancellationToken)
      {
         var list = await _db.Authors.ToListAsync(cancellationToken);
         return _mapper.Map<List<AuthorReadDTO>>(list);
      }
   }
}
using AutoMapper;
using Infrastructure.DTO;
using MediatR;
using Microsoft.EntityFrameworkCore;
using Persistence;

namespace Infrastructure.CQRS.LanguageCQRS;
```

```
public class SelectLanguagesListQuery: IRequest<List<LanguageDTO>>
{
   public class SelectLanguagesListQueryHandler : IRequestHandler<SelectLanguagesListQuery, List<LanguageDTO>>
   {
      public SelectLanguagesListQueryHandler(EbookContext db, IMapper mapper)
      {
         _db = db;
         _mapper = mapper;
      }
      private readonly EbookContext _db;
      private readonly IMapper _mapper;
      public async Task<List<LanguageDTO>> Handle(SelectLanguagesListQuery request, CancellationToken cancellationToken)
      {
         var languages = await _db.Languages.ToListAsync(cancellationToken);
         return _mapper.Map<List<LanguageDTO>>(languages);
      }
   }
}
```

ДОДАТОК Б. Код клієнтської частини

```
import axios from "axios"
import { getAuthors, getFormats, getGenres, getLanguages, getPublishers, postBookurl, server_url } from "../../constants/urls";

export const SET_AUTHORS = (payload) => {
    return{
        type: "SET_AUTHORS",
        payload
    }
}

export const SET_ERRORS = (payload) => {
    return {
        type: "SET_ERRORS",
        payload
    }
}

export const AXIOS_GET_AUTHORS = () => {
    return (dispatch)=>{
        axios.get(getAuthors)
            .then(responce => {
                dispatch(SET_AUTHORS(responce.data))
            })
            .catch(errors => {
                dispatch(SET_ERRORS(errors))
            })
    }
}
```

```
import axios from "axios"
import {postBookurl, server_url } from "../../constants/urls"

export const SET_ERRORS = (payload) => {
   return {
      type: "SET_ERRORS",
      payload
   }
}

export const SET_PAGE_SIZE = (payload) => {
   return {
      type: "SET_PAGE_SIZE",
      payload
   }
}

export const SET_PAGE_SETTINGS = ({page, pageSize}) => {
   return{
      type:"SET_PAGE_SETTINGS",
      payload: {page, pageSize}
   }
}

export const SET_BOOKS = (payload) => {
   return {
      type: "SET_BOOKS",
      payload
   }
}
```

```
export const AXIOS_GET_BOOKS = ({page, pageSize,authorsId,genresId,sortState, minPrice,maxPrice}) => {
    return(dispatch) => {

        let url = new URL(`http://localhost:5263/Book/getBooks/page=${page}/pageSize=${pageSize}`);

        if(authorsId){
            authorsId.forEach((authorId)=>url.searchParams.append('Authors', authorId));
        }
        if(genresId){
            genresId.forEach((genreId)=>url.searchParams.append('Genres', genreId));
        }


        if(minPrice & maxPrice){
            url.searchParams.append('MaxPrice',maxPrice);
            url.searchParams.append('MinPrice',minPrice);
        }

        if(sortState){
            url.searchParams.append('SortState',sortState);
        }

        axios.get(url)
            .then(responce => {
                dispatch(SET_BOOKS(responce.data))
            })
```

```
            .catch(errors => {
                dispatch(SET_ERRORS(errors))
            })
    }
}
export const SET_BOOKS_BY_NAME = (payload) => {
    return {
        type: "SET_BOOKS_BY_NAME",
        payload
    }
}
export const AXIOS_GET_BOOKS_BY_NAME = ({name}) => {
    return (dispatch) => {
        axios.get(`http://localhost:5263/Book?name=${name}`)
        .then(responce => {
            dispatch(SET_BOOKS_BY_NAME(responce.data))
        })
        .catch(errors => {
            dispatch(SET_ERRORS(errors))
        })
    }
}

export const SET_BOOK_BY_ID = (payload) => {
    return{
        type:'SET_BOOK_BY_ID',
        payload
    }
}
export const AXIOS_GET_BOOK_BY_ID = ({bookId}) => {
```

```
    let url = new URL('http://localhost:5263/Book/get');
    url.searchParams.append('id',bookId);

    return(dispatch)=>{
        axios.get(url)
            .then(responce => {
                dispatch(SET_BOOK_BY_ID(responce.data));
            })
            .catch(errors => {
                dispatch(SET_ERRORS(errors))
            });
    }
}

export const CREATE_BOOK = ()=> {
    return {
        type: "CREATE_BOOK",
    }
}
export const AXIOS_POST_BOOK = (book) => {
    return(dispatch)=>{
        axios.post(postBookurl,book)
            .then(() => {
                dispatch(CREATE_BOOK())
            })
            .catch(errors => {
                dispatch(SET_ERRORS(errors))
            });
    }
}
```

```
import axios from "axios"
import { getFormats } from "../../constants/urls"

export const SET_ERRORS = (payload) => {
    return {
        type: "SET_ERRORS",
        payload
    }
}



//get formats
export const SET_FORMATS = (payload) => {
    return {
        type: "SET_FORMATS",
        payload
    }
}

export const AXIOS_GET_FORMATS = () => {
    return(dispatch) => {
        axios.get(getFormats)
            .then(responce => {
                dispatch(SET_FORMATS(responce.data))
            })
            .catch(errors => {
                dispatch(SET_ERRORS(errors))
            })
    }
import axios from "axios"
```

```
export const SET_ERRORS = (payload) =>{
    return{
        type: 'SET_ERRORS',
        payload
    }
}

export const SUCCESS_REGISTER = () =>{
    return{
        type: 'SUCCESS_REGISTER'
    }
}


export const AXIOS_REGISTER_USER = (data) =>{
    return (dispatch) =>{
        axios.post('http://localhost:5263/Account/register', data)
        .then(()=>{
            dispatch(SUCCESS_REGISTER())
        })
        .catch(response=>{
            dispatch(SET_ERRORS(response.response.data.errors))
        })
    }
}
import { combineReducers, configureStore } from "@reduxjs/toolkit";
import { bookReducer } from "./reducers/bookReducer";
import { thunk } from "redux-thunk";
import { authorsReducer } from "./reducers/authorReducer";
```

```
import { formatsReducer } from "./reducers/formatsReducer";
import { genresReducer } from "./reducers/genresReducer";
import { languagesReducer } from "./reducers/languagesReducer";
import { publishersReducer } from "./reducers/publishersReducer";
import registrationReducer from "./reducers/registrationReducer";
import authenticationReducer from "./reducers/authenticationReducer";
import wishListReducer from "./reducers/wishListReducer";
import reviewReducer from "./reducers/reviewReducer";

const rootReducer = combineReducers({
   books:bookReducer,
   authors: authorsReducer,
   formats: formatsReducer,
   genres: genresReducer,
   languages: languagesReducer,
   publishers: publishersReducer,
   registration: registrationReducer,
   authentication: authenticationReducer,
   wishList: wishListReducer,
   review:reviewReducer
});
export const store = configureStore({reducer:rootReducer,
middleware:(getDefaultMiddleware) => getDefaultMiddleware({
   serializableCheck: false,
  }),});
let extensionsList = {};
extensionsList.image = [".jpg",".png"];
extensionsList.book = [".pdf"];

export function isValidFileType(fName, fType) {
```

```
    switch(fType){
        case "image":
            return
extensionsList.image.includes(fName.substring(fName.lastIndexOf("."))) ? true :
false;
        case "book":
            return
extensionsList.book.includes(fName.substring(fName.lastIndexOf("."))) ? true :
false;
    }
}
```